frontiers | Research Topics

# Robotics software engineering

**Edited by**
Ivano Malavolta, Federico Ciccozzi,
Christopher Timperley and Alwin Hoffmann

**Published in**
Frontiers in Robotics and AI

ISSN 1664-8714
ISBN 978-2-8325-6975-7
DOI 10.3389/978-2-8325-6975-7

October 2025

# Robotics software engineering

**Topic editors**
Ivano Malavolta — VU Amsterdam, Netherlands
Federico Ciccozzi — Mälardalen University, Sweden
Christopher Timperley — Carnegie Mellon University, United States
Alwin Hoffmann — XITASO GmbH, Germany

**Citation**
Malavolta, I., Ciccozzi, F., Timperley, C., Hoffmann, A., eds. (2025). *Robotics software engineering*. Lausanne: Frontiers Media SA. doi: 10.3389/978-2-8325-6975-7

*Dr Hoffmann is affiliated with XITASO GmbH IT & Software Solutions. All other Topic Editors declare no competing interests with regard to the Research Topic subject.*

## Table of contents

04 Editorial: Robotics software engineering
Federico Ciccozzi, Ivano Malavolta, Christopher Timperley, Andreas Angerer and Alwin Hoffmann

07 Architectural modelling for robotics: RoboArch and the CorteX example
Will Barnett, Ana Cavalcanti and Alvaro Miyazawa

27 A containerised approach for multiform robotic applications
Giuseppe Cotugno, Rafael Afonso Rodrigues, Graham Deacon and Jelizaveta Konstantinova

38 A survey of ontology-enabled processes for dependable robot autonomy
Esther Aguado, Virgilio Gomez, Miguel Hernando, Claudio Rossi and Ricardo Sanz

68 Composable and executable scenarios for simulation-based testing of mobile robots
Argentina Ortega, Samuel Parra, Sven Schneider and Nico Hochgeschwender

86 Software patterns and data structures for the runtime coordination of robots, with a focus on real-time execution performance
Maria I. Artigas, Rômulo T. Rodrigues, Lars Vanderseypen and Herman Bruyninckx

102 AAT4IRS: automated acceptance testing for industrial robotic systems
Marcela G. dos Santos, Sylvain Hallé, Fabio Petrillo and Yann-Gaël Guéhéneuc

115 EzSkiROS: enhancing robot skill composition with embedded DSL for early error detection
Momina Rizwan, Christoph Reichenbach, Ricardo Caldas, Matthias Mayr and Volker Krueger

133 Semantic composition of robotic solver algorithms on graph structures
Sven Schneider, Nico Hochgeschwender and Herman Bruyninckx

151 Energy efficiency in ROS communication: a comparison across programming languages and workloads
Michel Albonico, Manuela Bechara Cannizza and Andreas Wortmann

179 ROSA: a knowledge-based solution for robot self-adaptation
Gustavo Rezende Silva, Juliane Päßler, S. Lizeth Tapia Tarifa, Einar Broch Johnsen and Carlos Hernández Corbato

201 A model-based approach to automation of formal verification of ROS 2-based systems
Lukas Dust, Rong Gu, Saad Mubeen, Mikael Ekström and Cristina Seceleanu

frontiers | Frontiers in Robotics and AI

TYPE Editorial
PUBLISHED 29 September 2025
DOI 10.3389/frobt.2025.1686496

OPEN ACCESS

EDITED AND REVIEWED BY
Junaid Qadir,
Qatar University, Qatar

*CORRESPONDENCE
Federico Ciccozzi,
federico.ciccozzi@mdu.se
Ivano Malavolta,
i.malavolta@vu.nl

RECEIVED 15 August 2025
ACCEPTED 04 September 2025
PUBLISHED 29 September 2025

CITATION
Ciccozzi F, Malavolta I, Timperley C, Angerer A and Hoffmann A (2025) Editorial: Robotics software engineering.
*Front. Robot. AI* 12:1686496.
doi: 10.3389/frobt.2025.1686496

# Editorial: Robotics software engineering

Federico Ciccozzi[1]*, Ivano Malavolta[2]*, Christopher Timperley[3], Andreas Angerer[4] and Alwin Hoffmann[4]

[1]Mälardalen University, Västerås, Sweden, [2]Vrije Universiteit Amsterdam, Amsterdam, Netherlands, [3]Carnegie Mellon University, Pittsburgh, United States, [4]XITASO Holding GmbH, Augsburg, Germany

KEYWORDS

**robotic software, robotic software architecture, robotic software development, robotic software framework, software testing, software engineering**

Editorial on the Research Topic
Robotics software engineering

## Introduction

Robotics software engineering stands at the intersection of multiple disciplines, where physical interaction with dynamic and uncertain environments amplifies the complexity of traditional software challenges. As robots become indispensable in domains such as manufacturing, healthcare, transportation, and exploration, they must exhibit high levels of autonomy, adaptability, robustness, and safety. Achieving these qualities requires not only technical breakthroughs in algorithms and hardware but also a strong foundation in software engineering principles tailored to the unique demands of robotics.

Robotics inherently involves multidisciplinary integration: navigation, motion planning, manipulation, perception, control, and human-robot interaction must all coalesce within a coherent software framework. Engineering these systems requires the careful coordination of experts from each domain, whose contributions must reliably interoperate, often in real time. Further challenges arise from operating in environments that are partially observable, dynamic, and sometimes adversarial, which raises the stakes for ensuring correctness, security, and resilience.

This Research Topic, *Robotics Software Engineering*, brings together a diverse Research Topic of contributions aimed at addressing foundational and emerging challenges in this space. Rather than presenting a simple catalog of articles, this editorial aims to situate these works within the broader themes that are shaping the future of robotics software.

## Bringing rigor to robotics: model-based engineering and formal methods

As robotic applications become increasingly safety-critical, ensuring correctness through formal verification becomes not just desirable but necessary. However formal methods remain difficult to apply due to the manual effort required to create models and extract system parameters. Dust et al. at Mälardalen University (Sweden) addressed this issue head-on with a model-driven methodology for the automated formal verification of ROS 2 systems. By integrating model transformation pipelines with real execution traces, this work demonstrates how verification can become more modular, reusable, and accessible to non-experts. This toolchain lowers the barrier to rigorous analysis, allowing developers to iteratively assess critical system properties such as timing and scheduling without being formal methods specialists.

Similarly, Barnett et al. (University of York, UK) proposed RoboArch, an architectural modeling language layered atop the formal DSL RoboChart, which advances the discipline by providing verifiable architectural abstractions. When applied in industrial contexts such as nuclear robotics, RoboArch emphasizes the value of model-driven design in bridging the gap between informal software practices and formal correctness in real-world systems.

## Architectures for adaptivity and reusability

Adaptation is a recurring theme in robotic systems, where conditions often change unpredictably. Several contributions explored adaptive software architectures as key enablers of robustness and long-term autonomy. ROSA, a knowledge-driven framework for robot self-adaptation proposed by Silva et al. (TU Delft, Netherlands), exemplifies this direction. It captures application-specific knowledge in structured models and reasons over them at runtime to guide both task execution and architectural configuration—a co-adaptation capability rarely addressed in robotics.

Complementing this, the survey on ontology-enabled autonomy by Aguado et al. (Universidad Politécnica de Madrid, Spain) examined how semantic knowledge and reasoning improve robot behavior in open-ended environments. By analyzing trends in the use of ontologies for fault recovery, mission planning, and behavior selection, the article highlights how structured, declarative knowledge can foster more explainable and dependable autonomy.

The contribution by Schneider et al. (Hochschule Bonn-Rhein-Sieg, Germany and KU Leuven, Belgium), Semantic Composition of Robotic Solver Algorithms, introduced a composable, graph-based methodology for algorithm synthesis. By leveraging standards from the Semantic Web, the authors enabled the reuse and symbolic generation of solver code across application domains, from kinematics to probabilistic inference. These developments advance the field toward software that not only adapts itself but also explains its logic, a key step for collaborative and trustworthy robots.

## Improving software quality through early validation and testing

Traditional debugging and validation approaches are inadequate for robotics, where errors discovered at runtime can lead to costly damage or unsafe behavior. Therefore, early and automated validation is crucial.

With EzSkiROS, Rizwan et al. (Lund University) and colleagues addressed this issue by using embedded domain-specific languages (DSLs) which enable the early detection of errors in robotic skill composition. By embedding checks in the design and deployment phases, this approach detected both high-level contract violations and low-level implementation bugs before they manifested during execution. This shift left in the validation pipeline shortens the debugging loop and improves overall safety.

At the other end of the deployment pipeline, with AAT4IRS, Dos Santos et al. (Université du Québec à Chicoutimi, Canada) introduced a novel framework for automated acceptance testing in industrial robotic systems. Built on behavior-driven development principles, this approach uses natural language to specify test scenarios, enabling cross-functional collaboration between engineers and stakeholders. Mutation testing results showed strong fault detection capability, indicating the practical utility of the framework in high-stakes industrial environments.

Simulation-based testing also receives attention. Despite its potential, it remains underused due to the complexity of scenario definition. To address this issue, the article by Ortega et al. (University of Bremen and Ruhr University Bochum, Germany) presented a composable scenario framework for testing mobile robots in virtual environments. By enabling developers to incrementally build and reuse complex scenarios, the approach reduces overhead while improving test coverage and configuration error detection.

## Foundations and infrastructure: languages, patterns, and performance

The infrastructure underlying robotic software must be efficient, reliable, and extensible. Several contributions examine foundational aspects, including runtime patterns, data structures, and energy consumption.

The study by Artigas et al. (KU Leuven and Flanders Make, Belgium) introduced software coordination patterns such as acquire-release and cache-awareness, alongside data structures such as Petri nets and finite state machines, to support real-time task execution. The proposed runtime infrastructure separates event firing from handling, facilitating distributed deployment and enabling consistent coordination across multiple robots.

The contribution by Albonico et al. (Federal University of Technology of Paraná, Brazil) and colleagues addressed an increasingly important concern—energy efficiency—by comparing the resource usage of ROS 2 nodes written in C++ and Python. Empirical results confirmed that C++ outperforms Python in energy consumption, particularly in high-frequency communication tasks, offering valuable guidance to developers who are optimizing for battery-powered or resource-constrained platforms.

Containerization also emerges as a promising strategy for scalable integration Cotugno et al. (Ocado Technology, UK) and colleagues proposed a containerized approach for multiform robotic architectures, demonstrating how virtualization can simplify the integration of third-party components without compromising performance. Evaluated in a real-world industrial robot, this method showed that modern software engineering practices such as containerization can be successfully adapted to robotics, reducing setup complexity while maintaining real-time guarantees.

## Toward a mature discipline of robotic software engineering

Taken together, the articles in this Research Topic reflect a field that is rapidly maturing—seeking not only functional solutions to robotic problems but principled, reusable, and verifiable engineering practices. From architectural modeling to energy-aware programming, from scenario-based testing to self-adaptive reasoning, each contribution addresses a facet of the broader challenge: how to engineer robotic systems that are not only intelligent, but also trustworthy, maintainable, and ready for real-world deployment.

This Research Topic fosters synergy between academia and industry, theoretical rigor and practical deployment. It invites the community to further explore the foundational questions of variability, modularity, reusability, validation, and automation in robotic software development. As robots increasingly share our spaces and tasks, the importance of sound engineering for their software only grows.

We hope these contributions inspire continued innovation and cross-disciplinary collaboration in the journey toward robust and dependable robotic systems.

## Author contributions

FC: Writing – original draft. IM: Writing – review and editing. CT: Writing – review and editing. AA: Writing – review and editing. AH: Writing – review and editing.

## Funding

The author(s) declare that no financial support was received for the research and/or publication of this article.

## Conflict of interest

Authors AA and AH were employed by XITASO Holding GmbH.

The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## Generative AI statement

The author(s) declare that no Generative AI was used in the creation of this manuscript.

Any alternative text (alt text) provided alongside figures in this article has been generated by Frontiers with the support of artificial intelligence and reasonable efforts have been made to ensure accuracy, including review by the authors wherever possible. If you identify any issues, please contact us.

## Publisher's note

frontiers | Frontiers in Robotics and AI

TYPE Original Research
PUBLISHED 28 October 2022
DOI 10.3389/frobt.2022.991637

# Architectural modelling for robotics: RoboArch and the CorteX example

Will Barnett, Ana Cavalcanti* and Alvaro Miyazawa

Department of Computer Science, University of York, York, United Kingdom

Check for updates

OPEN ACCESS

EDITED BY
Rob Skilton,
United Kingdom Atomic Energy Authority, United Kingdom

REVIEWED BY
Domenico Redavid,
University of Bari Aldo Moro, Italy
Enrico Mingo Hoffman,
Pal Robotics S.L., Spain

*CORRESPONDENCE
Ana Cavalcanti,
Ana.Cavalcanti@york.ac.uk

SPECIALTY SECTION
This article was submitted to Field Robotics,
a section of the journal
Frontiers in Robotics and AI

RECEIVED 11 July 2022
ACCEPTED 23 September 2022
PUBLISHED 28 October 2022

CITATION
Barnett W, Cavalcanti A and Miyazawa A (2022), Architectural modelling for robotics: RoboArch and the CorteX example.
*Front. Robot. AI* 9:991637.
doi: 10.3389/frobt.2022.991637

The need for robotic systems to be verified grows as robots are increasingly used in complex applications with safety implications. Model-driven engineering and domain-specific languages (DSLs) have proven useful in the development of complex systems. RoboChart is a DSL for modelling robot software controllers using state machines and a simple component model. It is distinctive in that it has a formal semantics and support for automated verification. Our work enriches RoboChart with support for modelling architectures and architectural patterns used in the robotics domain. Support is in the shape of an additional DSL, RoboArch, whose primitive concepts encapsulate the notion of a layered architecture and architectural patterns for use in the design of the layers that are only informally described in the literature. A RoboArch model can be used to generate automatically a sketch of a RoboChart model, and the rules for automatic generation define a semantics for RoboArch. Additional patterns can be formalised by extending RoboArch. In this paper, we present RoboArch, and give a perspective of how it can be used in conjunction with CorteX, a software framework developed for the nuclear industry.

KEYWORDS

**software engineering, patterns, RoboStar framework, nuclear industry, verification, simulation, test, proof**

## 1 Introduction

Robotic systems are being used in an increasingly diverse range of applications, and in more dynamic and unstructured environments. With autonomy and the ability to operate in close proximity to humans, safety becomes an issue. Furthermore, robotic systems and their software are becoming more complex. In previous work, we have contributed to the verification of robotic systems using a domain-specific language with a formal semantics, namely, RoboChart (Miyazawa et al., 2017, 2019).

In this paper, we present an approach to defining RoboChart models for software that use architectures of wide interest in robotics. It is based on a novel domain-specific notation, RoboArch, presented here for the first time. It embeds robotics software architectural concepts and enables automatic generation, *via* model transformation, of partial RoboChart models, that is, sketches of RoboChart models that can be completed by designers with application-specific descriptions (of actions and state machines).

The definition of a system's architecture during its design has been considered a beneficial technique as the scale of software systems has grown. The architecture provides a structural representation that enables the evaluation of system attributes and of alternative system designs and modifications (Bass et al., 2012). From experience, practitioners have identified structures and relationships within system architectures that solve recurring problems. These solutions have been generalised as architectural patterns that are reusable in the design of new systems (Gamma et al., 1995). For the robotics domain, some common patterns have emerged: notably, the use of layers for robot control (Siciliano and Khatib, 2016, pp. 286–289).

In many other complex multidisciplinary domains, Model-Driven Engineering (MDE) is being used successfully to mitigate complexity (Franz et al., 2018). The core principle of MDE is to use abstract models of a system as the primary artefact(s) of its development process. This promotes identification of the underlying concepts free from specific implementation dependencies. The use of abstract models also facilitates the automation of the software development process. In this way developers can devote their time to understanding and solving the domain-specific problems.

Domain-specific languages (DSL) facilitate the development of models by embedding core concepts of a target domain, and enabling the definition of concise representations understood by practitioners. This avoids the need for each development team to identify these concepts, resulting in duplication of work and hindering reusability. Over the last 25 years, there have been considerable developments in MDE for robotics, with the creation of many DSL for its different sub-domains (Nordmann et al., 2016).

Some examples of DSL for robotics include: RobotML (Dhouib et al., 2012), SmartSoft (Stamper et al., 2016), and BCM (Bruyninckx et al., 2013). These DSLs, like the majority available, do not have formally defined semantics (Cavalcanti et al., 2021b). Therefore, the support for formal verification of robotic systems is limited. A recent literature survey (Luckcuck et al., 2019) found sixty-three examples of the application of formal methods within the robotics domain. Formal methods enable the early verification (proof, simulation, and testing) of a system through the use of rigorous automated techniques with mathematical foundations. Early use of verification techniques and high levels of automation enable the development of systems that are more reliable and cheaper.

RoboChart is a DSL for modelling robotics software controllers using state machines and a simple component model; RoboChart makes innovative use of formal methods for automated verification. The associated tool, RoboTool[1], provides features of MDE, which include a graphical interface for creating models, and automatic generation of source code and mathematical descriptions. Additionally, RoboChart supports automatic verification of properties such as deadlock and livelock freedom using model checking, along with semi-automatic verification techniques using theorem proving (Cavalcanti A. L. C. et al., 2021).

To date, RoboChart has been used to model more than twenty proof-of-concept case studies. They have facilitated the development and demonstration of RoboChart and its verification technology. None of them, however, adopt an elaborate software architecture. For larger robotic systems, support for modelling taking advantage of commonly used architectural patterns can enable explicit modelling of the structure of systems with potential to assist in reuse and compositional design and reasoning.

RoboArch allows the description of layered designs for robotic control software, and of design patterns for each layer. In this paper, we not only give an overview of RoboArch *via* a motivating example, but also present its complete metamodel and set of well-formedness conditions that specify the valid RoboArch models. We also describe our model-transformation approach, based on 50 rules, mechanised to generate automatically a sketch of a RoboChart model from a RoboArch architectural design of a system.

Besides supporting the description of architectural designs, RoboArch formalises a notion of a layered architecture and other patterns. Most of these patterns are described in the literature only informally, sometimes with different variations described by different authors. At best, patterns are realised in an implementation or programming language. Such descriptions necessarily mix the core concepts of the architectural patterns with those of the application or programming language. In contrast, the RoboArch formalisation identifies the core concepts of a pattern and their relationship.

The CorteX framework (Caliskanelli et al., 2021) has been designed for use in nuclear robotics to address the challenges of developing their complex robotic systems that need to be maintained over long periods of time, often to deal with changing requirements due to the unknown operational conditions. CorteX favours the development of maintainable and extensible systems through specialised data and communications designs. Designs for the CorteX middleware are inherently concurrent.

Our vision is the alliance of RoboArch and CorteX to support 1) the identification and formalisation of the architectural designs that rely on CorteX and 2) the elicitation of assurance evidence to increase confidence in CorteX-based software and support the construction of assurance arguments. By integrating CorteX with RoboArch, and, *via* RoboArch, to RoboChart, we connect CorteX to the RoboStar approach to Software Engineering for Robotics (Cavalcanti A. L. C. et al., 2021). With that, we enable, automatic generation of mathematical models that specify the meaning of the RoboArch designs,

1 robostar.cs.york.ac.uk/robotool/

and automatic and semi-automatic verification of properties *via* model-checking and theorem proving. Further specialisation of the approach can lead to automatic generation of CorteX code for simulation and deployment.

Our novel contributions in this paper are as follows.

1) Design of RoboArch for description of layered architectures for robotic control software.
2) Definition of the metamodel and well-formedness conditions of RoboArch.
3) Description of a technique for model-to-model transformation from RoboArch to RoboChart.
4) Formalisation of the reactive-skills architectural pattern for design of control layers, illustrating a general approach to formalise patterns using RoboArch and RoboChart.
5) Discussion of perspectives for allying the use of RoboArch and CorteX, and, in particular, of the formalisation of CorteX in RoboArch and RoboChart.

These results enable use of MDE in the development of control software for robotic systems in a way that focusses on use of well-known patterns allied with the advantages of modern verification techniques. RoboChart, and, therefore, RoboArch, are part of a design and verification framework, called RoboStar, that supports automated generation of simulations, tests, and proof.

In the next section, we describe related work on architectures for robotics. Section 3 presents RoboArch: metamodel, well-formedness conditions, and translation to RoboChart. Section 4 shows how a design pattern can be formalised in RoboArch using the example of reactive skills. In Section 5 we conclude, discussing our approach for the integration of RoboArch and CorteX as future work.

## 2 Related work

In this section, we discuss the literature on architectural patterns for robotics. Crucially, this justifies our choice of layer as a core concept in RoboArch, but also indicates other patterns of interest, including reactive skills, which we also formalise in this paper.

RoboArch is not related to the homonym in (Bonato and Marques, 2009), which is a tool to support the development of mobile robots. The focus in (Bonato and Marques, 2009) is on implementation, not modelling, of hardware-software co-designs based on hardware and software components, and code generation for FPGA, not software architectures. Moreover, there is no semantics or support for verification beyond simulation for the notation adopted by RoboArch to define the compositions.

Other works that share our aim to reduce effort in the development of control software in robotics focus on the programming, rather than the modelling, level. The result is a variety of middleware, encouraging code reuse and component-based development (Bruyninckx, 2001; Metta et al., 2006; Ando et al., 2008; Chitta et al., 2017; Muratore et al., 2017). These works provide useful resources for programming, but do not address the issues arising from a code, rather than model-based, approach to development. Work on RoboArch and RoboChart is complementary. In particular, we consider here how we can provide direct support for use of the modern CorteX middleware that has a track record in the nuclear industry.

Historical architectural patterns include Sense Plan Act (SPA) (Siciliano and Khatib, 2016, *p.* 285) and subsumption (Brooks, 1986). SPA is an example of a pattern that is deliberative: time is taken to plan what to do next, and then the plan is acted out with no sensing or feedback during acting. A robot using SPA in a dynamically changing world can be slow and error prone in response to environmental change.

Conversely, subsumption is an example of an architectural pattern that is reactive, where the environment is constantly sensed and used to directly shape the robot's actions. A robot using subsumption responds rapidly to a changing world; however, complex actions are difficult to achieve.

More recent hybrid architectural patterns combine the principles from SPA and subsumption to benefit from both the deliberative and reactive properties. In total, twenty-two architectural patterns used by robotics systems have been identified from the literature; these are listed in Table 1. Five have been selected for discussion based upon evidence of application, reuse, and activity of development. The collective publications that focus on an architectural pattern have been used to find evidence of application, with the scale of any documented application used to give preference to patterns that have been used in large deployments in the real world. The number of publications where an architectural pattern was used in a new application has been used to asses reuse. Finally, preference has been given to patterns with recent activity, determined by the date and frequency of publications where the pattern has been used.

**LAAS** was developed at LAAS[2] in 1998 for autonomous robots. A fundamental goal of LAAS is to provide both deliberative and reactive capabilities required for autonomy.

The LAAS pattern is made up of the following three layers. The *Functional Layer* provides basic robot actions that are organised into modules consisting of processing functions, task loops, and monitoring functions for reactive behaviour. An *Execution Control Layer* selects functions from the functional layer to carry out sequences of actions determined by the decision layer. Finally, the *Decision Layer* plans the

2 Laboratory for Analysis and Architecture of Systems CNRS.

TABLE 1 The patterns identified from the literature.

| Pattern | Focus | Year |
|---|---|---|
| CoSiMA Wigand et al. (2018) | Safe real-time robots | 2018 |
| [a]IRSA Backes et al. (2018) | Autonomous robots | 2018 |
| [a]SERA García et al. (2018) | Decentralised teams | 2018 |
| Aerostack Sanchez-Lopez et al. (2017) | Autonomous unmanned aerial systems | 2017 |
| [a]CARACaS Huntsberger and Woodward, (2011) | Autonomous robots | 2011 |
| EFTCoR Álvarez et al. (2006) | Service robot control | 2006 |
| Syndicate Sellner et al. (2006) | Autonomous teams | 2006 |
| DDX (Corke et al., 2004) | Distributed robot controllers | 2004 |
| [a]CLARAty (Volpe et al., 2001) | Autonomous robots | 2001 |
| HARPIC (Luzeaux and Dalgalarrondo, 2001) | Autonomous robots | 2001 |
| [a]LAAS (Alami et al., 1998) | Autonomous robots | 1998 |
| Remote Agent (Muscettola et al., 1998) | Autonomous robots | 1998 |
| ORCCAD (Borrelly et al., 1998) | Robot control | 1998 |
| Planner Reactor (Lyons and Hendriks, 1995) | Autonomous robots | 1995 |
| Reactive Skills (Yu et al., 1994) | Autonomous robots | 1994 |
| CIRCA (Musliner et al., 1993) | Real-time intelligent robots | 1993 |
| ATLANTIS (Gat, 1992) | Autonomous robots | 1992 |
| Layered Competencies (Bonasso, 1991) | Autonomous robots | 1991 |
| Motor Schema (Arkin, 1989) | Robot control | 1989 |
| NASREM (Albus et al., 1989) | Autonomous robots | 1989 |
| AuRA (Arkin, 1987) | Autonomous robots | 1987 |
| Subsumption (Brooks, 1986) | Autonomous robots | 1986 |

[a]Selected for further discussion.

sequence of actions necessary to achieve mission goals and supervises the execution of the plans.

The functional layer consists of a network of modules that provide services related to a particular sensor, actuator, or data resource of the robot. All modules have a fixed generic structure made up of a controller and execution engine. A tool can be used to generate module source code. The services provided by the modules are accessed by the executive layer above and other modules from the functional layer through the use of a non-blocking client-server communication model.

The execution control layer bridges the slow, high-level, processing of the decision layer, and the fast, low-level, control of the functional layer. It has an executive module that takes sequences of actions from the decision layer, and selects and triggers the functions that the functional layer must carry out. In addition, the executive receives replies from the functional layer and reports activity progress to the decision layer.

The decision layer has one or more pairs of a supervisor and a planner. The supervisor takes a sequence of actions from the planner and manages their execution by communicating them to the execution layer, and responding to reports received from it. The planner creates a sequence of actions to achieve a goal. The supervisor also passes down situations to monitor and associated responses within the constraints of the plan. These responses enable the lower layers to react without the need for involvement of the decision layer, improving response time and reducing unnecessary replanning.

LAAS has been used in the implementation of the ADAM rough terrain planetary exploration rover (Chatila et al., 1995), and of three Hilare autonomous environment exploration robots as part of the MARTHA European project. More recently, Behaviour Interaction Priority (BIP) models have been used to verify the functional layer of the LAAS pattern (Silva et al., 2015).

**CLARAty** (Coupled Layer Architecture for Robotic Autonomy) was developed at NASA in 2001 for planetary surface-exploration rovers. CLARAty is designed to be reusable and to support multiple robot platforms; it consists of two-layers: a functional layer, and a decision layer formed by combining the planning and executive layers from a three-layer architecture. A key concept defined in CLARAty is granularity, which reflects the varying levels of deliberativeness available to the robotic system.

The functional layer provides a software interface to the hardware capabilities of the robot, and it is structured using an object-oriented hierarchy. At the top of the hierarchy is the Robot superclass from which everything inherits. At subsequent levels down the hierarchy, classes are less abstract and each provide functionality for a piece of the robot's hardware. At the bottom of the hierarchy, each class provides access to a specific piece of hardware functionality and its current state.

Classes can provide functionality that requires minimal input from the decision layer, therefore, this type of class can be considered more reactive. For example, the class for a rover may offer a method for obstacle avoidance. Alternatively classes can provide functionality that requires regular input from the decision layer, therefore, the class can be considered more deliberative. For example, the class for a robotic arm may offer a method for setting the position for one of its five motors.

The single decision layer enables state information between planner and executive to be shared, which means that the planner becomes tightly integrated with the executive. Consequently, discrepancy between the planner and the functional layer's state is minimised.

The CLARAty pattern has been used for a variety of robot platforms: Rocky 8, FIDO, ROCKY 7, K9 Rovers, and ATRV Jr COTS platform (Nesnas et al., 2006). The different platforms have a variety of deployment architectures, from a single processor requiring hard real-time scheduling, to distributed microprocessors using soft real-time scheduling.

**CARACaS** (Control Architecture for Robotic Agent Command and Sensing) is an architectural pattern developed at NASA in 2011 for control of autonomous underwater vehicles (AUV), and autonomous surface vehicles (ASV). CARACaS-based software supports operation in uncontrolled environments ensuring the vehicles obey maritime regulations. A CARACaS design supports cooperation between different vehicles and makes use of dynamic planning to adapt to the current environmental conditions and mission goals.

The five main elements of CARACaS are as follows. *Actuators* interface the actuators of the vehicle. A *Behaviour Engine* coordinates and enables the composition of behaviours acting on the actuators. The arbitration mechanisms controlling the enabling and disabling of behaviours are subsumption, voting, and interval programming. A *Perception Engine* creates maps for safe navigation and hazard perception from the sensors. A *Dynamic Planning Engine* chooses activities to accomplish mission goals while observing resource constraints. For that, it uses Continuous Activity Scheduling Planning Execution and Replanning (CASPER) (Chien et al., 2000), and issues commands to the Behaviour Engine. Finally, a *World Model* contains state information including plans, maps, and other agents.

Layers are not defined in Huntsberger and Woodward (2011), but a CARACaS design can be partitioned into two layers. At the lowest level, a behavioural layer includes the Actuators, and the Behaviour and Perception Engine elements. The higher layer consists of the Dynamic Planning and the World Model.

Although CARACaS is targeted at autonomous water-based vehicles, it contains all of the required elements to be applied more generally as a pattern for the control of robots.

**IRSA** (Intelligent Robotics System Architecture) was developed at NASA in 2018 to streamline the transition of robotic algorithms from development onto flight systems by improving compatibility with existing flight software architectures. IRSA uses concepts from other patterns: CARACaS and CLARAty.

The main elements of IRSA are as follows. A *Primitive* provides low-level behaviours that can have control loops. *Behaviour* provides autonomy, transitioning between multiple states during execution. The *Executive* receives and executes a sequence of instruction commands from the planner. The *Planner* uses the system state from the world model to produce the sequence of command instructions. A *Sequence* contains the instructions that the robot must perform. A *Verifier* verifies whether the sequence is valid. Finally, the *Robot World Model* maintains a model of the robot with local and global state information.

An IRSA design can be mapped onto a three-layer pattern with a common world model accessible to all layers. The behavior and the primitive elements provide control over the robot; so, these two elements can be placed in the bottom layer. The executive receives sequences of commands and manages their execution using the behaviours. Therefore, the executive is the middle layer. The planner uses the state of the system from the world model to create a sequence of commands checked by the verifier. Therefore, the planner, sequence, and verifier elements are in the layer above the executive.

The IRSA architectural pattern has been deployed on a variety of test beds: comet surface sample return, Europa lander, Mars 2020 Controls and Autonomy, and the RoboSimian DARPA challenge.

**SERA** (The Self-adaptive dEcentralised Robotic Architecture) has been developed at the Chalmers University of Technology in 2018. SERA's primary goal is to support decentralised self-adaptive collaboration between robots or humans, and it is based on the three-layer self-management architectural pattern. SERA has been evaluated in collaboration with industrial partners in the Co4Robots H2020 EU project.

The layers of the SERA pattern are as follows. The *Component Control Layer* provides software interfaces to the robot's sensors and actuators, grouped into control action components responsible for particular areas of functionality. The *Change Management Layer* receives the local mission and creates a plan in order satisfy its goals. It executes the plan by calling appropriate control actions from the component control layer. Finally, the *Mission Management Layer* manages the local mission for each robot and communicates with other robots in order to synchronise and achieve the global mission.

The mission management layer receives a mission specification from a central station as a temporal logic formulae. The mission manager checks its feasibility and, if it is feasible, passes the mission to the adaptation manager in the layer below. If the mission is infeasible, a communication and collaboration manager communicates and synchronises with the other robots involved in the mission. During the synchronisation, an updated achievable mission that meets the original mission specification is computed.

This pattern places more functionality in the lowest component control layer. A key feature of SERA is communication among robots, which provides greater flexibility in achieving the mission goals.

## 2.1 Discussion

Generally no particular pattern or selection of patterns are widely used. There is a tendency for each project to establish its own pattern. Between research groups, however, there is some reuse of patterns.

Layers are a common theme among many of the recent architectural patterns. Even when layers have not been explicitly specified, the elements of a pattern are structured such that they can be mapped onto a layered architectural pattern. All patterns have a functional layer that interacts with the robots sensors and actuators. The upper layers following the functional layer vary in number and purpose.

The functional layer is required by all architectural patterns because every robot requires a means to sense and interact with its environment. From the patterns surveyed, this layer can be categorised as either service or behavioural. CLARAty, LAAS and SERA are all examples of patterns that have a service-based functional layer, whereas, CARACaS and IRSA have behavioural-based functional layers.

Examples of behavioural control patterns that can be used for functional layer include subsumption (Brooks, 1986) and reactive skills as used by the control layer of 3T (Bonasso et al., 1997). It is common for the decision layer to be placed directly above the functional layer.

Patterns that do not use an executive layer take different approaches to managing the system's state. For instance, SERA and CLARAty use information in the decision layer to hold system state. Whereas, CARACaS uses a world model layer that is accessible by all other layers to hold system state.

Some patterns such as SERA have an additional social layer for collaboration between teams of robots. Similarly LAAS supports this through adding supervisor-planner pairs, but considers this to be an extension of the decision layer rather than a new layer. Generally the layered pattern lends itself to the addition of new layers for extending the level of system capability.

RoboArch directly supports the definition of layered architectures, with an arbitrary number of layers. A degenerate layered architecture with just one layer can be used to define a design that does not actually uses layers. As indicated above, however, the use of more elaborate layers, some using specific patterns themselves, is common. In what follows, we present the RoboArch notation.

## 3 Materials and methods: RoboArch

In this section, we show how a layered design can be described using RoboArch. We give an overview using the example of an office delivery robot from (Siciliano and Khatib, 2016, pp. 291–295) (Section 3.1). In Section 3.2 we present the complete metamodel and well-formedness conditions of RoboArch. Finally, in Section 3.3, we describe the RoboChart model defined by a RoboArch design. In the next Section 4, we show an example of how a pattern for the control layer can be characterised and used.

## 3.1 Overview

RoboArch is a self-contained notation that can be used independently. As mentioned, however, its semantics is given by translation rules that define a (sketch of a) RoboChart model. This not only gives RoboArch a precise and formal semantics, but also paves the way for the use of the RoboStar framework to design and verify the control software. Figure 1 gives an overview of the possibilities.

As indicated in Figure 1, a key concept in RoboArch is that of a robotic platform. RoboArch designs are platform independent, so the robotic platform here describes the services the robot provides that can be used in the development of the control software. The services are abstractions of the robot's sensors and actuators defined *via* the declaration of input and output events and operations that can be realised *via* actual sensors and actuators. The same approach is taken in RoboChart.

To give an overview of the RoboArch notation, we consider the example of a robot whose goal is to deliver items of post within a typical office building, transporting them from a central mailroom to each of the offices within the building. To achieve its goal the robot must safely navigate along the corridors of the building while avoiding any obstacles such as people and furniture.

```
system MailDelivery

type Coordinate
datatype Office {
    number:nat
    location:Coordinate
}
...
interface Base {
    Move (  request : Velocities )
    const ZERO_LINEAR_VELOCITY : Velocity
    const ZERO_ANGULAR_VELOCITY : Velocity
}
...
robotic platform DeliveryRobot {
    provides Base
    provides Audio
    uses PointCloud
    uses EnvColourPoints
}

layer Pln: PlanningLayer {
    inputs deliveryComplete, pickupFailed;
    outputs deliverMail: Office ;
} ;

layer Ctl: ControlLayer {
    requires Base
    uses PointCloud
    ...
    inputs destination:Coordinate, doorToFind:nat,
            getLocation, speak: string;
    outputs location:Coordinate, destinationReached,
             doorFound, mailPickedUp;
} ;

connections
    Pln on deliverMail to Exe on deliverMail,
    DeliveryRobot on envPoints to Ctl on envPoints,
    ...
```

**Listing 1.** A `system` and its type declarations.

A RoboArch model for the mail delivery system is sketched in Listing 1. A `system` clause gives a name to a model and introduces the outer scope to define the layers and the robotic platform. The robotic platform must be used by a single layer,

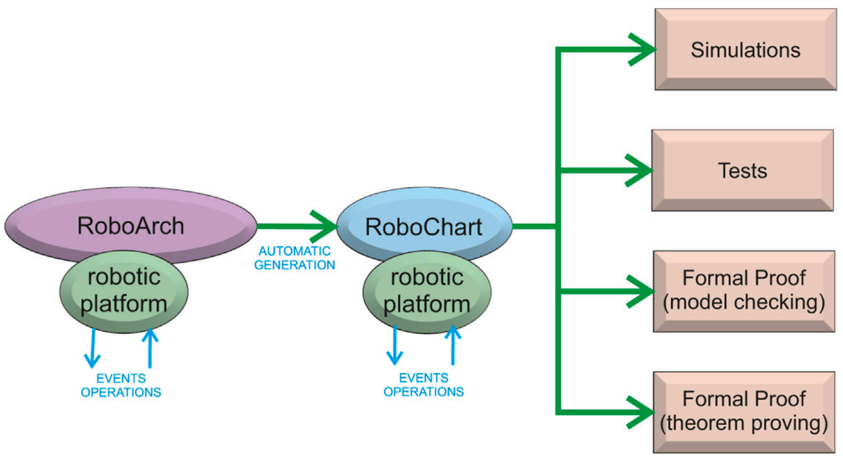


FIGURE 1
RoboArch in the context of RoboStar. With a RoboArch architectural design, we can generate automatically a sketch of a RoboChart behavioural model. Using the RoboChart model, we can take advantage of a plethora of modern verification techniques supported by automated generation of artefacts.

usually the control layer. In addition to the architectural elements, a RoboArch model also contains definitions for types, functions, interfaces, and connections. For our example, Line 1 of Listing 1 declares the `system` with the name `MailDelivery`.

RoboArch adopts the type constructors and typing rules of the well-established data modelling notation Z (Woodcock and Davies, 1996), allowing the definition of primitive types, records, sets, and so on. RoboChart and all RoboStar notations adopt the same typing approach. By adopting the Z type system, we benefit from a well-known powerful type system, which has the expected facilities to define a rich, possibly abstract, data model, and that is supported by verification tools. In our example, the next few lines define types. Most type definitions are omitted here, but the complete example is available[3].

Robotic platforms are normally defined in terms of interfaces. For our example, the robotic platform is named `DeliveryRobot`, and its definition references interfaces `Base`, `Audio`, `PointCloud`, and `EnvColourPoints`, some omitted in Listing 1. Interfaces group events or operations, and are referenced using `provides` and `uses` clauses in a platform definition. The `Base` interface models the interactions that control movement. There is one operation `Move` and two constants. `Move` is an abstraction for motor functionality that can be accessed by the software *via* a call to this operation. It is a service provided by the platform, since `Base` is declared in a `provides` clause. The interfaces declared in `uses` clauses contain events that represent points of interaction (inputs and outputs), corresponding to inputs from sensors, or outputs to actuators. They are used by connecting the platform events to those of a layer.

The design in Listing 1 is a typical three-layer architecture. Every layer has a unique name, and optionally can have a type, a pattern, inputs and outputs. The three specific layer types are `ControlLayer`, `ExecutiveLayer`, and `PlanningLayer`. We can also not provide a type so that a customised architectural structure can be defined. The services of a layer are accessed through its inputs and outputs.

The `layer` clause is used to define the layer name and type. In Listing 1, we show a layer with name `Pln` and type `PlanningLayer`. It has one output `deliverMail` of type `Office` that requests the number of the office to which mail is currently being delivered. There are two inputs `deliveryComplete` and `pickupFailed` that have no associated value type; their occurrence indicates an outcome of the currently requested delivery. The `inputs` and `outputs` are used to communicate with another layer or the robotic platform; in our example communication is with an executive layer, omitted in Listing 1.

A layer of control type can directly communicate with a robotic platform, and so reference platform interfaces. The control layer for our example is `Ctl`. Its inputs and outputs communicate with the executive layer and `DeliveryRobot`. The `requires` and `uses` clauses reference the interfaces with the operations of the platform that it requires and, the events that it uses. While an `ExecutiveLayer` and a `PlanningLayer` cannot require or use services of a platform, a generic layer also can.

The connections among the layers and the robotic platform are defined under a system's `connections` clause. Each connection is unidirectional and connects an input or output

3 https://robostar.cs.york.ac.uk/case_studies/

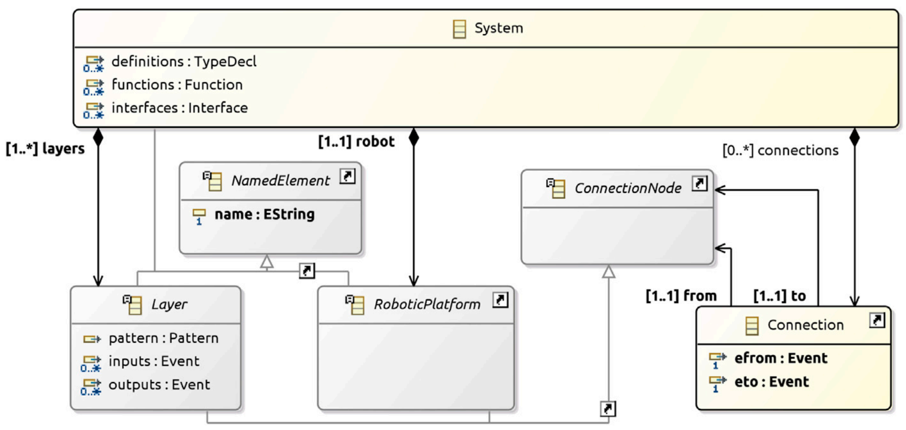


FIGURE 2
System metamodel.

on a layer or event of the platform to another. Listing 1 shows some of the connections for our mail delivery example. For example, the first declares a connection from the Pln layer's deliverMail output event to an Exe layer's deliverMail input event. The second connection is between the robotic platform (DeliveryRobot) and the control layer (Ctl). Several other connections are omitted.

In the next section, we give a complete description of the structure of RoboArch designs.

## 3.2 Metamodel and well-formedness

Figure 2 presents the RoboArch metamodel: the classes, and their attributes and associations, that represent a RoboArch design. The main class is System, whose objects have definitions of layers, robotic platform, connections, definitions, functions, and interfaces. The classes TypeDecl, Function, and Interface defining types for attributes of System come from the RoboChart metamodel (Miyazawa et al., 2020).

The RoboticPlatform class also comes from the RoboChart metamodel. RoboticPlatforms have a name and can declare events and variables as well as reference interfaces.

Layers can optionally have a pattern that defines their behaviour. (An example is presented in the next section.) Layers can also have inputs and outputs, which are Events, a concept also from RoboChart. An Event can have a type, which, if present, defines the values that can be communicated.

Systems, RoboticPlatforms and Layers are NamedElements: they have a name attribute. RoboticPlatforms and Layers are also ConnectionNodes: elements that can be connected *via* their events. Connections are between a source efrom and a target eto event that belong to the to and from ConnectionNodes.

Layer is further defined in Figure 3; it has four subclasses. A GenericLayer represents the most general kind of layer, without a declared type, offering flexibility to model systems with minimal restrictions. The three other kinds of layers, ControlLayer, ExecutiveLayer, and PlanningLayer, have specific well-formedness conditions (discussed later) that characterise the connections and patterns of a valid architectural design.

GenericLayers and ControlLayers can communicate with the RoboticPlatform. They are, therefore, represented by subclasses of an abstract class PlatformCommunicator. The objects of this class have required and defined attributes that record the interface declarations.

As mentioned, Layers can have a pattern. Figure 3 includes examples of patterns represented by subclasses of Pattern, namely, ReactiveSkills, Subsumption, Htn, and PlannerScheduler. To formalise a pattern for use in RoboArch designs, we need to add a subclass of Pattern to represent it. In the next section, we detail ReactiveSkills as an example of how a pattern can be formalised. Section 5 describes a RoboArch pattern for CorteX designs. Current work is considering the formalisation of Subsumption and Htn.

Not all models that can be created obeying the metamodel are valid. For instance, considering just the restrictions defined by the metamodel, we can create an architecture that connects events of different types. No typing rules are captured in the metamodel. As another example, the metamodel allows the specification of an architecture without connections with the robotic platform. Such design is for a software that does not carry out any visible task, and we regard it as invalid. Although we could translate such designs to RoboChart, there is little point in

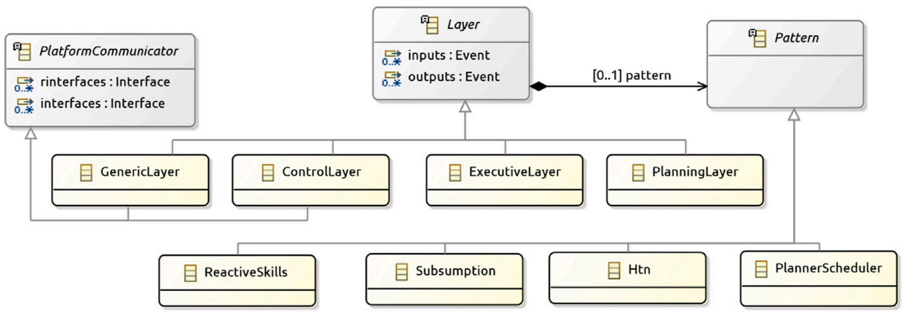


FIGURE 3
Layers metamodel.

TABLE 2 The well-formedness conditions of RoboArch.

| Condition | Description |
|---|---|
| S1 | A System must have one or more connections that relate a single Layer to a RoboticPlatform or there must be a Layer that has at least one or more required interfaces (elements in rinterfaces) |
| S2 | For Systems with more than one Layer, each Layer must have at least one input or output |
| S3 | For Systems with more than two Layers, their ordering given by Connections must be: ControlLayers < ExecutiveLayers < PlanningLayers |
| S4 | Connections must associate a Layer with at most two other Layers |
| S5 | Connections involving the ControlLayer must associate it with at most one other Layer |
| S6 | The connections of a System must associate events defined by interfaces of GenericLayers and ControlLayers with events of the RoboticPlatform |
| S7 | Connections efrom and eto event types must match |
| S8 | Connections must connect Layer inputs to outputs or *vice versa* |

delaying the identification of problems by working with invalid designs.

Instead, we define well-formedness conditions, presented here in Table 2, which characterise the valid designs. These conditions provide modellers with additional guidance and support for validation when defining an architecture.The conditions can be checked by the RoboArch tool.

S1 ensures that it is possible to interact with the platform. Because a Layer must provide a service, S2 ensures that it provides a means for external interaction. S3 ensures that, if used, the ExecutiveLayer is the middle intermediate layer between the planning and control layers with no direct communication between them. S4, S5, and S6 are concerned with the proper use of layers, without bypassing communications, and creating inappropriate dependencies. S7 prevents type errors, and S8 ensures correct data flow.

In the next section we describe how a RoboArch design can be formalised in RoboChart, and how transformation rules can be used to generate RoboChart models.

## 3.3 RoboArch in RoboChart

Table 3 presents an informal account of how RoboArch elements can be mapped to a RoboChart model. Transformation rules formalise this mapping, defining the (formal) semantics of RoboArch; their implementation allows the automatic generation of RoboChart models. RoboChart's formal semantics underpins RoboArch and allows properties of a RoboArch design to be verified (see Figure 1). Figure 4 presents parts of the RoboChart model for the design in Listing 1.

The top-level transformation rule, shown in Figure 5, maps a RoboArch System to the RoboChart type definitions, functions, interfaces, and robotic platform that it declares. Importantly, the top rule defines a valid RoboChart module for the `system`. The mapping provides a graphical representation as well as a semantics for these elements, since RoboChart is a diagrammatic language. Due to space restrictions, we cannot present all the transformation rules, but they are available[4].

A module is the RoboChart element representing a (parallel) robotic control software. In Figure 4, the module OfficeDelivery defines the RoboArch `system` of the same name.

A RoboChart module has its platform-independent behaviour characterised by a RoboticPlatform and one or more parallel Controllers whose behaviours are defined by one or more state machines running in parallel. The module

4 https://robostar.cs.york.ac.uk/publications/reports/roboarch_rules.pdf

TABLE 3 Mapping RoboArch to RoboChart.

| RoboArch | RoboChart |
|---|---|
| System | Module, TypeDecl, Function, Interface, RoboticPlatform |
| Layer | Controller |
| inputs and outputs | Events |
| Connection | Connection |

defined by a RoboArch `system` includes a reference to its RoboticPlatform, and one `Controller` for each Layer. In Figure 4, the module has *references* to the platform DeliveryRobot, and to three controllers Pln, Exe, and Ctl named after the layers of the RoboArch system.

The inputs and outputs of a Layer become events of the corresponding RoboChart controller. Connections between layers and the robot become connections between controllers and the robotic platform.

Figure 4 shows the RoboChart controller for the planning layer Pln. The events appear along the border of the controller. Inside, there is a reference to a single minimal machine stm0 because in this example no pattern is specified by the RoboArch design. The minimal machine, also shown in Figure 4, is a placeholder to be changed by the designer to specify their required behaviour. The minimal machine consists of a single initial junction, a state s0, and a transition that leads from the initial junction to s0.

For illustration, we show the top rule SystemToRCModule in Figure 5; it uses further rules (omitted here) to specify the RoboChart resulting elements rcdefs, rcfuns, rcifs, and rcmod that give the semantics of the system amsys given as input. The other rules are specified in the same style.

The resulting RoboChart type definitions rcdefs are the union of RoboArch `system` type definitions amsys. definitions and the generalised union ($\bigcup$) of the definitions resulting from applying a rule LayerToTypes to each RoboArch layer (amsys.layers). The types used in the rule definitions (TypeDecl, Interface, RCModule, and all others) are part of the RoboArch and RoboChart metamodels. They define the valid attributes (amsys.definitions, amssys layers, and so on). The definitions of the results rcfuns and rcifcs are similar to that of rcdefs, but use the rules LayerToFunctions and LayersToInterfaces.

The resulting RoboChart module rcmod is given by an object (specified by the construct $\lhd \_ \rhd_{RCModule}$) whose attributes define the name, nodes (controllers and robotic platform), and connections. The name of the module is the system name amsys. name. The nodes are the controllers defined by applying the rule LayersToControllers to the system's layers and a roboticPlatform as defined in the where clause. The connections of the module are those defined directly by amsys. connections.

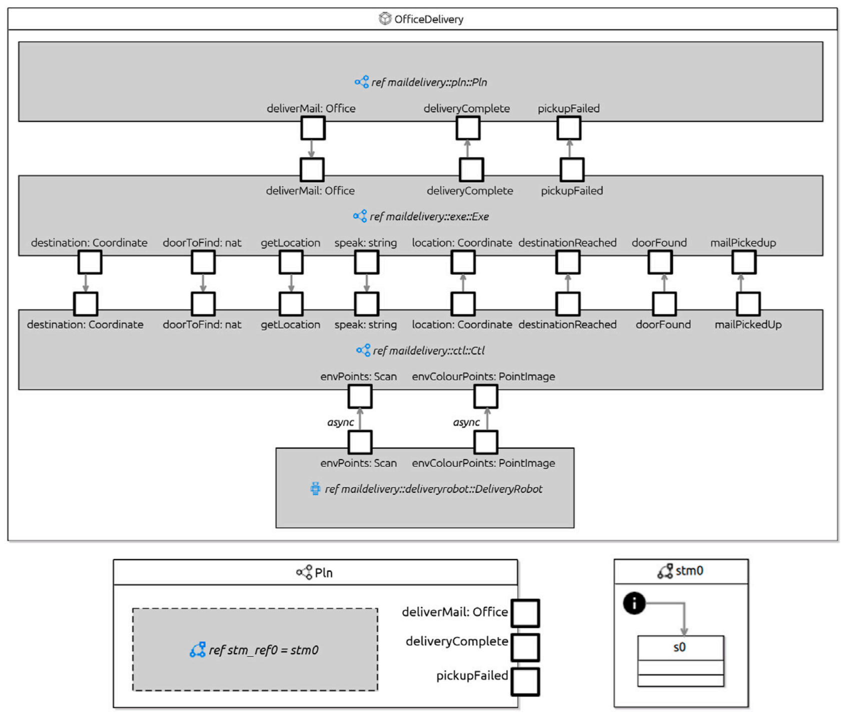
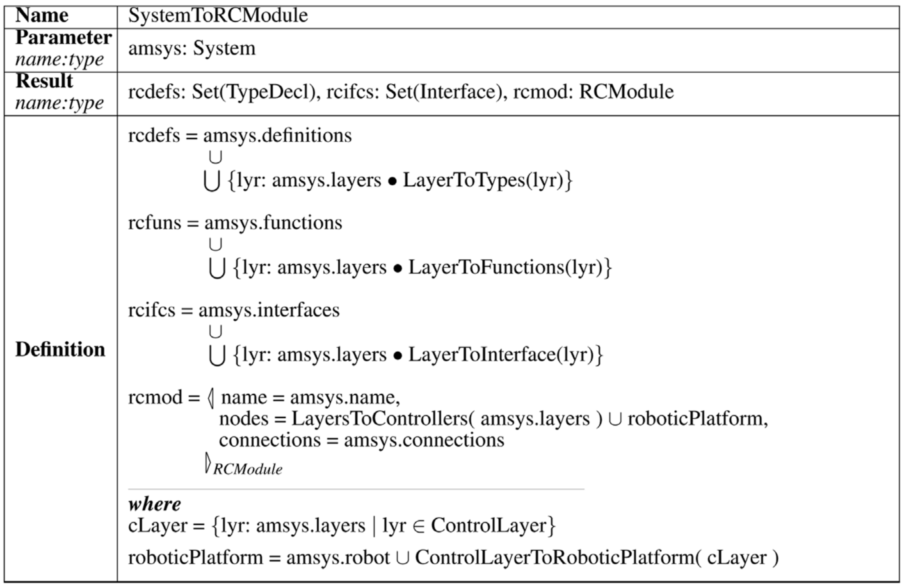


FIGURE 4
Delivery robot in RoboChart.



| **Name** | SystemToRCModule |
|---|---|
| **Parameter** *name:type* | amsys: System |
| **Result** *name:type* | rcdefs: Set(TypeDecl), rcifcs: Set(Interface), rcmod: RCModule |
| **Definition** | $\text{rcdefs} = \text{amsys.definitions} \cup \bigcup \{\text{lyr: amsys.layers} \bullet \text{LayerToTypes(lyr)}\}$ $\text{rcfuns} = \text{amsys.functions} \cup \bigcup \{\text{lyr: amsys.layers} \bullet \text{LayerToFunctions(lyr)}\}$ $\text{rcifcs} = \text{amsys.interfaces} \cup \bigcup \{\text{lyr: amsys.layers} \bullet \text{LayerToInterface(lyr)}\}$ $\text{rcmod} = \langle\!\vert\ \text{name} = \text{amsys.name}, \text{nodes} = \text{LayersToControllers( amsys.layers )} \cup \text{roboticPlatform}, \text{connections} = \text{amsys.connections}\ \vert\!\rangle_{\mathit{RCModule}}$ ***where*** $\text{cLayer} = \{\text{lyr: amsys.layers} \mid \text{lyr} \in \text{ControlLayer}\}$ $\text{roboticPlatform} = \text{amsys.robot} \cup \text{ControlLayerToRoboticPlatform( cLayer )}$ |

FIGURE 5
Example transformation rule.

The where clause defines the roboticPlatform to be the union of the RoboArch platform amsys. robot with the result of applying a rule ControlLayerToRoboticPlatform to the control layer cLayer. The platform amsys. robot is directly mapped to the RoboChart model. The layer cLayer is defined (*via* a set comprehension) as the layer lyr of amssys layers whose type is ControlLayer. The well-formedness conditions ensure that there is at most one such layer. With the use of ControlLayerToRoboticPlatform, we cater for the possibility that a pattern in the control layer extends the definition of the platform.

Although the translation of a layered design from RoboArch to RoboChart is reasonably direct, use of RoboArch, instead of constructing a RoboChart model from the start, has several advantages. RoboArch provides clear guidance on how to define and connect a robotic platform and the controllers; validation ensures definition of proper layers. On the other hand, translation to RoboChart provides support for verification. For example, we can prove that the RoboArch design is deadlock free.

In the next section, we show how we can enrich the definition of a layer.

## 4 Results: Reactive skills in RoboArch

With the RoboArch framework defined in the previous section, we can now formalise and use specific architectural patterns. In this section, we explain how to achieve that using the reactive-skill pattern for illustration. We first provide an overview of the pattern (Section 4.1), and then formalise it *via* a metamodel and well-formedness conditions (Section 4.2), and *via* transformation to RoboChart (Section 4.3).

### 4.1 Overview

The reactive-skills pattern can be used in the control layer, typically of a three-layer architecture (Bonasso et al., 1997). It combines deliberation and reactivity to improve robustness. The pattern has been used in a variety of applications: a robot to identify people and approach them (Wong et al., 1995), a trash collecting robot (Firby et al., 1995b), a robot that navigates a building (Firby et al., 1995a), and in the automation of remote-manipulation system procedures for the space shuttle (Bonasso et al., 1998). A framework that allows skills to be implemented using C, C++, Pascal, LISP and REX is available (Yu et al., 1994).

We characterise the reactive skills pattern by two concepts: skills and a skills manager. A *skill* performs an operation using input values, which can be from sensors or outputs of other skills. The skill's output values can establish associations to and from the robotic platform, or be the result of applying a computational transform to the skill's inputs. A set of skills is used together to accomplish a task identified in the dependant (typically executive) layer. A skills manager is a cyclic mechanism that coordinates communication between skills and provides an interface for the dependant layer to: run the skills required for

TABLE 4 The elements of reactive skills.

| Element | Description |
|---|---|
| Initialisation routine | When the system starts, the skill initialises itself |
| Startup | A skill performs required startup procedures each time it is activated |
| Reply | Response from the control layer to the dependant (executive) layer |
| Cleanup | When a skill is disabled, cleanup actions are performed |
| Parameter | A variable that allows a skill's behaviour to be adjusted by the dependant layer |
| Input | Receives the value of a data type |
| Output | A resulting value that contributes to the robot's behaviour |
| Computational transform | Once activated the skill continually computes its outputs from its inputs |
| Activate function | Allows a skill to be activated |
| Deactivate function | Allows a skill to be deactivated when it is no longer required |

a task, receive notifications from monitored events, and set and get parameter values of skills.

Skills can be of one of two types: D-Skill or C-Skill (Yu et al., 1994). D-Skills interface physical devices such as sensors and actuators with the other skills of the control layer; their input values are actuation commands and their output values are sensor data. C-Skills execute a computational transform using the skill's inputs to determine its outputs. By the monitoring of skills, the manager triggers events on desired conditions becoming true. Table 4 describes elements used by skills and skills managers.

In Listing 2, we sketch the design of the control layer of `MailDelivery` using reactive skills to specify the behaviours regarding moving the delivery robot to a given target location. The type of pattern specified by the `pattern` clause determines the subsequent clauses that can be used. For reactive skills, the subsequent clauses are `skills`, `connections`, and `monitors`.

Every `skill` has a unique name, and optionally parameters, a priority, inputs, and outputs. The `skills` clause declares the skills. There are separate clauses for defining each type of skill: `dskill` for D-Skills and `cskill` for C-Skills. In Listing 2, we show three D-Skills named `Move`, `ColourVision`, and `Proximity`, and one C-Skill `DetermineLocation`.

RoboArch `dskills` and `cskills` declare inputs and outputs using the `inputs` and `outputs` clauses. A skill communicating a value to a `dskill's input` results in the physical state of the device that the `dskill` represents being potentially affected. In our example, a value communicated to the `Move` D-Skill `velocity` input results in the velocities of the motors in the robot's base being set.

A value from a `dskill's output` represents the state of the environment, as sensed by the device the `dskill` represents. In our example, a value received from the `Proximity` skill's `envPoints` output determines a range of distances to surfaces in the delivery robot's field of view.

A C-Skill uses its inputs to compute its outputs resulting in behavior that can be used to accomplish parts of a task. In our example, the `DetermineLocation` skill takes a colour image of the environment and using an image-based localisation technique calculates the coordinates of the delivery robot. To perform this function, `DetermineLocation` has one `input image` of type `PointImage`, and one `output location` of type `Coordinate`. The computational transform that specifies the behaviour of C-Skills can be defined by customising the generated RoboChart model.

```
pattern ReactiveSkills;

skills
        dskill Move {
            inputs velocity:Velocities;
        },
        dskill ColourVision {
            outputs envColourPoints:PointImage;
        },
        dskill Proximity {
            outputs envPoints:Scan;
        },
        cskill DetermineLocation {
            inputs image:PointImage;
            outputs location:Coordinate;
        }
    ...

connections
    ColourVision on envColourPoints to DetermineLocation on image,
    ...

monitors ( DestinationReached |
  DetermineLocation::location == MoveToLocation::target );
} ;
```

Listing 2. Reactive skills movement.

Skills can communicate with each other *via* the skills manager. The source and destination of the communication (skills' inputs and outputs) are determined in a `connections` clause. Each connection is unidirectional and relates an input of one skill to the output of another. Our example declares a connection from the `ColourVision` output `envColourPoints` to the `DetermineLocation's image` input.

Layers that depend on a reactive-skills control layer may need to monitor for particular conditions becoming true. To minimise the frequency at which the dependant layer needs to check the conditions, the reactive-skills pattern provides events that are independently triggered to notify the dependant layer of the occurrence of any monitored conditions. The `monitors` clause declares the monitors for the layer. They have a name and specify the logical condition to be monitored in terms of skill outputs and parameters. For our example, a condition that is monitored is the arrival of the delivery robot at the target location. A monitor `DestinationReached` has a condition that evaluates to true when the `location` output of the `DetermineLocation` skill is equal to the `target` parameter of a `MoveToLocation` skill.

Next we describe the RoboArch metamodel and well-formedness conditions for reactive-skills designs.

## 4.2 Metamodel and well-formedness conditions

The class ReactiveSkills representing a reactive-skills design is a subclass of Pattern (see Figure 3). Figure 6 details its definition, giving a precise characterisation of the pattern.

In a ReactiveSkills design there must be at least two skills and exactly one skillsManager. Skills can have parameters, inputs, and outputs all represented by Variables. Skills can be asynchronous and have a priority.

Skill is an abstract class (it has no objects). Its subclasses CSkill and DSkill are the concrete classes, whose objects are subject to different well-formedness conditions presented below.

The SkillsManager establishes one or more interskillconnections and may have stateMonitors. Monitors have a name and a condition defined by a RoboChart expression. SkillConnections associate Skills defining the start and end of the connections, the output startOutput of start, and the input endInput of end.

The well-formedness conditions that apply to reactive-skills designs are presented in Table 5. RS1 and RS2 ensure the use of reactive skills as intended to provide the essential behaviours that use the sensors and actuators (*via* the services of robotic platform), which other layers depend on to carry out the robot's tasks. RS3 records that the inputs of the D-Skills correspond to events of the robotic platform. RS4 is needed because a C-Skill or D-Skill in isolation can perform no meaningful function that alters the state of the robot or its environment. A C-Skill requires a D-Skill in order to interact with a sensor or actuator *via* the services of the robotic platform. With RS5 and RS6, we ensure that every skill contributes to the behaviour of the system. RS7 to RS10 ensure that connections are between inputs and outputs of different skills of the right type. Finally, RS11 ensures that monitors are concerned with skill data.

Valid reactive-skill designs, that is, those that satisfy the above well-formedness conditions, can be transformed to (and so formally described as) a RoboChart model, as described in the next section.

## 4.3 Reactive skills in RoboChart

Rules that can be used to transform a reactive-skill design to RoboChart are available[5]. Here we give an overview of our approach formalised by the rules in modelling reactive-skill designs in RoboChart.

A RoboChart controller representing a layer that uses the reactive-skills pattern has one state machine for the skills manager, and one machine for each skill. The skills-manager machine has events to manage the activation and deactivation of skills, receive parameter values, and communicate monitor-event and information replies. A skill machine has events for each of its inputs, outputs, and parameters.

As an example, Figure 7 shows the machine for Proximity in Listing 2. That machine reflects the description of the design pattern summarised in Table 4, and is representative of the state machines that are automatically generated for D-Skills. The declaration at the top in Figure 7 introduces variables priorityParam, to record an input priority value, envPoints, to record the output of the skill, and booleans priorityParaminitialised and envPointsSenseReceived, recording information about inputs. An interface IProximity declares the events used to exchange information with the skills-manager machine.

A D-Skill state machine starts at the state Initialise, where it accepts a priority for the skill *via* an event: in our example, proximityPriorityParam. When that input is taken, the variable priorityParaminitialised is updated to record that. Once that variable has value true, a transition to the state Deactivated becomes enabled, and is taken. In Deactivated, the priority can still be updated, until the skills manager raises an activate event (activateProximity in the example), when the machine moves to the state Startup.

Typically, the designer needs to enrich the state Startup to add the actions that the skill carries out at start up, perhaps *via* an entry action, or *via* a state machine making Startup a composite state. When those actions complete their execution, a D-Skill state machine moves to the state Ready.

In Ready, a new priority and inputs from the platform may be received. In the example, the value envPoints may be received from the platform *via* an event platformEnvPoints. When that happens, the value of a corresponding Boolean variable, here envPointsSenseReceived, is updated. This can go on until the skills manager raises an execute event (executeProximity, in the example) when the machine moves to the state HandleActuationCommands. Actions there, defined by the designer, might deal with buffering, for example. If, however, no input has been received (just not envPointsSenseReceived in Proximity), the machine flags that the skill has completed its task (completeProximity) and goes back to Ready.

If an input has been received, the machine moves to HandleSensorData. In general, HandleSensorData may deal with several pieces of data coming from the platform. All those that have been received may be communicated to another skill, together with its priority. In our example, we have just

5 https://robostar.cs.york.ac.uk/publications/reports/roboarch_rules.pdf

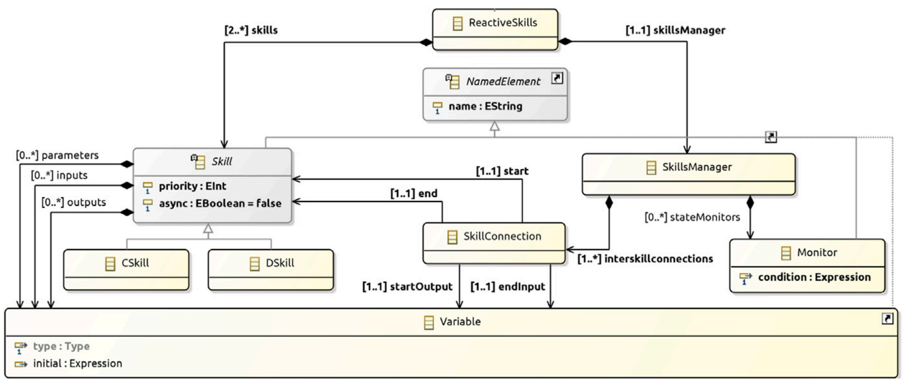


FIGURE 6
Reactive skills metamodel.

TABLE 5 The well-formedness conditions of reactive-skills designs.

| Condition | Description |
|---|---|
| RS1 | A Layer that has a pattern of type ReactiveSkills must be a GenericLayer or ControlLayer |
| RS2 | For a Layer with pattern type ReactiveSkills, at least one of the System's connections is from that layer to a RoboticPlatform or that layer has at least one rinterface |
| RS3 | For each event of ReactiveSkills Layer's interfaces, there must be a DSkill input with a matching name |
| RS4 | ReactiveSkills must contain a CSkill and a DSkill |
| RS5 | A CSkill must have at least one output |
| RS6 | A DSkill must have at least one output *or* input |
| RS7 | The start and end Skill of a SkillConnection must be distinct |
| RS8 | The startOutput of a SkillConnection must be an output of its start Skill |
| RS9 | The endInput of a SkillConnection must be an input of its end Skill |
| RS10 | The types of the startOutput and endInput of a SkillConnection must match |
| RS11 | A Monitor's condition must only refer to parameters, inputs, and outputs of the Skills |

envPoints, which is output *via* proximityEnvPoints. When that happens, the value of envPointsSenseReceived is updated back to false. When all data has been communicated, a D-Skill machine moves back to Ready.

Variations of the D-Skill state-machine definition take into account D-Skills that can output to the platform, and also D-Skills that have several inputs or several outputs.

A machine for a C-skill is shown in Figure 8. It is very similar to that of a D-Skill; the difference is that, instead of states HandleActuationCommands and HandleSensorData to deal with inputs and outputs of the platform, we have a single state ComputeOutputs. When the skills manager raises the execute event (executeDetermineLocation in the example), the machine moves to ComputeOutputs.

The designer must complete the definition of this state to reflect the calculations to be carried out by the skill. Once they finish executing, the machine returns to the state Ready, having signalled completion to the skills manager *via* a complete event: completeDetermineLocation in the example.

Finally, Figure 9 sketches the machine for the skills manager. The complete machine for our example that can be automatically generated is too large to include here. In the sketch, we show that a skills-manager machine starts in the state Initialise, in which it sets local variables, such as cycleSkills, recording the skills to execute in the next cycle. Afterwards, the skills-manager machine moves to HandleRequests.

In the state HandleRequests, for each request, there is a transition triggered by an event that represents a request from the dependant layer, whose transition action provides the required information, or updates variables to record the request: activate or deactivate skills, initiate event monitor, stop event monitor, or set skill parameters. Once the amount of time defined by the cycle of the skill manager is past, the machine moves to DoNextSkill. The cycle time is defined by a constant, whose value can be defined by the designer or left unspecified (until simulation or code generation).

DoNextSkill is a composite state that uses the cycleSkills variable to start all skills that are to execute in the current cycle. In the DoNextSkill machine, there is a state for each skill that sends its input values, raises the event that starts its execution (such as executeDetermineLocation) and updates a variable executingSkills. When all skills are set, cycleSkills gets empty, and the transition to ExecutingSkills is taken.

The state ExecutingSkills accepts the outputs of skills while they are executing. When an output is received, the machine moves to the state UpdateRecord, where the inputs to which the received output is connected are updated. This is done using

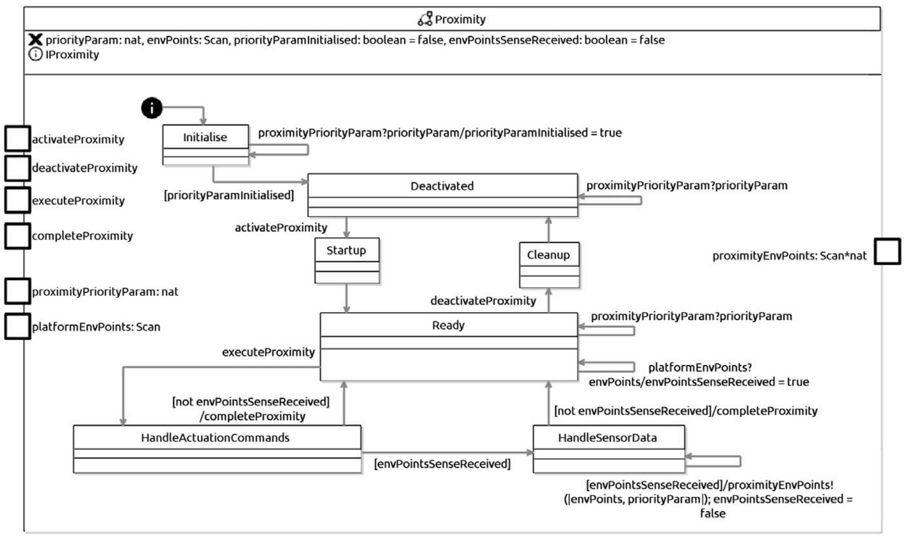

Proximity
priorityParam: nat, envPoints: Scan, priorityParamInitialised: boolean = false, envPointsSenseReceived: boolean = false
IProximity
activateProximity
deactivateProximity
executeProximity
completeProximity
proximityPriorityParam: nat
platformEnvPoints: Scan
proximityEnvPoints: Scan*nat
Initialise
proximityPriorityParam?priorityParam/priorityParamInitialised = true
[priorityParamInitialised]
Deactivated
proximityPriorityParam?priorityParam
activateProximity
Startup
Cleanup
deactivateProximity
Ready
proximityPriorityParam?priorityParam
executeProximity
platformEnvPoints?envPoints/envPointsSenseReceived = true
[not envPointsSenseReceived]/completeProximity
[not envPointsSenseReceived]/completeProximity
HandleActuationCommands
HandleSensorData
[envPointsSenseReceived]
[envPointsSenseReceived]/proximityEnvPoints!(|envPoints, priorityParam|); envPointsSenseReceived = false

FIGURE 7
RoboChart state machine for Proximity D-Skill.

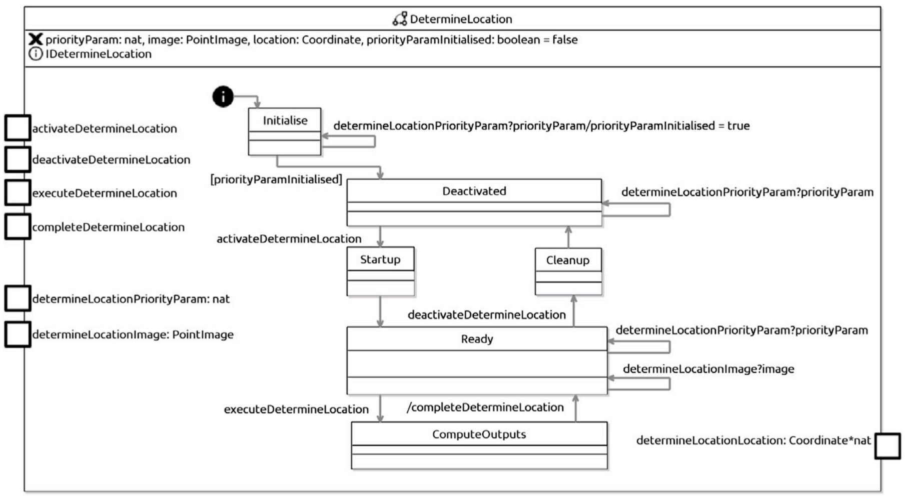

DetermineLocation
priorityParam: nat, image: PointImage, location: Coordinate, priorityParamInitialised: boolean = false
IDetermineLocation
activateDetermineLocation
deactivateDetermineLocation
executeDetermineLocation
completeDetermineLocation
determineLocationPriorityParam: nat
determineLocationImage: PointImage
determineLocationLocation: Coordinate*nat
Initialise
determineLocationPriorityParam?priorityParam/priorityParamInitialised = true
[priorityParamInitialised]
Deactivated
determineLocationPriorityParam?priorityParam
activateDetermineLocation
Startup
Cleanup
deactivateDetermineLocation
Ready
determineLocationPriorityParam?priorityParam
determineLocationImage?image
executeDetermineLocation
/completeDetermineLocation
ComputeOutputs

FIGURE 8
RoboChart state machine for DetermineLocation C-Skill.

an UpdateValue function (omitted here) that only updates the input if it is the first update of the cycle or if the new value comes from a skill with higher priority. After each update, the machine moves back to ExecutingSkills.

The state ExecutingSkills also accepts completion events from the skills (such as completeDetermineLocation), updating the executingSkills variable after each such event. When all skills have completed execution, executingSkills is empty and the transition to CheckMonitors is taken.

In CheckMonitors, there is a transition for each monitor. If a monitor condition occurs, a corresponding event notifies the depending layer. When all monitors are checked, the machine moves back to the state HandleRequests, after reinitialising variables such as cycleSkills.

Using the semantics of RoboChart that is automatically generated, we can prove properties of the design. We have, for example, proved deadlock and livelock freedom, and some other trace-based properties of some of the machines that are

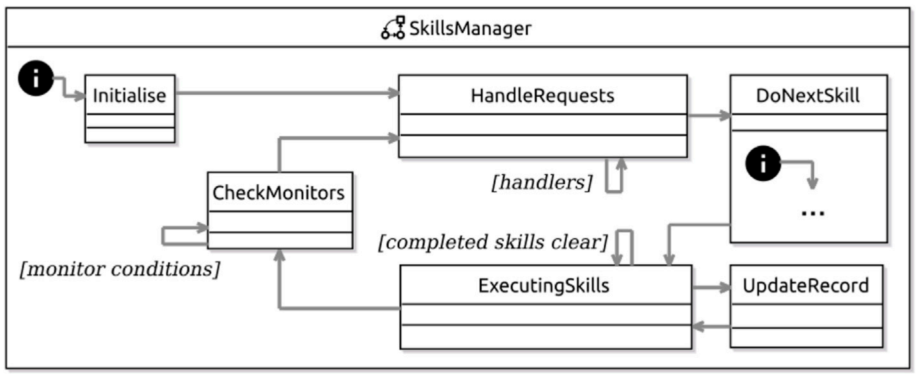


FIGURE 9
RoboChart skills-manager machine for the example.

automatically generated. In these proofs, we can cater for general properties of any design, and for application-specific properties.

In the next section, we discuss how we can define and formalise a CorteX-based design pattern in RoboArch, opening the same possibilities for CorteX and CorteX designs (see Figure 1).

## 5 Discussion: CorteX and RoboArch

As already said, CorteX is a framework tailored to the development of complex nuclear robotic systems. It primarily focuses on data representation and communication to solve issues of maintainability and extensibility. In this section, we discuss the integration of CorteX and RoboArch.

We envisage two main approaches for integrating RoboArch and CorteX. The first supports the generation of CorteX implementations of RoboArch models (Section 5.1). The second approach extends RoboArch to support modelling CorteX architectures (Section 5.2).

### 5.1 From RoboArch to CorteX

As discussed in Sections 3, 4, the semantics of RoboArch is specified in terms of RoboChart, which opens the possibility for the generation of several artefacts (see Figure 1). We can obtain automatically mathematical models for verification, such as CSP (Miyazawa et al., 2019) scripts, for verification of reactive and timed properties, and PRISM (Ye et al., 2021) reactive modules, for verification of probabilistic properties. We can also obtain code (Li et al., 2018) and RoboSim models describing simulations (Cavalcanti et al., 2019). RoboSim is a sister notation of RoboChart tailored to the design and verification of simulations with a similar component model and artefact-generation facilities.

A code generator that produces CorteX-compatible implementations of a RoboArch model can take advantage of some of the abovementioned functionalities. The first step requires the generation of the semantics of the RoboArch model in RoboChart as described in this paper. Since CorteX is a cyclic architecture, it is useful to transform (automatically) the resulting RoboChart model into a simulation model, written in RoboSim, *via* the RoboStar correctness-preserving model-to-model transformation. Next, we can use one of the RoboSim model-to-model transformations to generate an intermediate representation of imperative code and a model-to-text transformation tailored for CorteX. Currently, two transformations targeting the programming languages C and Rust are under development.

With the use of the intermediate representation, we guarantee that the semantics of RoboChart and RoboSim is preserved by the code. This follows from the fact that the generation of the intermediate representation is a mechanisation of the RoboSim semantics, and the model-to-text transformation is direct. For CorteX, each state machine can be implemented as a simplex, the basic unit of data and behaviour in CorteX code. This approach matches well the parallel paradigms of RoboChart and CorteX.

On the other hand, the translations from RoboChart to RoboSim and from RoboSim to the intermediary representation give rise to additional parallel components for orchestration of operation calls and during actions inside state machines. This can create an overhead in the target code. If this overhead becomes an issue, we can alternatively, directly convert the RoboSim model into code *via* a generator specifically tailored for CorteX. While this alternative involves significantly more work (as it does not reuse the existing intermediate representation generator), it allows for more control over the structure of the CorteX implementation, and a one-to-one match between state machines and simplexes.

In the approaches above, CorteX is used as a target middleware. An alternative explored in the next section is the use of CorteX concepts already at the design level, giving rise to an architectural pattern for CorteX. This enables design and verification for CorteX.

## 5.2 CorteX in RoboArch

A CorteX implementation does not explicitly have the notion of layers. In fact, one might even argue that a layered architecture is incompatible with CorteX due to its distributed nature. This, however, is not the case, since layers are not necessarily centralised or co-located, and a layer or set of layers can be deployed as a distributed system. Moreover, well-designed code separates planning and control functionality. It is, therefore, beneficial to use separate sets of CorteX simplexes to deal with planning and control.

For this reason, the use of layers does not prevent the adoption of CorteX, and, moreover, embedding CorteX designs in RoboArch as a pattern for any layer provides extra support to address the interoperability issue with non-CorteX applications such as ROS (Caliskanelli et al., 2021, *p.* 320). The use of a layered RoboArch design can help to ensure not only that code for planning and control is kept separate, but that a strict layered discipline is enforced, even if the code, as it is often the case, does not have a notion of layer.

Figure 10 depicts a metamodel for integrating CorteX into RoboArch; it is based on the description of CorteX in (Caliskanelli et al., 2021). As for reactive skills, we model the CorteX architecture as a RoboArch Pattern. Listing 3 and Listing 4 present the sketch of a layer CorTeXl that uses the CorteX pattern in the design of a simple application based on mobile robots inspired by an example in (Caliskanelli et al., 2021).

```
ontology MobileRobots
dmodule Object {
 dmodule MobileRobot {
  dmodule MobileBase {
   relationships {
    owns RotaryAxisConcept left
    owns RotaryAxisConcept right
   }
  }
 }
 ...
 amodule Controller {
  amodule MovementController {
   relationships {
    input ObstacleConcept obstacle
    output RotaryAxisConcept left
    output RotaryAxisConcept right
   }
  }
 }
}
```

Listing 3. Mobile robots ontology.

```
layer CorteXl: ControlLayer {
 requires IMove
 uses ISensor
 pattern CorteX;
 ontology MobileRobots
 simplexes
  r: Car {
   satisfies base b
   satisfies obstacle o
  }
  ...
  m: MovementController {
   satisfies input obstacle oc
   satisfies output left lc
   satisfies output right rc
  }
}
```

Listing 4. Example of layer using the `CorteX` pattern with `MobileRobots` ontology.

As discussed in Caliskanelli et al. (2021, pp. 317–319), a CorteX application is parameterised by an ontology, represented here by the attribute ontology. An object of class Ontology has a single attribute root of type CorteXType, which is an abstract class that can be realised as either a DescriptiveModule or an ActiveModule. The distinction is similar to that between passive and active classes.

A CorteXType may contain any number of CortexType children and sets of rules applicable to commands, data, and relationships. An ontology is, therefore, structured as a tree, a hierarchy of concepts akin to an object-oriented model. The rules establish constraints over CorteXTypes. A CommandRule has an identifier (for instance, move) and some parameters. Each ParameterRule defining a parameter has itself an identifier and a dataType (limited to integer, float, Boolean, or string), possibly an array, as defined by the attributes minCount and maxCount. We omit here the simple enumeration CortexDataType.

A DataRule is similar to a ParameterRule. Finally, a RelationshipRule describes a relationship with a CorteXType, defined by relatedType. It specifies a direction, using a value of an enumeration type CorteXRelationshipDirection including INPUT and OUTPUT, and a multiplicity.

Listing 3 shows an excerpt of the ontology for the example. It includes descriptive modules, such as `MobileBase`, and active modules, such as `MovementController`. `MobileBase` represents a two-wheeled robot and contains two concepts of type `RotaryAxisConcept`, which represent the data associated with the left and right wheels. The module `MovementController` specifies an input concept of type `ObstacleConcept` and two output concepts of type `RotaryAxisConcept` (both of these concepts are specified in the ontology as descriptive modules, but omitted in Listing 3).

A CorteX pattern also contains a set of simplexes. A Simplex has an identifier, a type from the ontology, and sets

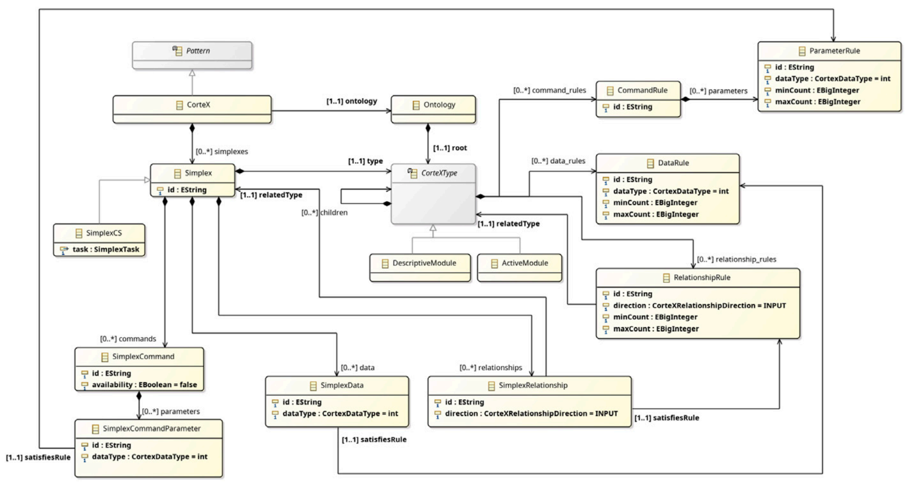


FIGURE 10
Metamodel of CorteX for integration with RoboArch.

of data, relationships, and commands. A SimplexData models a piece of primitive data containing an identifier and a dataType (limited to integer, float, Boolean, string, and possibly an array). The attribute satisfiesRule identifies a DataRule of the ontology that is implemented (satisfied) by the simplex.

A SimplexCommand has an identifier, availability, and a set of parameters. Like for SimplexData and DataRule, a SimplexCommandParameter satisfies a ParameterRule.

A SimplexRelationship describes a connection between two simplexes, namely, the simplex that contains it and the simplex identified by relatedType. In addition, a SimplexRelationship satisfies a RelationshipRule.

Listing 4 depicts the RoboArch control layer `CorTeXl` that uses the `CorteX` pattern and refers to the `MobileRobots` ontology in Listing 3. It requires interfaces `IMove` and `ISensor`; the first declares operations `setLeftMotorSpeed` and `setRightMotorSpeed`, and the second the event `obstacle`. Next, `CorTeXl` specifies its pattern (`CorteX`) and the pattern's components. These are the `ontology (MobileRobots)` and the set of `simplexes`. Each simplex has a name and a type from the ontology, and information about how the ontology `relationship` rules are satisfied. For instance, `MovementController` has name `m` and specifies three relationships; the first specifies that the relationship rule `left` (of the module `MovementController` in Listing 3) is satisfied by the simplex `lc` (declared in the pattern but omitted in Listing 4) of type `RotaryAxisConcept`.

A SimplexCS is a Simplex with a notion of task, which defines behaviours to be executed in particular points of its lifecycle. This is similar to D-Skills in the reactive-skills pattern, where the top-level execution protocol of the D-Skill is fixed and user-defined behaviours are run in particular stages of this protocol.

In our CorteX metamodel, we omit the concept of a ClusterCS, which is related to allocation of simplexes to computational units. This is an issue not covered in RoboStar technology. Automatic generation of CorteX code may, for example, define a simple default allocation of simplexes to a single computational unit for further elaboration by the CorteX designer later.

Additionally, we omit the notion of Simplex Trees. These are sets of simplexes, which are represented in our metamodel by the attribute simplexes of CorteX. So, each layer that uses a CorteX pattern has a single set of simplexes. With this metamodel, different sets can, and need to be, allocated in different layers. Further experience may indicate that we need several sets of simplexes in a layer, if the layer discipline turns out to be too restrictive in some cases. This simple extension is left as future work.

There are three well-formedness conditions that apply to a CorteX design as defined below. They are all related to the data, commands, and relationships of a Simplex and the rules that they indicate that are satisfied by them. Together the conditions ensure that the rules used in a Simplex are well defined.

**C1** The DataRule of a SimplexData is in the CorteXType of its Simplex.

**C2** The ParameterRule of a SimplexCommandParameter is in the CorteXType of its Simplex.

**C3** The RelationshipRule of a SimplexRelationship is in the CorteXType of its Simplex. In addition, the Simplex defined by its relatedType has the type defined by the relatedType of the RelationshipRule.

For designs that satisfy these restrictions, we can define a RoboChart sketch *via* transformation rules. The semantics of the CorteX pattern would be specified in RoboChart in line with the semantics of RoboArch. Each descriptive Simplex, that is, a simplex whose type is a DescriptiveModule, is specified by a RoboChart data type, and each active Simplex (ActiveModule type) is defined by a state machine, where the SimplexCommands are modelled as events, the SimplexData as variables, and the SimplexRelationships as connections. The semantics of a SimplexCS is specified by a state machine that enforces the specific execution protocol in a similar manner as done for D-Skills, which is illustrated in Figure 7.

As indicated in Figure 3, a layer contains input and output events for inter-layer communication. CorteX, on the other hand, does not use the same communication mechanism and requires a component to transform and route data between the layer and the CorteX application. This component can also be automatically generated similarly to how the semantics of RoboArch specifies the SkillsManager (Figure 9) for the reactive-skills architecture. Such a component partially solves the interoperability between different architectures; for instance, it allows the control layer in Figure 4, which uses the reactive-skills pattern, to communicate with an executive layer that uses a CorteX pattern.

To conclude, by allying RoboArch and CorteX, we can support the use of CorteX principles from an early stage of design. We can also support verification and automatic code generation. In this way, we further the CorteX agenda by supporting the development of traceable evidence of core properties of applications. Future work will consider significant case studies and automation.

## Data availability statement

Publicly available additional rules were used in this study. They can be found at: https://robostar.cs.york.ac.uk/publications/reports/roboarch_rules.pdf.

## Author contributions

WB, AC, and AM contributed to conception and design of the work. WB took a lead in its execution: definition of metamodel, well-formedness conditions, and transformations. WB wrote the first draft of Sections 1–4; we all contributed to the draft and its revision, and read and approved the submitted version.

## Funding

The work is funded by the United Kingdom EPSRC Grants EP/M025756/1, EP/R025479/1, and EP/V026801/2, and by the United Kingdom Royal Academy of Engineering Grant No CiET1718/45.

## Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## Publisher's note

## References

Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (1998). An architecture for autonomy. *Int. J. Rob. Res.* 17, 315–337. doi:10.1177/027836499801700402

Albus, J. S., Lumia, R., Fiala, J., and Wavering, A. J. (1989). "Nasrem – The NASA/NBS standard reference model for telerobot control system architecture," in *Industrial robots*.

Álvarez, B., Sánchez-Palma, P., Pastor, J. A., and Ortiz, F. (2006). An architectural framework for modeling teleoperated service robots. *Robotica* 24, 411–418. doi:10.1017/s0263574705002407

Ando, N., Suehiro, T., and Kotoku, T. (2008). "A software platform for component based rt-system development: Openrtm-aist," in *Simulation, modeling, and programming for autonomous robots*. Editors S. Carpin, I. Noda, E. Pagello, M. Reggiani, and O. von Stryk (Springer), 87–98.

Arkin, R. C. (1989). Motor schema — Based mobile robot navigation. *Int. J. Rob. Res.* 8, 92–112. doi:10.1177/027836498900800406

Arkin, R. C. (1987). *Towards cosmopolitan robots: Intelligent navigation in extended man-made environments*. Ph.D. thesis. Amherst: University of Massachusetts.

Backes, P., Edelberg, K., Vieira, P., Kim, W., Brinkman, A., Brooks, S., et al. (2018). "The intelligent robotics system architecture applied to robotics testbeds and research platforms," in *IEEE aerospace conference* (IEEE Computer Society).

Bass, L., Clements, P., and Kazman, R. (2012). *Software architecture in practice*. Upper Saddle River, NJ: Pearson Education.

Bonasso, R. P., Firby, R. J., Gat, E., Kortenkamp, D., Miller, D. P., and Slack, M. G. (1997). Experiences with an architecture for intelligent, reactive agents. *J. Exp. Theor. Artif. Intell.* 9, 237–256. doi:10.1080/095281397147103

Bonasso, R. P. (1991). "Integrating reaction plans and layered competences through synchronous control," in *12th international joint conference on artificial intelligence* (San Francisco, CA: Morgan Kaufmann Publishers Inc.), 1225–1231.

Bonasso, R. P., Kerri, R., Jenks, K., and Johnson, G. (1998). "Using the 3T architecture for tracking shuttle RMS procedures," in *IEEE international joint symposia on intelligence and systems*.

Bonato, V., and Marques, E. (2009). Roboarch: A component-based tool proposal for developing hardware architecture for mobile robots. *IEEE Int. Symposium Industrial Embed. Syst.*, 249–252.

Borrelly, J.-J., Coste-Manière, E., Espiau, B., Kapellos, K., Pissard-Gibollet, R., Simon, D., et al. (1998). The ORCCAD architecture. *Int. J. Rob. Res.* 17, 338–359. doi:10.1177/027836499801700403

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE J. Robot. Autom.* 2, 14–23. doi:10.1109/jra.1986.1087032

Bruyninckx, H., Klotzbücher, M., Hochgeschwender, N., Kraetzschmar, G., Gherardi, L., and Brugali, D. (2013). "The BRICS component model: A model-based development paradigm for complex robotics software systems," in *28th annual ACM symposium on applied computing* (New York, NY: ACM), 1758–1764.

Bruyninckx, H. (2001). Open robot control software: The orocos project. *IEEE Int. Conf. Robotics Automation* 3, 2523–2528.

Caliskanelli, I., Goodliffe, M., Whiffin, C., Xymitoulias, M., Whittaker, E., Verma, S., et al. (2021). *CorteX: A software framework for interoperable, plug-and-play, distributed, robotic systems of systems*. Springer, 295–344.

Cavalcanti, A. L. C., Barnett, W., Baxter, J., Carvalho, G., Filho, M. C., Miyazawa, A., et al. (2021a). *RoboStar Technology: A roboticist's toolbox for combined proof, simulation, and testing*. Springer International Publishing, 249–293. doi:10.1007/978-3-030-66494-7_9

Cavalcanti, A. L. C., Dongol, B., Hierons, R., Timmis, J., and Woodcock, J. C. P. (Editors) (2021b). *Software engineering for robotics* (Springer International Publishing). doi:10.1007/978-3-030-66494-7

Cavalcanti, A. L. C., Sampaio, A. C. A., Miyazawa, A., Ribeiro, P., Filho, M. C., Didier, A., et al. (2019). Verified simulation for robotics. *Sci. Comput. Program.* 174, 1–37. doi:10.1016/j.scico.2019.01.004

Chatila, R., Lacroix, S., Simeon, T., and Herrb, M. (1995). Planetary exploration by a mobile robot: Mission teleprogramming and autonomous navigation. *Auton. Robots* 2, 333–344. doi:10.1007/bf00710798

Chien, S., Knight, R., Stechert, A., Sherwood, R., and Rabideau, G. (2000). "Using iterative repair to improve the responsiveness of planning and scheduling," in *5th international conference on artificial intelligence planning systems* (Menlo Park, CA: AAAI Press), 300–307.

Chitta, S., Marder-Eppstein, E., Meeussen, W., Pradeep, V., Tsouroukdissian, A. R., Bohren, J., et al. (2017). ros_control: A generic and simple control framework for ROS. *J. Open Source Softw.* 2, 456. doi:10.21105/joss.00456

Corke, P., Sikka, P., Roberts, J. M., and Duff, E. (2004). "Ddx : A distributed software architecture for robotic systems," in *Australasian conference on robotics & automation*. Editors N. Barnes, and D. Austin (Sydney, NSW: Australian Robotics & Automation Association).

Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., and Ziane, M. (2012). *Simulation, modeling, and programming for autonomous robots*. SpringerSimulate and Deploy Robotic Applications, 149–160. chap. RobotML, a Domain-Specific Language to Design.

Firby, R. J., Kahn, R. E., Prokopowicz, P. N., and Swain, M. J. (1995a). "An architecture for vision and action," in *14th Int. Jt. Conf. Artif. Intell*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 1, 72–79.

Firby, R. J., Slack, M. G., and Drive, C. (1995b). "Task execution: Interfacing to reactive skill networks," in *Lessons learned from implemented software architectures for physical agents: Papers from the 1995 spring symposium*. Editor K. D. H. Henry, 97–111. Technical Report SS-95-02.

Franz, T., Lüdtke, D., Maibaum, O., and Gerndt, A. (2018). Model-based software engineering for an optical navigation system for spacecraft. *CEAS Space J.* 10, 147–156. doi:10.1007/s12567-017-0173-5

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns - elements of reusable object-oriented software*. Addison-Wesley.

García, S., Menghi, C., Pelliccione, P., Berger, T., and Wohlrab, R. (2018). An architecture for decentralized, collaborative, and autonomous robots. *IEEE Int. Conf. Softw. Archit.*, 75–7509.

Gat, E. (1992). "Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots," in *10th national conference on artificial intelligence* (Menlo Park, CA: AAAI Press), 809–815.

Huntsberger, T., and Woodward, G. (2011). *OCEANS'11 MTS/IEEE KONA*, 1–10. Intelligent autonomy for unmanned surface and underwater vehicles.

Li, W., Ribeiro, A. M. P., Cavalcanti, A. L. C., Woodcock, J. C. P., and Timmis, J. (2018). *From formalised state machines to implementations of robotic controllers*. Springer International Publishing, 517–529. doi:10.1007/978-3-319-73008-0_36

Luckcuck, M., Farrell, M., Dennis, L. A., Dixon, C., and Fisher, M. (2019). Formal specification and verification of autonomous robotic systems: A survey. *ACM Comput. Surv.* 52, 1–41. doi:10.1145/3342355

Luzeaux, D., and Dalgalarrondo, A. (2001). "HARPIC, an hybrid architecture based on representations, perceptions, and intelligent control: A way to provide autonomy to robots," in *Computational science* (Springer), 327–336.

Lyons, D. M., and Hendriks, A. J. (1995). Planning as incremental adaptation of a reactive system. *Robotics Aut. Syst.* 14, 255–288. doi:10.1016/0921-8890(94)00033-x

Metta, G., Fitzpatrick, P., and Natale, L. (2006). Yarp: Yet another robot platform. *Int. J. Adv. Robotic Syst.* 3, 8. doi:10.5772/5761

Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A. L. C., and Timmis, J. (2017). Automatic property checking of robotic applications. *IEEE/RSJ Int. Conf. Intelligent Robots Syst.*, 3869–3876. doi:10.1109/IROS.2017.8206238

Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A. L. C., Timmis, J., and Woodcock, J. C. P. (2019). RoboChart: Modelling and verification of the functional behaviour of robotic applications. *Softw. Syst. Model.* 18, 3097–3149. doi:10.1007/s10270-018-00710-z

Miyazawa, A., Ribeiro, P., Ye, K., Cavalcanti, A. L. C., Li, W., Timmis, J., et al. (2020). *RoboChart: Modelling, verification and simulation for robotics*. York, UK: Tech. rep., University of York, Department of Computer Science. Available at www.cs.york.ac.uk/robostar/notations/.

Muratore, L., Laurenzi, A., Hoffman, E. M., Rocchi, A., Caldwell, D. G., and Tsagarakis, N. G. (2017). Xbotcore: A real-time cross-robot software platform. *IEEE Int. Conf. Robotic Comput.*, 77–80.

Muscettola, N., Nayak, P. P., Pell, B., and Williams, B. C. (1998). Remote agent: To boldly go where no AI system has gone before. *Artif. Intell.* 103, 5–47. doi:10.1016/s0004-3702(98)00068-x

Musliner, D. J., Durfee, E. H., and Shin, K. G. (1993). Circa: A cooperative intelligent real-time control architecture. *IEEE Trans. Syst. Man. Cybern.* 23, 1561–1574. doi:10.1109/21.257754

Nesnas, I. A. D., Simmons, R., Gaines, D., Kunz, C., Diazcalderon, A., Estlin, T., et al. (2006). CLARAty: Challenges and steps toward reusable robotic software. *Int. J. Adv. Robotic Syst.* 3, 5–030. doi:10.5772/5766

Nordmann, A., Hochgeschwender, N., Wigand, D., and Wrede, S. (2016). A survey on domain-specific modeling and languages in robotics. *J. Softw. Eng. Robotics* 7, 75–99.

Sanchez-Lopez, J. L., Molina, M., Bavle, H., Sampedro, C., Fernández, R. A. S., and Campoy, P. (2017). A Multi-Layered Component-Based approach for the development of aerial robotic systems: The aerostack framework. *J. Intell. Robot. Syst.* 88, 683–709. doi:10.1007/s10846-017-0551-4

Sellner, B., Heger, F. W., Hiatt, L. M., Simmons, R., and Singh, S. (2006). Coordinated multiagent teams and sliding autonomy for large-scale assembly. *Proc. IEEE* 94, 1425–1444. doi:10.1109/jproc.2006.876966

Siciliano, B., and Khatib, O. (Editors) (2016). *Springer handbook of robotics* (Springer Handbooks Springer).

Silva, L., Yan, R., Ingrand, F., Alami, R., and Bensalem, S. (2015). "A verifiable and correct-by-construction controller for robots in human environments," in 10th annual ACM/IEEE international Conference on human-robot interaction extended abstracts *(ACM), HRI'15 extended abstracts*, 281.

Stamper, D., Lotz, A., Lutz, M., and Schlegel, C. (2016). The SmartMDSD toolchain: An integrated MDSD workflow and integrated development environment (IDE) for robotics software. *J. Softw. Eng. Robotics* 7, 3–19.

Volpe, R., Nesnas, I., Estlin, T., Mutz, D., Petras, R., and Das, H. (2001). The CLARAty architecture for robotic autonomy. *IEEE Aerosp. Conf.* 1.

Wigand, D. L., Mohammadi, P., Hoffman, E. M., Tsagarakis, N. G., Steil, J. J., and Wrede, S. (2018). "An open-source architecture for simulation, execution and analysis of real-time robotics systems," in *IEEE international conference on simulation, modeling, and programming for autonomous robots*, 93–100.

Wong, C., Kortenkamp, D., and Speich, M. (1995). "A mobile robot that recognizes people," in *7th IEEE international conference on tools with artificial intelligence*, 346–353.

Woodcock, J. C. P., and Davies, J. (1996). *Using Z - specification, refinement, and proof*. Prentice-Hall.

Ye, K., Cavalcanti, A. L. C., Foster, S., Miyazawa, A., and Woodcock, J. C. P. (2021). Probabilistic modelling and verification using RoboChart and PRISM. *Softw. Syst. Model.* 21, 667–716. doi:10.1007/s10270-021-00916-8

Yu, S. T., Slack, M. G., and Miller, D. P. (1994). "A streamlined software environment for situated skills," in *AIAA/NASA conference on intelligent robotics in field* (Houston, TX: Factory, Service, and Space NASA), 233–239.

TYPE Original Research
PUBLISHED 24 April 2024
DOI 10.3389/frobt.2024.1358978

Check for updates

OPEN ACCESS

EDITED BY
Ivano Malavolta,
VU Amsterdam, Netherlands

REVIEWED BY
Nico Hochgeschwender,
University of Bremen, Germany
Elvin Alberts,
VU Amsterdam, Netherlands

*CORRESPONDENCE
Giuseppe Cotugno,
g.cotugno@ocado.com

RECEIVED 20 December 2023
ACCEPTED 08 April 2024
PUBLISHED 24 April 2024

CITATION
Cotugno G, Rodrigues RA, Deacon G and Konstantinova J (2024), A containerised approach for multiform robotic applications. *Front. Robot. AI* 11:1358978.
doi: 10.3389/frobt.2024.1358978

# A containerised approach for multiform robotic applications

Giuseppe Cotugno*, Rafael Afonso Rodrigues, Graham Deacon and Jelizaveta Konstantinova

Ocado Technology, Welwyn Garden City, United Kingdom

As the area of robotics achieves promising results, there is an increasing need to scale robotic software architectures towards real-world domains. Traditionally, robotic architectures are integrated using common frameworks, such as ROS. Therefore, systems with a uniform structure are produced, making it difficult to integrate third party contributions. Virtualisation technologies can simplify the problem, but their use is uncommon in robotics and general integration procedures are still missing. This paper proposes and evaluates a containerised approach for designing and integrating multiform robotic architectures. Our approach aims at augmenting preexisting architectures by including third party contributions. The integration complexity and computational performance of our approach is benchmarked on the EU H2020 SecondHands robotic architecture. Results demonstrate that our approach grants simplicity and flexibility of setup when compared to a non-virtualised version. The computational overhead of using our approach is negligible as resources were optimally exploited.

KEYWORDS

**robotic architectures, microservices, robot integration, virtualization, robot software design**

## 1 Introduction

### 1.1 Motivation

Recently, complete robotic solutions, such as collaborative robots (Asfour et al., 2018) or robotic warehouse automation (Hamberg and Verriet, 2012) are frequently deployed in real world scenarios (Cotugno et al., 2020). For example, the purpose of the EU H2020 SecondHands project[1] is to develop a humanoid collaborative robot (the ARMAR-6 (Asfour et al., 2018)) to assist a maintenance technician in servicing conveyor belts in a real-world warehouse (Figure 1). The software architecture that is powering such robots can be very complex, with several components interrelated and dependent among each other.

Traditionally, robotic frameworks, like ROS, are used to simplify the development and integration of robotic software architectures. However, relying on a singular robotic framework makes the resulting system uniform as the set of software libraries and development tools, used to develop and interconnect core components instrumental to the robot's autonomy, is predefined and could be embedded in the wider code base by design or necessity. In addition, every component must be developed with a predefined structure. The

1 SecondHands: A Robot Assistant For Industrial Maintenance Task, Website: https://secondhands.eu/

expectation from using such robotic frameworks is that components developed by a research group can be easily integrated to a different architecture which runs the same framework.

Integrating third party software is becoming even more critical today, as various contributions valuable for robotics are shared online. For example, in SecondHands, image segmentation is performed using MaskRCNN (He et al., 2017): a deep network developed for segmenting common objects which has been re-trained for detecting tools. Robotic household assistants, competing in the Robot@Home competition, implement speech understanding by integrating several language processing components developed for general use (Matamoros et al., 2018). Such contributions are often developed without following the integration rules imposed by a robotic framework and it can be difficult to include them in a large system. Such contributions are called *multiform* in this paper as they do not conform to the development rules of a robotic framework and are constituted by several heterogeneous components.

For these reasons, our paper proposes a general purpose approach which facilitates the inclusion and interoperation of third party software into an existing robotic architecture. Our approach relies on the use of containers, which are lightweight virtual machines. The contributions of this paper are as follows:

- A container-based methodology is proposed, which allows systematic and minimally invasive integration with an existing robotic framework;
- The use of the containerised approach is demonstrated on a real world application of collaborative robot within the SecondHands project;
- The overall system is evaluated against a non-virtualised version in terms of integration complexity and run-time computational performance.

This paper is asking the following research question: *Is it possible to systematically design or incrementally adapt a robotic system to include and interoperate existing third party components?* In this work, a *third party component* is defined as a contribution developed by different developers not directly involved with the integration of a specific robotic system. The component might not follow any integration rule established for the system, for example it could be written in a different unsupported language, use a different robotic framework or run standalone, use incompatible libraries or different build tools, etc. Such a component can be a prototype proven to work standalone, whose integration in a larger system might be not-trivial. This differs from the scenario where a component has been developed using a robotic framework as that framework imposes rules that must be respected for the component to be useable. It is important to underline that our methodology suggests a set of principles to facilitate the systematic integration of components as opposed to an *ad hoc* approach which integrates third party components differently and might have different outcomes in terms of simplicity of integration and performance cost for different components. In the Evaluation section of the paper we will test two hypotheses: *1. Following all the guidelines of our methodology simplifies the integration complexity* and *2. Following all the guidelines of our methodology has a noticeable performance cost.*

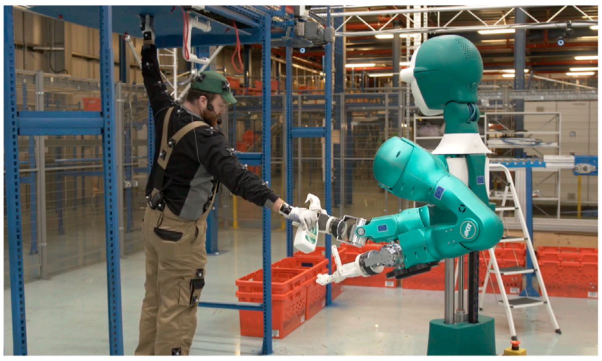


FIGURE 1
The robotic software architecture developed within the EU H2020 SecondHands project has to enable a humanoid robot to assist a technician during maintenance in real-world conditions. Its several parts constitute a multiform robotic system as they were not developed having a specific robotic framework in mind.

The structure of this paper is as follows: in Section 1.2 we compare our work to the state of the art, while in Section 2 our methodology is described and applied to the SecondHands robot architecture. Section 3 quantifies the integration complexity and runtime performance costs of using our methodology. Finally, Section 4.1 discusses our results and Section 4.2 summarises the key findings and limitations of the paper proposing avenues for future work.

## 1.2 Related work

Over the decades, the number of proprietary and open source robotic frameworks, used for software integration, has increased greatly. The YARP framework (Metta et al., 2006), is a cross-platform framework which mostly targets the humanoid robot iCub (Metta et al., 2008). The well-known ROS framework can interoperate with a large number of robots. Other less known frameworks are also targeted to a specific robot (NaoQi (Pot et al., 2009), Khepera III (Cotugno et al., 2011)), a family of robots (ArmarX (Vahrenkamp et al., 2015)) or a limited predefined selection of robots (OpenRDK (Calisi et al., 2008)). It is beyond the scope of this paper to propose a survey of the features of state-of-the-art robotic frameworks (Mohamed et al., 2008). However, we identify three common features: 1) implementation of a smallest entity able to provide some functionality (e.g. node, module), 2) definition of a communication protocol for exchanging information across those entities, 3) definition of a development methodology and standardised set of development tools. As a result, robotic architectures are deployed as distributed systems and are uniform: components have the same structure.

The homogeneity imposed by a robotic framework becomes a limitation when a robotic architecture includes third party software. Third party contributions have to be adapted to fit the design and tools imposed by the framework itself (Khandelwal et al., 2017). Yet such integrations might prove to be non-trivial (Cervera, 2019) and integrating components from two different frameworks adds complexity (Randazzo et al., 2018) even when the same hardware is used. For example, the SecondHands system integrates off-the-shelf deep networks and components developed differently by five research groups, making it a multiform system. In such system, it is impractical to re-implement every contribution to comply to the rules of a robotic framework. Code re-use can be achieved using virtualisation. In literature, two virtualisation approaches are popular: virtual machines (VMs) and containers. A VM is a software which emulates a PC both in its hardware and operative system. A container is a light-weight high performance (Seo et al., 2014) VM which shares with its host (the PC) only the kernel.

Within robotics, virtualisation has been pioneered by several authors. In Fres et al. (Fres and Alonso, 2010) a virtual machine is used to encapsulate the controller for a mobile robot using a novel programming language, while (Hinze et al., 2018) employs containers to run multiform grasping simulations with real-time control requirements. The limitation of these approaches is that the proposed systems are not designed to support a complex robotic architecture, as virtualisation is not used at its full power. In some cases, virtualisation is used merely as a tool to bypass a certain issue. For example, in (Rodrigues, 2017), containers are used for dynamical deployment of control software on mobile robots via the cloud, optimising the workload as needed. In (Liu et al., 2018), containers are used as a base infrastructure to support the learning and deployment of control strategies to insert pegs into holes. Those works solve a well-defined problem, however they cannot be applied to similar problems. Virtualisation here is used as a tool rather than being a structural part of the architecture.

Other works, instead, use virtualisation as a structural component of a larger framework. For example, Mohanarajah et al. (Mohanarajah et al., 2014) proposes a full robotic framework which relies on containers to execute different robot algorithms, while in (Turnbull and Samanta, 2013), virtual machines are used to provide different services for cloud robotics applications. The shortcoming of such approaches is that it is not described how the proposed frameworks can be integrated with preexisting robotic software without rewriting the old code. Design principles must be provided to guide the development of large multiform robotic architectures. Distributed systems can be used as a source of inspiration.

Modern distributed systems (e.g. cloud video or audio streaming platforms) implement a *microservice architecture* (Newman, 2015), where many heterogeneous components are deployed and networked across several machines to contain faults and balance high computational loads. A microservice-inspired framework has been already pioneered in (Wang et al., 2019) for coordinating and preparing robot software for deployment. In this work the use of ROS is compulsory, in contrast with our approach which is robotic framework-agnostic.

In order to develop reusable components, that can be deployed on different robotic architectures, there is a need to shift designs from monolithc architectures to more modular microservice-like architecture. Our work establishes a proposed approach to encode such modularity by design.

## 2 Methodology

The aim of the proposed containerised approach is to define design principles to facilitate the construction of a robotic multiform modular architecture whose elements can be integrated in a preexisting robotic system. The approach is based on three fundamental principles derived from microservice architectures: componentisation, virtualisation and automation (Newman, 2015). Table 1 summarises its theoretical foundations. Our approach favours the reuse of preexisting code, but the approach can be used as guidelines for a new robotic framework as well. Developing a new framework from the beginning might sound ideal but it is not always possible or feasible, especially in industrial applications. In industry it might be more prudent to refactor existing code bases, proven to work in a real world production setting, and to incrementally evolve a robotic system to serve ever expanding requirements. With this assumption, rewriting a framework is costly and has high risks due to the fact that bugs can be introduced at any time and it takes time and effort to bring a new framework to feature parity with a previous code base. Additionally, an initial well thought design might prove to be limited by the time it is deployed to production as requirements might have changed in the meantime. As such, we conceived our approach for adapting existing code as this is a more frequent scenario than a full redesign.

Our approach relies on the creation of blueprints: virtual environments adapted for the execution of components. To use a virtualised component a container is generated from the blueprint, which acts as a template. We chose docker[2] for creating blueprints as it is an industry standard and it has tools for automatically deploying subsystems from the cloud (i.e. *docker-compose*). Additionally, several operations of our methodology (Table 1. *C*) are automated using a Continuous Integration (CI) tool, which is a tool designed for automating software development tasks. We chose Travis CI for this as it is a cloud-based CI, but others are also suitable.

According to our methodology, for the integration of a heterogeneous component it is required to follow the below workflow, summarised in Figure 2:

1. Define which operations the component will perform. Evaluate the opportunity of factoring out existing features integrated in a pre-existing robotic framework, if it is appropriate to do so at this stage (Table 1: *A.1*).

2.1. Define how the new component will communicate with the existing code base and other components (Table 1: *A.3*).

2.2. Create a blueprint for the heterogeneous component using docker, making sure that the component can be executed correctly (Table 1: *B.1*).

2.3. Add the chosen communication interfaces to the blueprint and test it in isolation. In our case, those were interface definitions in the ArmarX native communication library, but other approaches like e.g. ROS service handlers can be used for different systems. Other components and input required for testing can be included in the blueprint as a mock-up (Table 1: *A.2*).

3. Configure a CI, in our case Travis CI, to automatically prepare and test in isolation the blueprint at every commit (Table 1: *C.1*, *C.2*).

2 Docker Community Edition - https://www.docker.com/products/docker-desktop

TABLE 1 Founding principles of the proposed containerised approach for multiform robotic architectures.

| Principle | Explanation | Properties | Properties description |
|---|---|---|---|
| A. Componentisation | Dividing complex robot software into well defined software units | 1. Defined Functionalities | Identification of operations that a component provides to a client. The definition of a component is specific for the target application |
| | | 2. Tested in Isolation | Must always be possible to execute a component in isolation, using mock-up inputs/outputs if needed, to simplify testing |
| | | 3. Defined Interfaces | Clear means to interact with a component to execute its operations |
| B. Virtualisation | Creating a virtual environment where a component can operate | 1. Defined Working Environment | Construction of a virtual environment easy to execute where a component operates, ideally fully isolated from the host |
| | | 2. Cloud Hosting | Individual virtualised components, tested to be executable, must be accessible by users from the cloud |
| | | 3. Subsystem Preparation | Optionally, an ensemble of virtualised components can be automatically fetched from the cloud and assembled in a subsystem working out of the box |
| C. Automation | Automatically preparing, testing and sharing virtualised components on the cloud | 1. Automatic Build of Blueprints | Blueprints (templates) of the components and their virtual environment must be automatically prepared for every improvement |
| | | 2. Automatic Testing of Blueprints | New developments of a virtualised component shall be automatically tested to ensure that the component can at a minimum be executed with no errors |
| | | 3. Automatic Updates on the Cloud | Updated and tested blueprints of virtualised components should be automatically shared on the cloud to be retrieved by a third party |
| | | 4. Automatic Versioning | Blueprints and components' code must be automatically versioned and kept in sync, so that specific versions can be obtained predictably |

4. Ensure components are versioned appropriately by Travis CI (Table 1: *C.4*).
5. Once testing is completed, the blueprint is uploaded on the cloud (e.g. Google cloud in our case) and new versions are automatically uploaded by Travis CI (Table 1: *B.2*, *C.3*).
6. Optionally, a script can be prepared to automatically fetch and interconnect all the components of a subsystem of the robot (Table 1: *B.3*) using *docker-compose*. This facilitates the deployment of components closely related to each other.

Our methodology is an industry perspective on how to integrate multiform components to produce robotic systems ready for a production scenario. It builds up from principles well assessed in microservice development and applies them to robotics. Our contribution suggests an approach to make integration systematic and less error prone, which are key requirements in industry to ensure that robotic systems are reliable and robust from their first production release. We did find that poor integration can negatively impact the performance of novel research contributions to the point that they cannot be used in a production setting (Triantafyllou et al., 2021). Our approach is demonstrated on the real-world scenario of the SecondHands project[3]. The SecondHands robot (the ARMAR-6 (Asfour et al., 2018)) has a mobile base, multimodal sensory capabilities and the ability to physically and verbally interact with humans. Using its sensors, the robot has

3 SecondHands: A Robot Assistant For Industrial Maintenance Task, Website: https://secondhands.eu/

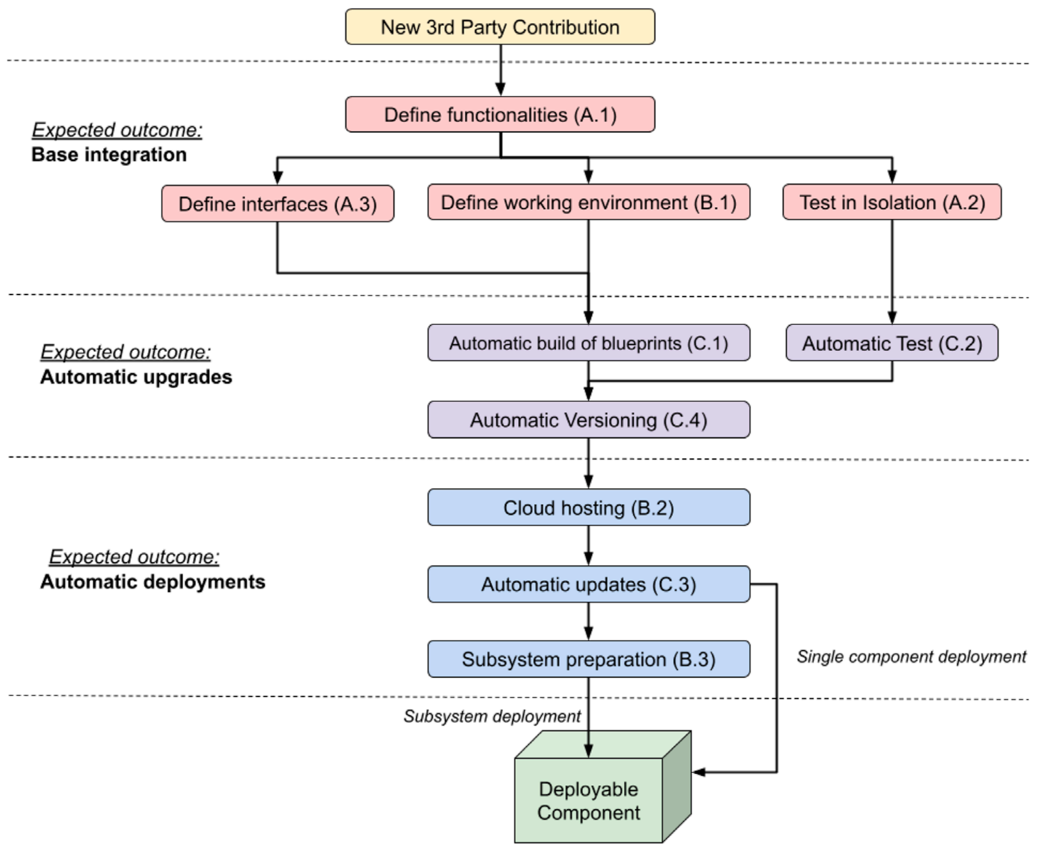


FIGURE 2
Proposed workflow of our methodology. Properties are grouped based on the expected outcome obtained by their application. Arrows indicate dependencies, e.g. cloud hosting should not be implemented if versioned blueprints are not available. Properties that are at the same level can be implemented concurrently. Please note that Principle B.3 is optional and its application is recommended if components are to be deployed together as a subsystem. The workflow can be applied to a new contribution or a pre-existent codebase, in which case A.1 can be used to decide features to factor out into an isolated component to be handled separately.

to predict and provide the help adequate to the situation. Our methodology can be applied to other architectures as it is not mandatory to refactor pre-existing code encapsulated into a robotic framework, such as ROS, if this is undesirable. Such code can be treated as a stand alone component, with well defined interfaces and responsibilities. Those responsibilities can be as broad as practically feasible to ensure the right balance between timely and incremental deliveries of new functionalities and overall architectural cohesion. Also, it is always possible to refactor any component, or to wrap them up with necessary boilerplate code, in order to fulfil the Properties of *Principle A*, if such a component does not adhere to them already. The Properties of *Principle B* can be applied to any working software as, to the best of the authors' knowledge, it is unlikely that a software cannot run in a virtual environment sharable over the cloud. The Properties of *Principle C* are best practices to follow to guarantee consistency, mostly enforced with robust CI pipelines. Also in this case, to the best of authors' knowledge, there is no reason to assume that CI pipelines cannot be crafted to fulfil the above Properties. Since our methodology does not require a mandatory rewrite of previous software, even if integrated in a robotic framework, and software can be reshaped to fulfil the Properties of our methodology, we believe that our approach can be generalised to other architectures.

The SecondHands architecture, shown in Figure 3, is augmenting the ArmarX robotics framework (Vahrenkamp et al., 2015) which provides several base functionalities and it is the framework used to operate the robot. The uniform structure of ArmarX cannot be altered as it is used for several other applications. All the other components of SecondHands have been developed by different research groups independently from ArmarX and represent the multiform part of the system. Such components have been integrated in a virtual environment and networked using ArmarX's native communication libraries. This was a design choice aimed at maximising compatibility with ArmarX. Components and their own virtual environments were made available from Google Cloud in the form of blueprints. Within SecondHands, two base blueprints were developed: one providing GPU support (for, e.g. neural networks) and another one more lightweight and without GPU support. For every change, blueprints were automatically built and tested in isolation by Travis CI, using mock-up components and inputs when required. A successfully built and tested blueprint was automatically uploaded on the cloud and versioned by Travis, following the four Properties of Principle C. The multiform part of the system communicates with ArmarX by requesting the execution of a state machine (Vahrenkamp et al., 2015) to perform a particular task, like handing over a tool. An ArmarX state machine relies on several

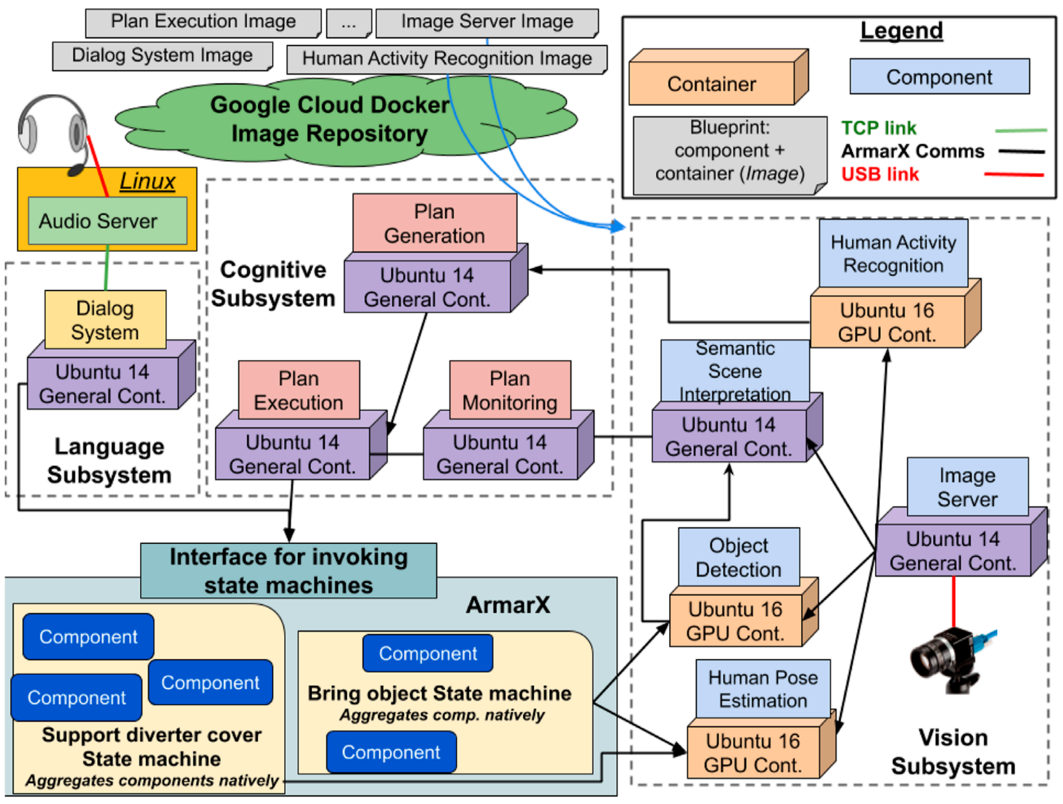


FIGURE 3
Overview of the SecondHands architecture. Blueprints (templates) of components and subsystems are available on Google Cloud. In the original architecture, a subsystem can be deployed on a dedicated PC. The multiform part of the system (cuboids and squares) only calls ArmarX's state machines using a dedicated interface. ArmarX can call the multiform components directly. All communication are handled using ArmarX native communication libraries.

ArmarX components and can request output from an heterogeneous component, such as the detection of the posture of the technician. This design choice allows to handle ArmarX itself as a component with Defined Interfaces and Functionalities (Table 1: *A.1*, *A.3*) whose internal changes do not affect the rest of the architecture. Similarly, internal changes to other heterogeneous components do not affect the rest of the system as components are isolated in their own containers. An exhaustive description of the functionalities of the SecondHands architecture is beyond the scope of this section, Section 3 describes its most important parts. As a result, our methodology extended ArmarX with several new components originally developed in different ways. The components are grouped in three subsystems: Cognitive, Language and Vision, which can be fetched and interconnected automatically using *docker-compose* scripts. The SecondHands robot is equipped with four on-board PCs with different hardware available. A detailed overview of the available hardware is given in Section 3.1. Figure 3 shows how every component is deployed on the robot for its usual operations, where every subsystem, including ArmarX itself, is deployed on a PC. Components are deployed on each machine leveraging *Properties B.3 and B.2* (Subsystem Preparation and Cloud Hosting), as every machine has a *docker-compose* configuration used to bring up the latest stable versions of every container, downloading them from the cloud if required. By using an appropriate versioning system (Principle C.4) it is possible to guarantee that only components tested and confirmed to work together will be brought up.

## 3 Evaluation

In this section, the SecondHands architecture, presented in Section 3.1 is used to evaluate the impact on resources overhead and integration complexity when all the principles of our methodology are followed and when they are not. The aim of the evaluation is to verify hypotheses 1 and 2, assessing if our methodology simplifies the complexity of integration and what is the added performance cost of using it. The evaluation criteria for the integration complexity are the number and type of modifications to a PC's configuration required to install a given component. The evaluation criterion for the resource overhead is the additional load placed by docker containers on the PC's resources (RAM, CPU, GPU and network traffic) when compared with the same system running natively.

### 3.1 Experimental setup

A scaled down version of the SecondHands architecture of Section 2, integrated using our approach (*Docker setup*), is compared

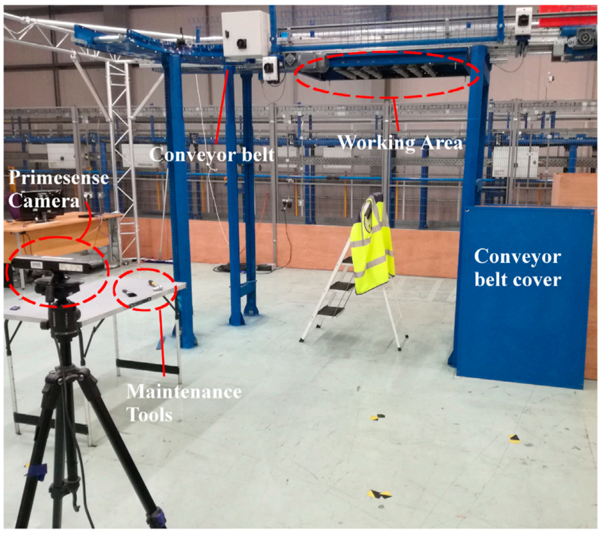


FIGURE 4
Experimental setup closely representing a real warehouse. A human operator had to wear a high visibility vest and perform maintenance in the working area using real tools.

to a *Native setup* of the same system which did not follow our approach. Both systems are expected to recognise the actions and speech of a technician working in a realistic environment and to control robot's hardware. The components of the evaluated system are: 1) the *Human Activity Recognition Component* (Alati et al., 2019) which uses a neural network running on a GPU to detect the technician's actions, 2) an *Image Server* which broadcasts images from a camera, 3) a *Dialogue System Component* (Constantin et al., 2018) which processes natural language, and 4) other components which form the infrastructure of the system (e.g. interfaces with ArmarX, communication among components, etc.).

The components were deployed on a replica of the computational infrastructure of the ARMAR-6 in Ocado Technology's Robotics Research lab. The infrastructure consists of four workstations, all Quad-core i7 Pentiums with 16 GB RAM, running Ubuntu 14, networked over Ethernet via a switch and connected to the ARMAR-6 hand, cameras and sound system. The four workstations are identified with a name describing their role: *1)* The *Vision PC* has an Nvidia GeForce GTX 1080 GPU to support vision processing and is directly connected to the robot's cameras (i.e. PrimenSense Carmine). *2)* The *Speech PC* is connected to microphones and a PreSonus AudioBox iTwo sound system to support natural language processing. *3)* The *Real-Time PC* has interfaces to the robot's hardware and controls an ARMAR-6 humanoid hand (Asfour et al., 2018). The hand is underactuated, has two degrees of freedom and can be operated only via ArmarX. It is operated to demonstrate that the integration of all the components is functional even for complex robotic hardware. *4)* The *Planning PC* is used for any other remaining functionality. In our setup, the communication was administrated by the *Real-Time PC* to further increase bandwidth consumption and stress the system.

## 3.2 Results: integration complexity

The integration complexity evaluation assesses how our approach influences the ease of integration of a multiform system in the worst case scenario. The complexity is measured in terms of additional configurations of the PC required to integrate and execute a component. To evaluate the integration complexity in its worst case scenario, the hard constraint of not altering the original component's code or container's blueprint was imposed. The evaluation is performed on a fully integrated system, so that interactions can be captured in a realistic setup. Additionally, for the Native setup, components were deployed on the least possible number of machines to maximise the interactions. Similarly, for the Docker setup, components were as isolated as possible from the host (the PC) as this setup is more complex. As such, it was attempted to deploy both setups only on the *Vision PC*, accessing the sound system on the *Speech PC* remotely, using Linux's audio server (Pulseaudio).

We classified sources of complexity in four main categories: 1) unset environment variables, 2) missing host configurations, 3) library incompatibilities, 4) driver incompatibilities. The first two categories are easy to address, requiring either to run a setup script or a persistent modification of the PC's configuration files (for, e.g. driver loading). However, they still require prior knowledge, i.e. a documented procedure. The last two categories are more complex to handle. Library incompatibilities appear when two components require conflicting versions of a library. If solvable, such issues require an *ad hoc* workaround. Driver incompatibilities are when the Linux kernel does not support a driver and a new Linux version needs to be installed on the PC.

When integrating a Native setup, it was observed that most components required a setup script to configure the working environment, while the Dialogue component required a customised host configuration. The most serious issue found was a GPU driver incompatibility with the Activity Recognition component that can only be solved by upgrading the operating system. As such the *Vision PC* was upgraded and the other components were deployed on the *Planning PC*. The camera was also relocated to the *Planning PC* since, to the best of our knowledge, it cannot be accessed remotely. The audio system on the *Speech PC* was still remotely accessed through Pulseaudio. The *Real-Time PC* was left unchanged. When integrating the Docker setup, no issues were experienced besides configuring the containers' networking. As such, to allow for a comparison with the Native setup, the Docker setup was deployed on the same machines. It is still possible to deploy the full system on a single machine if our methodology is fully followed. Hence, a third containerised setup (*All-in-one setup*), where all components co-exist on the *Vision PC*, was prepared. This setup was also evaluated in Section 3.3. The purpose of this additional evaluation is to give more comprehensive results for testing hypothesis 2, considering also the case where a single machine is bearing the load of the full system as there is no technical limitation preventing this to happen.

It can be observed that the Docker setup was easier to prepare as component's blueprints were ready to be used. Using blueprints, prepared as indicated in our approach, the setup is delegated to the original developers, who know their components more thoroughly than end-users and can pre-configure them easily.

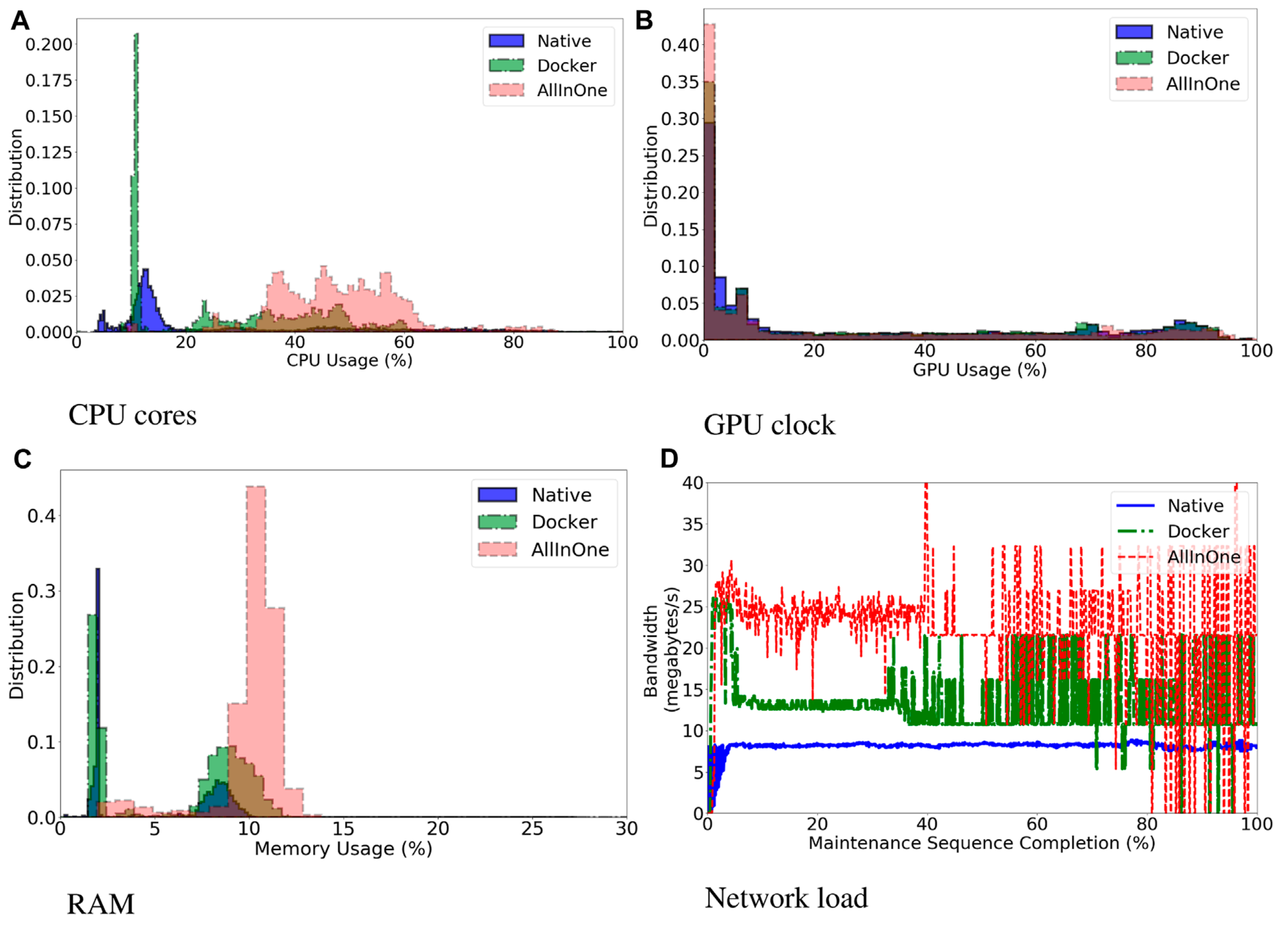


FIGURE 5
Overall histogram distribution of resource usage, discriminated by system setup, for CPU **(A)**, GPU **(B)** and RAM **(C)**. Plotting number of occurrences of a given load % during the steady-state of the task. Histograms more shifted to the right indicate a more loaded system. Figure **(D)** shows the network load in terms of mean bandwidth consumption over the % of completion of the steady-state of the task. Plotted as a time series.

## 3.3 Results: workload analysis

Workload analysis is performed to understand the computational costs of employing our approach. The three setups produced in Section 3.2, Docker, Native and All-in-one, processed a live simplified maintenance sequence performed in a close reproduction of a real warehouse, shown in Figure 4. The sequence required a human operator to dismantle a conveyor belt's protective cover causing the ARMAR-6 hand to close. Afterwards he would fetch and climb on a ladder to ask for a brush, clean the conveyor, extend the arm to give away the brush and give a verbal "stop" command, causing the ARMAR-6 hand to open. An example sequence can be viewed in the Supplementary Material. The sequence was performed 16 times by two people for a total of 48 experiments. Any trial which failed to operate the hand was repeated. The three setups were all assessed in terms of network bandwidth, memory, CPU and GPU workloads.

As described in Section 3.2, the Docker and Native setups were deployed on two PCs, while the All-in-one setup was deployed on the GPU-equipped *Vision PC*. For the Native setup, the CPU and RAM usages were monitored using the python system profiling library, *psutil*, while the Network bandwidth was monitored with the Linux's network traffic monitoring utility, *nethogs*. Only processes related to each component of the system were monitored via *psutil*. For the Docker and All-in-one setups, the same information was accessible through *docker stats*[4], docker's native monitoring tools, which profile CPU, RAM and network bandwidth and are the industry standard tools used to evaluate containers' key performance metrics. *docker stats* require the docker daemon to run. The daemon is also required to run containers. The computational cost of running the daemon is fixed and does not vary over time, as it is the case for the other components. As such, it was tracked in our results. The daemon is highly optimised to run with minimal footprint in production environments with several containers and the cost of running docker was evaluated to be an additional *0.25%* load on the RAM and less than *0.001%* load on the CPU. GPU workload and memory were measured via Nvidia logging tools for all setups. Only the steady-state part of

4 *Docker stats* documentation - https://docs.docker.com/engine/reference/commandline/container_stats/

TABLE 2 Quantitative characterisation of resource consumption. For each resource and setup is shown the mean value and the InterQuartile Range (IQR). Values are percentage of total resource used, for GPU and RAM, and absolute % of CPU cores used.

| | Native | | Docker | | All-in-one | |
|---|---|---|---|---|---|---|
| Resource | Mean | IQR | Mean | IQR | Mean | IQR |
| CPU | 25.72 | 27.26 | 29.90 | 32.85 | 48.15 | 16.51 |
| GPU | 28.50 | 57.00 | 27.86 | 56.98 | 24.33 | 48.81 |
| RAM | 5.30 | 6.52 | 6.16 | 7.51 | 10.07 | 1.13 |

a trial was analysed. This corresponds to the moment when the first image is broadcast until the hand opens fully. The start-up phase was not considered as the resource consumption is not stable initially and it is not representative of the true load of the system experienced when it is actively used or put under stress during normal operations. In a real world scenario, it is possible to define a process that prevents the system from being used while booting. Readings from individual components were collected at intervals of 250 ms each, relative clock drift measured among computers was negligible.

Figure 5 shows the distribution of resource consumption, expressed in % of total resources used for RAM and GPU, and absolute % of cores used for CPU. Each plot aggregates the data of all trials and components for each setup. Figures 5A, B, C are histograms showing the distribution of resource workload. The $X$ axis indicates different workloads while the $Y$ axis indicates number of occurrences. A distribution shifted to the right indicates an overall more loaded system. A spike indicates a workload that occurred more often than the others. Figure 5D is a time series, where on the $X$ axis is shown the percentage of completion of the maintenance sequence, the steady state of the system, and on the $Y$ axis the amount of bandwidth consumed. A summary of the data is shown in Table 2. The GPU memory usage is not reported as it is identical across the three setups. The overall load for each resource was calculated as usage difference relative to the Docker setup ($UD_d$). It was calculated for each resource as follows:

$$\begin{aligned} L[R_v] &= \frac{E[R_v]}{C} \\ UD_d[R] &= \frac{L[R_n] - L[R_d]}{L[R_d]} \end{aligned}$$

Where $v$ stands for the setup type (Docker $d$, or Native $n$), $E[R_v]$ is the mean usage value of a resource $R$ (CPU, RAM, GPU) for the setup $v$, $C$ is the maximum capacity of a resource, $L[R_v]$ is the total load percentage of a resource $R$ for setup $v$, and $UD_d[R]$ is the usage difference for resource $R$ relative to the Docker setup ($d$).

The resource usage difference ($UD_d$) is either small (**0.002*pp*** - CPU), slightly more (**+0.43*pp*** - RAM) or slightly less (**−0.60*pp*** - GPU clock). Additionally, as can be observed in Table 2, the differences of the InterQuartile Ranges between Docker and Native setups are 5.59*pp* (CPU), 0.99*pp* (RAM) and −0.02*pp* (GPU clock), which is also small. It can be concluded that the overhead of employing a containerised approach when the system is running at run-time is negligible as the Docker setup does not sensibly affect the overall system load.

The impact of deploying the whole system on one PC was also analysed. The All-in-one setup takes more time to fully load the GPU's memory ($13.20 \pm 0.63s$) when compared to the other two setups ($7.91 \pm 0.13s$ Native, $10.09 \pm 0.25s$ Docker). By observing the histograms of Figure 5, it can be seen that the All-in-one setup has a larger overhead since it is consistently using CPU and RAM more than the other two setups. This can be observed as the distributions on Figures 5A, C are shifted more to the right when compared to the other setups. The reason for such overhead can be seen in Figures 5D. The All-in-one setup produces a higher volume of information at higher speed than the other two setups suggesting that system is transferring more information and, as such, is loading its resources more.

## 4 Discussion and summary

### 4.1 Discussion

The results of Section 3.3 are used to test hypothesis 2: *Following all the guidelines of our methodology has a noticeable performance cost.* The Docker and All-in-one setup follow both all the three principles of the methodology, while the Native setup follows only *Principle A*, since it does use the same components' code as the other setups. The results demonstrate that the All-in-one setup have a higher workload, as shown by the histograms of Figure 5A higher production of information at a higher speed as seen in Figures 5D. *docker stats* monitors the information at container level, as such a higher bandwidth consumption indicates that the containers communicate more among each other. Those two results combined suggest that, qualitatively, the overall system is able to process more information at the cost of loading the host machine more. A possible explanation of this result is that data transits internally in the operative system via the docker daemon and can be delivered at a faster pace to the recipient.

Indeed, a similar, less pronounced, result can be observed on the Docker setup. In this case, there were more instances when the CPU was less loaded than in the All-in-one setup, as shown by the spike in Figure 5A, although the CPU was still active as its workload was spread between 20% and 60% for the whole steady-state of the trials. This likely happened because the data would have had to be packaged in a format suitable to be transmitted over the physical wire and then unpacked several times as individual packages were sent up over the network until reaching the recipient. This is one of the reasons why communication over a physical wire introduces a lag, which can be exacerbated by network traffic further loading the bandwidth forcing access points to limit the maximum amount of information that can be transferred at the same time. We do believe that in our case, packing and unpacking data over the network stack limited the amount of information travelling and reduced the overall load of every machine, which could process information at a slower pace.

In the All-in-one setup the host machine had to increase its workload to keep up with the information flow as it is able to process more data as soon as this is readily available. In the Docker setup, the host machines were not as loaded. This suggests that the machines could have processed more information if that was available. As such, the Docker setup might be more limited in the amount of information that could process at the same time as it will have to wait for the network stack packing and unpacking data packets before processing them. Another point to underline is that the technical specifications of the host machine for the All-in-one setup were sufficient to handle the full load of the system and to process information at a faster rate. This might not have been possible for a different robotic system, where the overall load could have been excessive for a single machine causing lack of responsiveness or even safety concerns.

This observation suggests there could be benefits in deploying more components on the same host machine as this could save time otherwise spent networking several hosts together. At the same time, overloading a machine could have side effects in the overall responsiveness of the system. Deciding how many hosts to use in a robotic system is a design decision that can be approached quantitatively by evaluating present and expected resource consumption and can keep financial and engineering costs lower. This decision has to be weighed by the fact that the control part of a robot is a real-time system, where timely responsiveness is one of the main factors to consider to ensure the overall safety of the robots and its users.

The results observed for the Docker setup were different from those observed for the Native setup, where the Network load was much lower despite both setups running the same code. This is confirmed by the fact that there were more instances when the CPU was less used than the Docker setup as its distribution in Figure 5A is more shifted to the left. We do believe that our results do not take into account issues with low level configurations of the network stack of the operative system which had an impact on the obtained results. This highlights the benefits of using a containerised approach such as the one presented in our work as docker optimised the network stack for efficient communication without the need to customise the operative system manually. As such, we can conclude that following all the principles of our methodology has a cost which can be offset by a better integration overall, which seems to falsify hypothesis 2 in our scenario.

Our results in Section 3.2 are used to test hypothesis 1: *Following all the guidelines of our methodology simplifies the integration complexity.* The results suggest that having components prepackaged into well defined software units (*Principle A* of our methodology) is a prerequisite but it is not the only aspect to consider. For all three setups the components were prepared according to *Principle A* and yet incompatibilities among components required *ad hoc* adjustments to have an integrated system. *Principle B*, having a virtual environment where the component can operate, was a key factor in facilitating the integration as this allowed to prepare the system in a systematic and streamlined way, identical for every component. *Principle C*, automatically preparing, testing and sharing virtualised components in the cloud, in our case, ensured consistency within components, further consolidating the integration approach in predefined and known steps. It could be that following just *Principles A and B* might be enough to obtain a painless integration. However it might be not sufficient to avoid *ad hoc* configurations and a streamlined integration process due to lack of consistency across components.

## 4.2 Summary

In this paper we proposed a containerised approach, inspired by microservice architectures, and its main principles aimed at augmenting existing robotic architectures with third party components. The principles can also be used as design guidelines for novel robotic frameworks. We applied our methodology to the SecondHands robot architecture and we evaluated our approach, both in terms of integration complexity and computational overhead, against the same architecture deployed without following our approach.

This study demonstrates that our approach grants more flexibility of integration as the same system can be deployed in different ways, even on the same PC, without substantially altering the configuration of the hosting PC. Additionally, we found that containers do not substantially impact the runtime performance of a system. Containerised setups are more reactive than native setups, and systems deployed on a single machine offer the highest reactivity at the cost of a larger workload. Our approach is relevant for robotics as it demonstrates how it is possible to augment an existing system with otherwise incompatible components, limiting the impact on existing code.

Moreover applications designed to run natively come with some form of configuration procedure. Although our approach aims at eliminating the need of any configuration other than deployment, it would be useful to test our approach on other architectures. Additionally, a further analysis of the latency of communication could provide information on the reactivity of native or containerised systems. For the best of our knowledge, it is not possible to evaluate the latency without modifying the original software and, as such, this evaluation will be performed in future work.

Finally, it is worth to note that the key to reusability lies on the quality of the original blueprints. This paper aims at providing guidelines to encourage usability and facilitate integration, however if the blueprints are not designed to be reusable, it is harder to intergate fully modular multiform robotic architectures.

## Data availability statement

The raw data supporting the conclusion of this article will be made available by the authors, without undue reservation.

## Ethics statement

Written informed consent was obtained from the individual(s) for the publication of any potentially identifiable images or data included in this article.

## Author contributions

GC: Conceptualization, Formal Analysis, Investigation, Methodology, Supervision, Visualization, Writing–original draft, Writing–review and editing. RR: Data curation, Formal Analysis, Software, Validation, Writing–original draft. GD: Funding acquisition, Project administration, Resources, Supervision, Writing–review and editing. JK: Funding acquisition, Project administration, Resources, Supervision, Writing–review and editing.

## Funding

The author(s) declare that financial support was received for the research, authorship, and/or publication of this article. This work is supported by the EU H2020 SecondHands project, research and Innovation programme (call: H2020-ICT-2014-1, RIA) under grant agreement No 643950.

## Conflict of interest

Authors GC, RR, GD, and JK were employed by Ocado Technology.

## Publisher's note

## Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/frobt.2024.1358978/full#supplementary-material

## References

Alati, E., Mauro, L., Ntouskos, V., and Pirri, F. (2019). "Anticipating next goal for robot plan prediction," in *Proceedings of SAI intelligent systems conference* (Springer), 792–809.

Asfour, T., Kaul, L., Wächter, M., Ottenhaus, S., Weiner, P., Rader, S., et al. (2018). "Armar-6: a collaborative humanoid robot for industrial environments," in *2018 IEEE-RAS 18th international conference on humanoid robots (humanoids)* (IEEE), 447–454.

Calisi, D., Censi, A., Iocchi, L., and Nardi, D. (2008). "Openrdk: a modular framework for robotic software development," in *2008 IEEE/RSJ international conference on intelligent robots and systems* (IEEE), 1872–1877.

Cervera, E. (2019). Try to start it! the challenge of reusing code in robotics research. *IEEE Robotics Automation Lett.* 4, 49–56. doi:10.1109/lra.2018.2878604

Constantin, S., Niehues, J., and Waibel, A. H. (2018). *Multi-task learning to improve natural language understanding. ArXiv* abs/1812.06876.

Cotugno, G., D'Alfonso, L., Muraca, P., and Pugliese, P. (2011). "A new extended kalman filter based on actual local information for mobile robots," in *9th European workshop on advanced control and diagnosis, ACD 2011*.

Cotugno, G., Turchi, D., Russell, D., and Deacon, G. (2020). "Secondhands: a collaborative maintenance robot for automated warehouses. implications for the industry and the workforce," in *Inclusive robotics for a better society* (Springer International Publishing), 195–200.

Fres, O. A., and Alonso, I. G. (2010). "Rovim: a generic and extensible virtual machine for mobile robots," in *2010 fifth international conference on systems* (IEEE), 37–40.

Hamberg, R., and Verriet, J. (2012). *Automation in warehouse development*. Springer.

He, K., Gkioxari, G., Dollar, P., and Girshick, R. (2017). "Mask r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2961–2969.

Hinze, C., Tasci, T., Lechler, A., and Verl, A. (2018). "Towards real-time capable simulations with a containerized simulation environment," in *2018 25th international conference on mechatronics and machine vision in practice (M2VIP)*, 1–6.

Khandelwal, P., Zhang, S., Sinapov, J., Leonetti, M., Thomason, J., Yang, F., et al. (2017). Bwibots: a platform for bridging the gap between ai and human–robot interaction research. *Int. J. Robotics Res.* 36, 635–659. doi:10.1177/0278364916688949

Liu, N., Liu, Z., Wei, Q., and Cui, L. (2018). "A containerized simulation platform for robot learning peg-in-hole task," in *2018 13th IEEE conference on industrial electronics and applications (ICIEA)*, 1290–1295.

Matamoros, M., Harbusch, K., and Paulus, D. (2018). "From commands to goal-based dialogs: a roadmap to achieve natural language interaction in robocup@ home," in *Robot world cup* (Springer), 217–229.

Metta, G., Fitzpatrick, P., and Natale, L. (2006). Yarp: yet another robot platform. *Int. J. Adv. Robotic Syst.* 3, 8. doi:10.5772/5761

Metta, G., Sandini, G., Vernon, D., Natale, L., and Nori, F. (2008). "The icub humanoid robot: an open platform for research in embodied cognition," in *Proceedings of the 8th workshop on performance metrics for intelligent systems (ACM)*, 50–56.

Mohamed, N., Al-Jaroodi, J., and Jawhar, I. (2008). "Middleware for robotics: a survey," in *Ram*, 736–742.

Mohanarajah, G., Hunziker, D., D'Andrea, R., and Waibel, M. (2014). Rapyuta: a cloud robotics platform. *IEEE Trans. Automation Sci. Eng.* 12, 481–493. doi:10.1109/tase.2014.2329556

Newman, S. (2015). *Building microservices: designing fine-grained systems*. Sebastopol, CA: O'Reilly Media, Inc.

Pot, E., Monceaux, J., Gelin, R., and Maisonnier, B. (2009). "Choregraphe: a graphical tool for humanoid robot programming," in *RO-MAN 2009 - the 18th IEEE international symposium on robot and human interactive communication*, 46–51.

Randazzo, M., Ruzzenenti, A., and Natale, L. (2018). Yarp-ros inter-operation in a 2d navigation task. *Front. Robotics AI* 5 (5), 5. doi:10.3389/frobt.2018.00005

Rodrigues, R. A. (2017). *An adaptive robotics middleware for a cloud-based bridgeOS*. Master's thesis, Istituto Tecnico Lisboa.

Seo, K.-T., Hwang, H.-S., Moon, I.-Y., Kwon, O.-Y., and Kim, B.-J. (2014). Performance comparison analysis of linux container and virtual machine for building cloud. *Adv. Sci. Technol. Lett.* 66, 2. doi:10.14257/astl.2014.66.25

Triantafyllou, P., Afonso Rodrigues, R., Chaikunsaeng, S., Almeida, D., Deacon, G., Konstantinova, J., et al. (2021). A methodology for approaching the integration of complex robotics systems: illustration through a bimanual manipulation case study. *IEEE Robotics Automation Mag.* 28, 88–100. doi:10.1109/MRA.2021.3064759

Turnbull, L., and Samanta, B. (2013). "Cloud robotics: formation control of a multi robot system utilizing cloud infrastructure," in *2013 proceedings of IEEE southeastcon* (IEEE), 1–4.

Vahrenkamp, N., Wächter, M., Kröhnert, M., Welke, K., and Asfour, T. (2015). The robot software framework armarx. *it-Information Technol.* 57, 99–111. doi:10.1515/itit-2014-1066

Wang, S., Liu, X., Zhao, J., and Christensen, H. I. (2019). "Rorg: service robot software management with linux containers," in *2019 international conference on robotics and automation (ICRA)*, 584–590. doi:10.1109/ICRA.2019.8793764

TYPE Systematic Review
PUBLISHED 10 July 2024
DOI 10.3389/frobt.2024.1377897

OPEN ACCESS

EDITED BY
Ivano Malavolta,
VU Amsterdam, Netherlands

REVIEWED BY
Nico Hochgeschwender,
University of Bremen, Germany
Giovanni De Gasperis,
University of L'Aquila, Italy

*CORRESPONDENCE
Esther Aguado,
e.aguado@upm.es

RECEIVED 28 January 2024
ACCEPTED 29 May 2024
PUBLISHED 10 July 2024

CITATION
Aguado E, Gomez V, Hernando M, Rossi C and Sanz R (2024), A survey of ontology-enabled processes for dependable robot autonomy. *Front. Robot. AI* 11:1377897.
doi: 10.3389/frobt.2024.1377897

# A survey of ontology-enabled processes for dependable robot autonomy

Esther Aguado[1,2]*, Virgilio Gomez[1,2], Miguel Hernando[2], Claudio Rossi[2] and Ricardo Sanz[1,2]

[1]Autonomous Systems Laboratory, Universidad Politécnica de Madrid, Madrid, Spain, [2]Centre for Automation and Robotics, Universidad Politécnica de Madrid-CSIC, Madrid, Spain

Autonomous robots are already present in a variety of domains performing complex tasks. Their deployment in open-ended environments offers endless possibilities. However, there are still risks due to unresolved issues in dependability and trust. Knowledge representation and reasoning provide tools for handling explicit information, endowing systems with a deeper understanding of the situations they face. This article explores the use of declarative knowledge for autonomous robots to represent and reason about their environment, their designs, and the complex missions they accomplish. This information can be exploited at runtime by the robots themselves to adapt their structure or re-plan their actions to finish their mission goals, even in the presence of unexpected events. The primary focus of this article is to provide an overview of popular and recent research that uses knowledge-based approaches to increase robot autonomy. Specifically, the ontologies surveyed are related to the selection and arrangement of actions, representing concepts such as autonomy, planning, or behavior. Additionally, they may be related to overcoming contingencies with concepts such as fault or adapt. A systematic exploration is carried out to analyze the use of ontologies in autonomous robots, with the objective of facilitating the development of complex missions. Special attention is dedicated to examining how ontologies are leveraged in real time to ensure the successful completion of missions while aligning with user and owner expectations. The motivation of this analysis is to examine the potential of knowledge-driven approaches as a means to improve flexibility, explainability, and efficacy in autonomous robotic systems.

KEYWORDS

**ontology, robot autonomy, adaptation, robustness, resilience**

## 1 Introduction

Autonomous robots have endless possibilities; they are applied in a variety of sectors such as transport, logistics, industry, agriculture, healthcare, education, energy, etc. Robotics has shown enormous potential in diverse tasks and environments. However, there are still open issues that compromise autonomous robot dependability. To support their deployment in real-world scenarios, we need to provide robots with tools to act and react properly in unstructured environments with high uncertainty.

Various strategies support the pursuit of a better grasp of intelligence: neuroscience tries to understand how the brain processes information; mathematics seeks computation and rules to draw valid conclusions using formal logic or handling uncertainty with probability and statistics; control theory and cybernetics aim to ensure that the system reaches desired goals, etc. (Russell and Norvig, 2021). One way that we explore in this survey is the use of symbolic explicit knowledge as a means to enhance system intelligence.

Knowledge representation and reasoning (KR&R) is a sub-area of artificial intelligence that concerns analyzing, designing, and implementing ways of representing information on computers so that computational agents can use this information to derive information implied by it (Shapiro, 2003). Reasoning is the process of extracting new information from the implications of existing knowledge. One of the challenges of getting robots to perform tasks in open environments is that programmers cannot fully predict the state of the world in advance. KR&R provides some background to reason about the runtime situation and act in consequence. In addition to adaptability, these approaches can provide an explanation; knowledge can be queried so humans or other agents can understand why a robot acts in a certain way. Lastly, KR&R provides reusability. The robot needs information about its capabilities and the environment in which it is involved; this information can be shared among different agents, applications, or tasks because knowledge bases can be stored in broadly applicable modular chunks.

To be useable by robots, knowledge bases must be machine-understandable because the robot shall read, reason about, and update its content. In the context of intelligent robots, *ontologies* allow us to define the conceptualizations that the robot requires to support autonomous decision making. These ontologies are written in specific computer languages. In this article, we present a review of the ontologies used by autonomous robots to perform complex missions. In our analysis, we focus on the critical aspects of robot autonomy, that is, how KR&R can contribute to increasing the dependability of these types of systems. We aim to draw a landscape on how ontologies can support decision making to build more dependable robots performing elaborated tasks. The main contributions of this work are:

1. A classification of ontologies based on concepts for autonomous robots,
2. An analysis of existing approaches that use ontologies to facilitate action selection, and
3. A discussion of future research directions to increase dependability in autonomous robots.

This article is structured as follows. Section 2 provides background information on KR&R with a special focus on ontologies for robot autonomy. Section 3 discusses the classification criteria for ontologies that support robust autonomy in robotic applications. In Section 4, we describe the selection process for a systematic review in the field of ontologies for autonomous robots and briefly discuss other surveys in the field. In Section 5, we classify and compare the selected approaches based on their main application domain. Section 6 provides a general discussion and describes future research directions. Lastly, Section 7 presents the article's conclusions.

## 2 Knowledge representation and reasoning for robot autonomy

In this section, we introduce the use of ontologies to increase dependability in autonomous systems and the main languages for knowledge-based agents. Finally, we also discuss the essential and desirable capabilities of fully autonomous robots.

Dependability is the ability to provide the intended services of the system with a certain level of assurance, including factors such as availability, reliability, safety, and security (SEBoK Editorial Board, 2023). As an introductory exploration into this field, Guiochet et al. (2017) provide an analysis of techniques used to increase safety in robots and autonomous systems. Avižienis et al. (2004) offer definitions and conceptualizations of aspects related to dependability and create a taxonomy of dependable computing and its associated faults as a framework.

Dependability ensures consistent performance that goes beyond reliability, as it also considers factors such as fault tolerance, error recovery, and maintaining service levels. Various models of tackling these challenges include fault trees, failure modes and effect analysis (FMEAs), and safety case modes. There are also model-based approaches in which fault analysis can be used in runtime to evaluate goals and develop and execute alternative plans (Abbott, 1990). Similarly, this survey focuses on using explicit models to select the most suitable actions for the situation the robot faces to provide the desired outcomes. This survey concentrates on leveraging ontologies as a means to enhance dependability rather than analyzing dependability aspects concerning robotics.

### 2.1 Languages for ontologies

The most extended approach to robot programming is procedural: The intended robot behavior is explicitly encoded in an imperative language. Knowledge-based agents depart from this, using declarative approaches to abstract the control flow. A knowledge base (KB) stores information that allows the robot to deduce how to operate in the environment. This information can be updated with new facts, and repeated queries are used to deduce new facts relevant to the mission. A successful agent often combines *declarative knowledge* with procedural programming to produce more efficient code (Russell and Norvig, 2021). This declarative knowledge is commonly encoded in logic systems to formally capture conceptualizations.

Prolog (PROgramming in LOGic) is the most widely used declarative programming language. Many knowledge-based systems have been written in this language for legal, medical, financial, and other domains (Russell and Norvig, 2021). Prolog defines a formalism based on decidable fragments of first-order logic (FOL). In addition to its notation, which is somewhat different from standard FOL, the main difference is the closed-world assumption. The closed-world assumption asserts that only the predicates explicitly defined in the KB are true; there is no way to declare that a sentence is false.

Another widely used language for writing ontologies is Ontology Web Language (OWL). OWL is not a programming language like Prolog but rather a family of languages that can be used to represent knowledge. It is designed for applications that need to process information rather than simply presenting information to humans. Therefore, OWL facilitates greater machine interpretability with a panoply of different formats, such as extensible markup language (XML), resource description framework (RDF), and RDF Schema (RDF-S). The language has been specified by the W3C OWL Working Group (2012) and is a cornerstone of the semantic web.

The formal basis of OWL is description logic (DL): a family of languages with a compromise between expressiveness and scalability. DL uses decidable fragments of FOL to reason with more expressiveness. The main difference between Prolog and OWL is that the latter uses the open-world assumption. This means that a statement can be true whether it is known or not; that is, only explicitly false predicates are false, in contrast to the closed-world assumption in which only explicitly true predicates are true.

Regarding accessibility, OWL can be encoded by hand or by editors such as Protégé (Musen, 2015) for a user-friendly environment. OWL can be integrated into robotic software through application program interfaces (APIs) such as Jena Ontology API[1] or OWLAPI[2] implemented in Java or OWLREADY (Lamy, 2017) implemented in Python. There are also reasoners such as FaCT++[3], Pellet[4]>, or HermiT[5].

Prolog can also be used to reason about knowledge represented by OWL or to directly encode a Prolog ontology. The main Prolog APIs are GNU-PROLOG[6] and SWI-PROLOG[7]. There is even a package for the robot operating system (ROS) called *rosprolog*[8] that interfaces between SWI-Prolog and ROS.

## 2.2 Fundamental and domain ontologies

Ontological systems can be classified according to several criteria. We distinguish three levels of abstraction based on Guarino's hierarchy (Guarino, 1998): upper level, domain, and application. *Upper-level* or foundational ontologies conceptualize general terms such as object, property, event, state, and relations such as parthood, constitution, participation, etc. *Domain* ontologies provide a formal representation of a specific field that defines contractual agreements on the meaning of terms within a discipline (Hepp et al., 2006); these ontologies specify the highly reusable vocabulary of an area and the concepts, objects, activities, and theories that govern it. *Application* ontologies contain the definitions required to model knowledge for a particular application: information about a robot in a specific environment, describing a particular task. Note that the environment or task knowledge could be a subdomain ontology, depending on the reusability it allows. In fact, the progress from upper-level to application ontologies is a continuous spectrum of concept subclassing, with somewhat arbitrary divisions into levels of abstraction reified as ontologies.

Perhaps the most extended upper-level ontology is the Suggested Upper Merged Ontology (SUMO) (Niles and Pease, 2001; Pease, 2011)[9]. SUMO is the largest open-source ontology that has expressive formal definitions of its concepts. Domain ontologies for medicine, economics, engineering, and many other topics are part of SUMO. This formalism uses the Standard Upper Ontology Knowledge Interchange Format (SUO-KIF), a logical language to express concepts with higher-order logic (a logic with more expressiveness than first-order logic) (Brown et al., 2023).

Another relevant foundational ontology for this research is the Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE), described as an "ontology of universals" (Gangemi et al., 2002), which means that it has classes but not relations. It aims to capture the ontological categories that underlie natural language and human common sense (Mascardi et al., 2006). The taxonomy of the most basic categories of particulars assumed in DOLCE includes, for example, abstract quality, abstract region, agentive physical object, amount of matter, temporal quality, etc. Although the original version of the few dozen terms in DOLCE was defined in FOL, it has since been implemented in OWL; most extensions of DOLCE are also in OWL.

The DOLCE + DnS Ultralite ontology[10] (DUL) simplifies some parts of the DOLCE library, such as the names of classes and relations with simpler constructs. The most relevant aspect of DUL is perhaps the design of the ontology architecture based on patterns.

Other important foundational ontologies are the Basic Formal Ontology (BFO) (Arp et al., 2015), the Bunge–Wand–Weber Ontology (BWW) (Bunge, 1977; Wand and Weber, 1993), and the Cyc Ontology (Lenat, 1995). BFO focuses on continuant entities involved in a three-dimensional reality and occurring entities, which also include the time dimension. BWW is an ontology based on Bunge's philosophical system that is widely used for conceptual modeling (Lukyanenko et al., 2021). Cyc is a long-term project in artificial intelligence that aims to use an ontology to understand how the world works by trying to represent implicit knowledge and perform human-like reasoning.

### 2.2.1 Robotic domain ontologies

Ontologies have gained popularity in robotics with the growing complexity of actions that systems are expected to perform. A well-defined standard for knowledge representation is recognized as a tool to facilitate human–robot collaboration in challenging tasks (Fiorini et al., 2017). The IEEE Standard Association of Robotics and Automation Society (RAS) created the Ontologies for Robotics and Automation (ORA) working group to address this need.

They first published the Core Ontology for Robotics and Automation (CORA) (Prestes et al., 2013). This standard specifies the most general concepts, relations, and axioms for the robotics and automation domain. CORA is based on SUMO and defines what a robot is and how it relates to other concepts. For this, it defines four main entities: robot part, robot, complex robot, and robotic system.

1 https://jena.apache.org/documentation/ontology
2 https://github.com/owlcs/owlapi/wiki
3 http://owl.cs.manchester.ac.uk/tools/fact
4 https://github.com/stardog-union/pellet
5 http://www.hermit-reasoner.com/
6 http://www.gprolog.org/
7 https://www.swi-prolog.org/
8 https://github.com/KnowRob/rosprolog
9 https://www.ontologyportal.org, https://github.com/ontologyportal
10 http://ontologydesignpatterns.org/wiki/Ontology:DOLCE+DnS_Ultralite

CORA is an upper-level ontology currently extended in the IEEE Standard 1872-2015 (IEEE SA, 2015) with other subontologies, such as CORAX, RPARTS, and POS.

CORAX is a subontology created to bridge the gap between SUMO and CORA. It included high-level concepts that the authors claimed to not be explicitly defined in SUMO and particularized in CORA, in particular those associated with design, interaction, and environment. RPARTS provides notions related to specific kinds of robot parts and the roles they can perform, such as grippers, sensors, or actuators. POS presents general concepts associated with spatial knowledge, such as position and orientation, represented as points, regions, and coordinate systems.

However, CORA and its extensions are intended to cover a broad community, so their definitions of ambiguous terms are based solely on necessary conditions and do not specify sufficient conditions (Fiorini et al., 2017). For this reason, concepts in CORA must be specialized according to the needs of specific subdomains or robotics applications.

CORA, like most of the other application ontologies considered here, is defined in a language of very limited expressiveness, mostly expressible in OWL-Lite, and is therefore limited to simple classification queries. Although it is based on upper-level terms from SUMO, it recreated many terms that could have been used directly from SUMO. Moreover, given its choice of representation language, it did not use the first- and higher-order logic formulas from SUMO, limiting its reuse to only the taxonomy.

The IEEE ORA group created the Robot Task Representation subgroup to produce a *middle-level ontology* with a comprehensive decomposition of tasks, from goal to subgoals, that enables humans or robots to accomplish their expected outcomes at a specific instance in time. It includes a definition of tasks and their properties and terms related to the performance capabilities required to perform them. Moreover, it covers a catalog of tasks demanded by the community, especially in industrial processes (Balakirsky et al., 2017).

This working group also created three additional subgroups for more specific domain knowledge: Autonomous Robots Ontology, Industrial Ontology, and Medical Robot Ontology. Only the first of these has been active. The Autonomous Robot subgroup (AUR) extends CORA and its associated ontologies for the domain of autonomous robots, including, but not limited to, aerial, ground, surface, underwater, and space robots.

They developed the IEEE Standard for Autonomous Robotics Ontology (IEEE SA, 2022) with an unambiguous identification of the basic hardware and software components necessary to provide a robot or a group of robots with autonomy. It was conceived to serve different purposes, such as to describe the design patterns of Autonomous Robotics (AuR) systems, to represent AuR system architectures in a unified way, or as a guideline to build autonomous systems consisting of robots operating in various environments.

In addition to the developments of the IEEE ORA working group, there are other relevant domain ontologies, such as OASys and the Socio-physical Model of Activities (SOMA). The Ontology for Autonomous Systems (OASys) (Bermejo-Alonso et al., 2010) captures and exploits concepts to support the description of any autonomous system with an emphasis on the associated engineering processes. It provides two levels of abstraction systems in general and autonomous systems in particular. This ontology connects concepts such as architecture, components, goals, and functions with the engineering processes required to achieve them.

The SOMA for Autonomous Robotic Agents represents the physical and social context of everyday activities to facilitate accomplishing tasks that are trivial for humans (Beßler et al., 2021). It is based on DUL, extending their concept to different event types such as action, process, and state, the objects that participated in the activities, and the execution concept. It is worth mentioning that SOMA was intended to be used in the runtime along with the concept of narratively enabled episodic memories (NEEMs), which are comprehensive logs of raw sensor data, actuator control histories, and perception events, all semantically annotated with information about what the robot is doing and why using the terminology provided by SOMA.

The relationship between upper-level ontologies and domain ontologies is a relationship of progressive domain focalization (Sanz et al., 1999). The frameworks described in the following sections are mostly specializations of these general robotic ontologies and other foundations.

## 2.3 Capabilities for robot autonomy

Etymologically, autonomy means being governed by the laws of oneself rather than by the rules of others (Vernon, 2014). Beer et al. (2014) provide a definition more closely related to robotics: autonomy as the extent to which a robot can *sense* its environment, *plan* based on that environment, and *act* on that environment with the intention of reaching some task-specific *goal* (either given or created by the robot) without external control. A related, systems-oriented perspective pursued in our lab considers autonomy as a relationship between system, task, and context (Sanz et al., 2000).

Rational agents use sense-decide-act loops to select the best possible action. According to Vernon (2014), cognition allows one to increase the repertoire of actions and extend the time horizon of one's ability to anticipate possible outcomes. He also reviews several cognitive architectures to support artificial cognitive systems and discusses the relationship between cognition and autonomy. For him, cognition includes six attributes: perception, learning, anticipation, action, adaptation, and, of course, autonomy.

Following the link between cognition and autonomy, Langley et al. (2009) also review cognitive architectures and establish the main functional capabilities that autonomous robots must demonstrate. Knowledge is described as an internal property to achieve the following capabilities: (i) recognition and categorization to generate abstractions from perceptions and past actions, (ii) decision making and choice to represent alternatives for selecting the most prosperous action considering the situation, (iii) perception and situation assessment to combine perceptual information from different sources and provide an understanding of the current circumstances, (iv) prediction and monitoring to evaluate the situation and the possible effects of actions, (v) problem solving and planning to specify desired intermediate states and the actions required to reach them, (vi) reasoning and belief maintenance to use and update the KB in dynamic environments, (vii) execution and action to support deliberative and reactive behaviors, (viii) interaction and communication to share knowledge with other agents, and (ix) remembering, reflection, and learning to use meta-reasoning to use past executions as experiences for the future.

Brachman (2002) argues about the promising capabilities of reflective agents. For him, real improvements in computational agents come when systems know what they are doing, that is, when the agent can understand the situation: what it is doing, where, and why. Brachman establishes practically the same foundations as Langley et al. (2009) but explicitly mentions the necessity of coordinated teams and robust software and hardware infrastructure.

In conclusion, most authors recall the importance of the features described above with different levels of granularity. In the next section, we explain and conceptualize those functional capabilities that enable robot operation autonomously. Note that the systems under study use explicit knowledge—ontologies—as the backbone to achieve autonomy.

## 3 Processes for knowledge-enabled autonomous robots

In this section, we introduce the classification on which we will base the review of ontologies for dependable robot autonomy in Section 5. The classification criterion is based on the capabilities introduced in Section 2.3. It establishes the fundamental processes that an autonomous robot should perform.

### 3.1 Perception

A percept is the belief produced as a result of a perceptor sensing the environment in an instant. Perception involves five entities: sensor, perceived quality, perceptive environment, perceptor, and the percept itself.

A *sensor* is a device that detects, measures, or captures a property in the environment. Sensors can measure one particular aspect of the physical world, such as thermometers, or capture complex characteristics, such as segmenting cameras.

A *perceived quality* is a feature that allows the perceptor to recognize some part of the environment—or the robot itself. Note that this quality is mapped into the percept as an instantiation, a belief produced to translate physical information to the system model. Examples of perceived qualities are the temperature, visual images of the environment, and the rotation of a wheel measured by a rotative encoder placed in a robot.

A *perceptive environment* is the part of the environment that the sensor can detect. The perceptive region can be delimited by the sensor's resolution or to save memory or other resources.

An agent that perceives is a *perceptor*. It constitutes the link between perception and categorization because it takes sensor information and categorizes it. Usually, the perceptor embodies the sensor, as is the case in autonomous robots, but the two could be decoupled if the system processes information from external sensors. Finally, the *percept* is the inner entity—a belief—that results from the perceptual process.

### 3.2 Categorization

Percepts are instantaneous approximate representations of a particular aspect of the physical world. To provide the autonomous robot with an understanding of the situation in which it is deployed, the perceptor must abstract the sensor information and recognize objects, events, and experiences.

Categorization is the process of finding patterns and categories to model the situation in the robot's knowledge. It can be done at different levels of granularity. Examples of categorizations appear everywhere: at sensor fusion processes from different types of sensors, with their corresponding uncertainty propagation; at the classification of an entity as a mobile obstacle when it is an uncontrolled object approaching the robot; or, in a more abstract level, when a mobile miner robot recognizes the type of mine ore depending on its geo-chemical properties.

In general, this step corresponds to a combination of information about objects, events, action responses, physical properties, etc., to create a picture of what is happening in the environment and in the robot itself. For this purpose, the robot shall incorporate other processes, such as reasoning and prediction.

### 3.3 Decision making

An autonomous robot must direct its actions towards a goal. When an action cannot be performed, the robot shall implement mechanisms to make decisions and select among the most suitable alternatives for the runtime situation.

Note the difference between decision making and planning. Decision making has a shorter time frame because it focuses on the successful completion of the plan. An example of a decision at this level could be to slightly change the trajectory of a robot to avoid an obstacle and then return to the initial path. Planning, on the other hand, is concerned with achieving a goal; it has a longer time horizon to establish the action sequences to complete a mission. For example, the miner robot presented above must examine the mine, detect the mineral vein, and dig in that direction.

Decision making acts upon the different alternatives that the robot can select, for example, in terms of directions and velocity, but also in terms of what component with functional equivalence can substitute for a defective one. The execution of the action—how it is done—changes slightly, but the plan—the action sequence that achieves the goal—remains the same.

### 3.4 Prediction and monitoring

Once the robot has an internal model, it can supervise the situation. This model can be given *a priori* or created before stating the task; it could also be learned. The model reflects the understanding of the robot about its own characteristics, its interactions with the environment, and the relationships between its actions and its outcomes. At runtime, the robot can use the model to predict the effect of an action. It can also anticipate future events based on the way the situation is evolving.

This mechanism also allows the robot to monitor processes and compare the result obtained with the expected response. In the event of inconsistencies, it can inform an external operator or use adaptation techniques to solve possible errors. For example, if a robot is stuck, it may change its motion direction to get out, and if this is not possible, it can alert the user.

The prediction process can also benefit from a learning procedure to improve the models from experience and refine them over time. Furthermore, monitoring can be used during learning, as it detects errors that can help improve the model.

## 3.5 Reasoning

Reasoning is the process of using existing knowledge to draw logical conclusions and extend the scope of that knowledge. It requires solid definitions and relationships between concepts. It uses instances of such concepts to ground them to the robot's operation and to be used during the mission.

The reasoning process can be used to infer events based on the current percepts, such as the dynamic object approaching the robot, or to infer the best possible action to overcome it based on its background knowledge, such as reducing speed and slightly changing the direction. Lastly, as robots operate in dynamic worlds—specifically, they operate by changing their environment—the knowledge shall evolve over time.

Reasoning includes two processes: (i) infer and maintain beliefs and (ii) discard beliefs that are no longer valid. Logical systems support assertions and retractions for this purpose; however, they must be handled carefully to maintain ontological consistency. Truth maintenance is a critical capability for cognitive agents situated in dynamic environments.

## 3.6 Planning

Planning is the process of finding a sequence of actions to achieve a goal. To reach a solution, the problem must be structured and well defined, especially in terms of the starting state, which the robot shall transform into a desired goal state. The system also needs to know the constraints to execute an action and its expected outcome, that is, preconditions and postconditions. These conditions are also used to establish the order between actions and the effect that they may have on subsequent actions.

The required information is usually stored in three types of models: environment, robot, and goal models. Most authors only mention the environmental model, which includes the most relevant information about the robot world and its actions, tasks, and goals; however, we prefer to isolate the three models to make explicit the importance of proprioceptive information and performance indicators for a more dependable autonomous robot.

Plans can completely guide the behavior of the robot or suggest a succession of abstract actions that can be expanded in different ways. This can result in branches of possible actions, depending on the result of previous states.

Planning is also closely related to monitoring; the supervision output can conclude the effectiveness of the plan or detect some unreachable planned actions. In this case, the plan may need adjustments, such as changing parameters or replacing some actions. Replanning can use part of the plan or draw a completely new structure depending on the progress and status of the plan and the available components.

Lastly, successful plans or sub-plans can be stored for reuse. These stored plans can also benefit from learning, especially with regard to the environmental response to changes and action constraints and outputs.

## 3.7 Execution

A key process in robotic deployments is the execution of actions that interact with the environment. The robot model must represent the motor skills that produce such changes. Execution can be purely deliberative or combined with more reactive approaches; for example, a patrolling robot may reduce its speed or stop to ensure safety when close to a human—reactiveness—but it also needs a defined set of waypoints to fully cover an area—deliberation.

Hence, an autonomous robot must facilitate the integration of both reactive and deliberative actions within a goal-oriented hierarchy. A strictly reactive approach would limit the ability to direct the robot's actions toward a defined objective. Meanwhile, an exclusively deliberative approach might be excessively computationally intensive and lead to delayed responses to instantaneous changes.

Another aspect of execution is control. Robots use controllers to overcome small deviations from their state. These controllers can operate in open-loop or closed-loop mode. Open-loop controllers apply predetermined actions based on a set of inputs, assuming that the system will respond predictably. Although they lack the ability to correct runtime deviations, they are often simpler and faster. Closed-loop controllers provide a more accurate and precise action based on inputs and feedback received. Control grounds the decision-making process by specifying the final target value for the robot effectors. It constitutes the final phase of action execution.

## 3.8 Communication and coordination

In many applications, robots operate with other agents—humans or robots of a different nature. Communication is a key feature to organize actions and coordinate them towards a shared goal. Moreover, in knowledge-based systems, communication provides an effective way to obtain knowledge from other agents' perspectives.

Shared information provides a means to validate perceived elements, fuse them with other sources, and provide access to unperceivable regions of the world. However, this requires a way to exchange information between agents in a neutral, shared conceptualization that is understandable and useable for both.

Once communication is established, we shall coordinate the actions of the systems involved. Decision-making and planning processes should take into account the capabilities and availability of agents to direct and sequence their actions toward the most promising solution. For example, in multirobot patrols, agents shall share their pose and planned path to avoid collisions. Another example could be exploring a difficult-to-access mine in which a wheeled robot could be used for most of the inspection activities, and a legged robot could be used to inspect the unreachable areas.

## 3.9 Interaction and design

Interaction and design are often omitted when analyzing autonomous capabilities. Although they are part of the design phase, this engineering knowledge holds considerable influence over the robot's performance and dependability.

Interaction between agents can be handled through coordination; however, the embodiment of robots can produce interactions between software and/or hardware components. Robots should be aware of the interaction ports and the possible errors that arise from them. This concern is presented by Brachman (2002), as awareness of interaction allows the robot to step back from action execution and understand the sources of failure. This becomes particularly significant during the integration of diverse components and subsystems, where the application of systems engineering techniques proves to be highly beneficial.

Hernández et al. (2018) argue about the need to exploit functional models to make explicit design decisions and alternatives at runtime. These models can provide background knowledge about requirements, constraints, and assumptions under which a design is valid. With this knowledge, we can endow robots with more tools for adaptability, providing the capability to overcome deviations or contingencies that may occur. For example, a manipulator robot with several tools may have one optimal tool for a task, but if this component is damaged, it can use an alternative tool to solve the problem in a less-than-optimal way.

## 3.10 Learning

Most of the processes described above can improve their efficacy through learning: categorization, decision making, prediction and monitoring, planning, execution, coordination, design, etc. Learning can be divided into three steps: remember, reflect, and generalize.

- *Remember* is the ability to store information from previous executions.
- *Reflect* involves analyzing remembered information to detect patterns and establish relationships.
- *Generalize* is the process of abstract conclusions derived from reflection and subsequently extended to use them in future experiences.

The classification of these competencies reveals their incorporation into KBs, identifies potential underrepresented elements, and explores the contributions of knowledge structures to the decision-making process.

## 4 Review process

One of our main objectives in this article is to conduct a systematic analysis of recent and relevant projects that use ontologies for autonomous robots. The methodology followed is inspired by relevant surveys discussed in the next section, such as those by Olivares-Alarcos et al. (2019) and Cornejo-Lupa et al. (2020). To avoid personal bias, the entire article selection process has been cross-analyzed by two people, and the framework analysis has been validated by the five authors.

The first step in our review process was to search for relevant keywords in scientific databases. Specifically, we used the most extended literature browsers, *Scopus*[11] and *Web of Science*[12]. These databases provide a wide range of peer-reviewed literature, including scientific journals, books, and conference proceedings. Scopus includes more than 7,000 publishers, and WOS includes more than 34,000 journals, including important journals in the field, such as those from IEEE, Springer, and ACM. It also has a user-friendly interface to store, analyze, and display articles. Moreover, related articles cited in the analyzed articles are included in the review process to ensure that relevant articles were not missed. The search was done in terms of title, abstract, or keywords containing the terms robot, ontology, plan, behavior, adapt, autonomy, or fault. In practice, we used the following search string. The search can be replicated using the provided query. However, it is important to note that the survey was conducted in 2022, and there may have been developments or new publications since then. Additionally, a list of the analyzed works and intermediate documents is available upon request.

The required terms are "ontology" and "robot," as they are the foundation of our survey. Using this restriction may seem somewhat limiting because there could be knowledge-based approaches to cognitive robots that are not based on ontologies. However, in this review, we are specifically interested in using explicit ontologies for this purpose, hence the strong requirement regarding "ontology." We also target at least one keyword related to (i) selection and arrangement of actions—autonomous, planning, behavior—or (ii) overcoming contingencies—fault, adapt. Note that we use asterisks to be flexible with the notation.

This search returned 695 articles after removing duplicates. To mitigate potential biases, we implemented specific inclusion criteria based on publication dates and citation counts. We only include articles widely cited, with 20 or more citations, before 2010; articles between 2010 and 2015 with five or more citations; and articles between 2015 and 2018 with two or more citations. All articles from 2018 onward were included. With these criteria, we reduced the list to 351 articles. The selection process applied these criteria to identify articles that are not only recent but have also demonstrated impact and influence within their respective publication periods.

Then, we analyzed the content of the articles. We included works in which the main point of the article is the ontology. In particular, the work (i) proposes or extends the ontology and (ii) uses the ontology to select, adapt, or plan actions. We eliminated a number of articles on how to use ontologies to encode simulations, on-line generators of ontologies, or ontologies only used for conversation, perception, or collaboration without impact on robot action. The application of these criteria produced 26 relevant articles.

Finally, we complemented our search with articles from related surveys described in Section 4.1. This ensured that we included all relevant and historical articles in the field with a snowball process. In this step, we obtained 22 more articles; some of them were already included in the previous list, and others did not meet our

11 https://www.scopus.com

12 https://www.webofscience.com/wos/alldb/basic-search

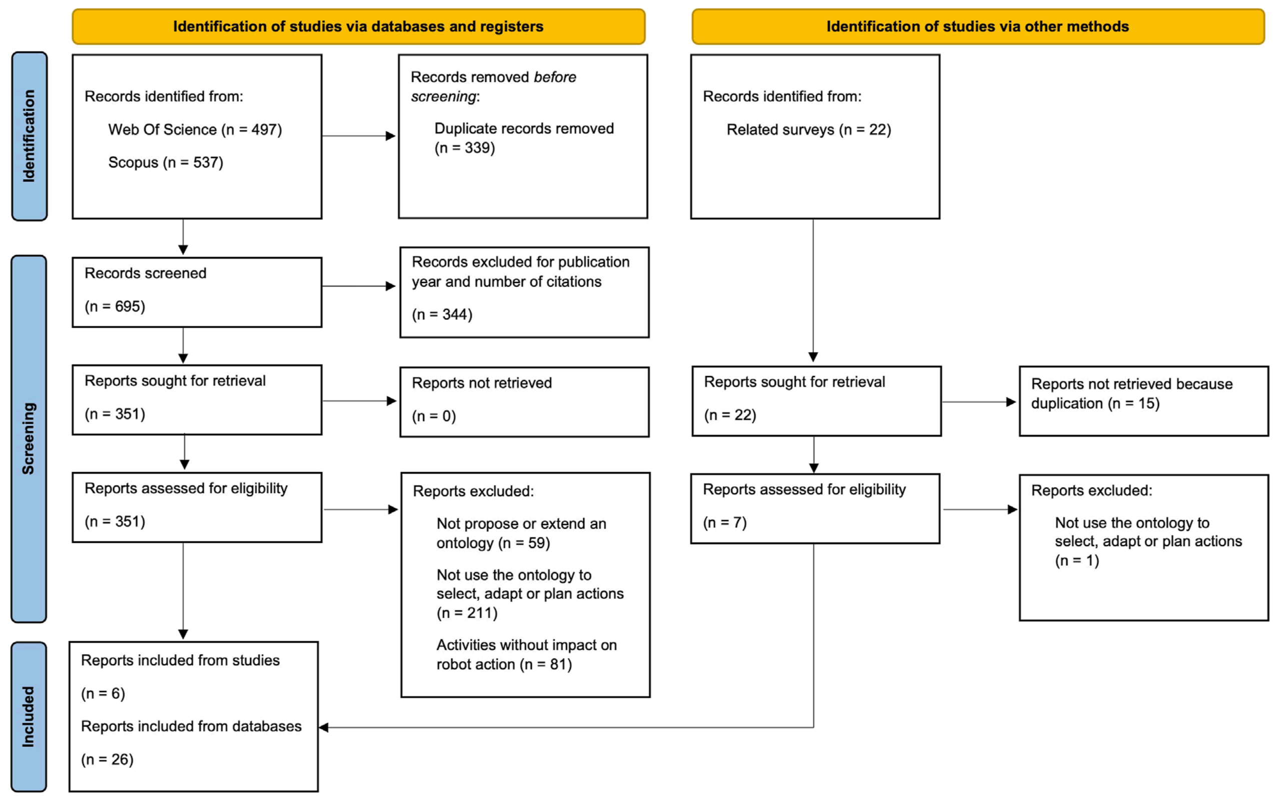

Identification of studies via databases and registers
Identification of studies via other methods
Identification
Screening
Included
Records identified from:
Web Of Science (n = 497)
Scopus (n = 537)
Records removed *before screening*:
Duplicate records removed (n = 339)
Records identified from:
Related surveys (n = 22)
Records screened (n = 695)
Records excluded for publication year and number of citations (n = 344)
Reports sought for retrieval (n = 351)
Reports not retrieved (n = 0)
Reports sought for retrieval (n = 22)
Reports not retrieved because duplication (n = 15)
Reports assessed for eligibility (n = 351)
Reports excluded:
Not propose or extend an ontology (n = 59)
Not use the ontology to select, adapt or plan actions (n = 211)
Activities without impact on robot action (n = 81)
Reports assessed for eligibility (n = 7)
Reports excluded:
Not use the ontology to select, adapt or plan actions (n = 1)
Reports included from studies (n = 6)
Reports included from databases (n = 26)

FIGURE 1
PRISMA flow diagram, adapted from Page et al. (2021).

inclusion criteria. After evaluating them, we included six more articles, resulting in a total of 32 articles in deep review. The entire process is depicted in Figure 1 with a Preferred Reporting Items for Systematic reviews and Meta-Analyses (PRISMA) flow diagram.

## 4.1 Related surveys

This section evaluates a variety of comparative studies and surveys on the application of KR&R to several robotic domains. These works address robot architecture, as well as specific topics such as path planning, task decomposition, and context comprehension. However, our approach transcends these specific domains, aiming to offer a more comprehensive view of the role of ontologies in enhancing robot understanding and autonomy.

Gayathri and Uma (2018) compare languages and planners for robotic path planning from the KR&R perspective. They discuss ontologies for spatial, semantic, and temporal representations and their corresponding reasoning. In Gayathri and Uma (2019), the same authors expand their review by focusing on DL for robot task planning.

Closer to our perspective are articles that compare and analyze different approaches for ontology-based robotic systems, specifically those articles that focus on what to model in an ontology. Cornejo-Lupa et al. (2020) compare and classify ontologies for simultaneous localization and mapping (SLAM) in autonomous robots. The authors compare domain and application ontologies in terms of (i) robot information such as kinematic, sensor, pose, and trajectory information; (ii) environment mapping such as geographical, landmark, and uncertainty information; (iii) time-related information and mobile objects; and (iv) workspace information such as domain and map dimensions. They focus on one important but specific part of robot operation, autonomous navigation, and, in a particular type of robot, mobile robots.

Manzoor et al. (2021) compare projects based on application, ideas, tools, architecture, concepts, and limitations. However, this article does not examine how architecture in the compared frameworks supports autonomy. Their review focuses on objects, environment maps, and task representations from an ontology perspective. It does not compare the different approaches regarding the robot's self-model. It also does not tackle the mechanisms and consequences of using such knowledge to enhance the robot's reliability.

Perhaps the most relevant review from our perspective is the one by Olivares-Alarcos et al. (2019). They analyze five of the main projects that use KBs to support robot autonomy. They ground their analysis in (i) ontological terms, (ii) the capabilities that support robot autonomy, and (iii) the application domain. Moreover, they discuss robots and environment modeling. This work is closely related to the development of the recent IEEE Standard 1872.2 (IEEE SA, 2022).

Our approach differentiates itself from previous reviews because we focus on modeling not only robot actions and their environments but also engineering design knowledge. This type of knowledge is not often considered but provides a deeper understanding of the robot's components and its interaction, design requirements, and possible alternatives to reach a mission. We also base our comparison on explicit knowledge of the mission and how we can ensure that it satisfies the user-expected performance. For this reason, we focus on works that select, adapt, or plan actions. Our search used more flexible inclusion criteria to analyze a variety of articles, even if they address only some of the issues or their ontologies are not publicly available. We have adopted this wide perspective to draw a general picture of different approaches that build the most important capabilities for dependable autonomous robots.

## 4.2 Review scope

Our analysis focuses on exploring the role of ontologies in advancing the autonomy of robotic systems. For this purpose, we delve into research articles that address ontologies created or extended to select, adapt, or plan actions autonomously. We excluded studies that solely encode simulations or utilize ontologies for non-action-related tasks, aiming to concentrate on contributions directly impacting robot action. Furthermore, our scope excludes non-ontology-based approaches, even if they are important for dependability or autonomy, such as all the developments in safe-critical systems.

An ontology is a shared conceptualization that structures objects into classes, which can be seen as categories within the domain of discourse. While interaction with the world involves specific objects, referred to as individuals or instances, much reasoning occurs at the class level. Classes can possess properties that characterize the collection of objects they represent or are related to other classes. Additionally, they are organized through inheritance, expressed by the subclass relation. Figure 2 represents a simple ontology with partial information about zones in a manufacturing plant.

Consider the assertion of the class `ProductionZone` with the property `robot_accessible`. According to this assertion, every instance of `ProductionZone`, such as the specific area `production_01`, is considered `robot_accessible`. In this hierarchy, `Assembly` is a subclass of `ProductionZone`. Given this hierarchy, we can deduce that every instance of `AssemblyZone` inherits the properties of its superclass. Every individual of it, such as `assembly_01`, inherits the `robot_accessible` property. Zones without the property, such as `Office` or `Control Room`, are considered restricted.

This kind of reasoning based on classification and consistency can be used to derive new assertions about elements in the robot environment and the robot itself. In this example, the robot can use this information to traverse only the accessible areas when selecting a route to traverse the plant. Therefore, the use of knowledge-driven policies facilitates the fulfillment of safety regulations and compliance standards and can enhance human understanding of the decisions the robot is making.

If the ontology represented components, capabilities, and goals, these conclusions could be used to adapt the system. Figure 3 represents a robotic ontology based on components, capabilities, goals, and values to determine which design alternatives are available

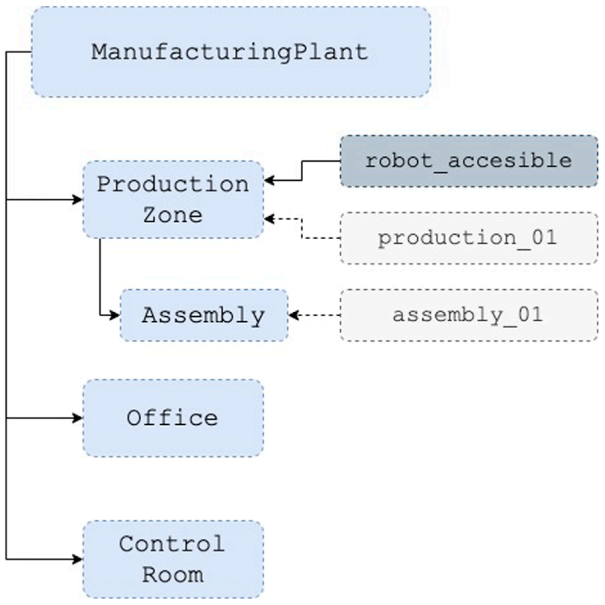


FIGURE 2
Simple example of manufacturing zones ontology. Classes in light blue, properties in dark blue, and individuals in gray.

for the robot. Following the approach of TOMASys Hernández et al. (2018), we are working on a system that selects the most suitable reconfiguration action for the robot. For example, if a component fails, the ontology can find another component that provides an equivalent capability so the robot can use it to complete the mission. This KB is also useful for determining which interfaces the system requires in that case, which metrics would be affected, and which stakeholders should be advised of the change. More information on the fundamental aspects of this ontology can be found in Aguado et al. (2024).

The analysis developed here does not address different approaches to solving a specific problem; rather, it focuses on examining, for each framework, the capabilities explicitly represented in ontologies or those that leverage the ontology at runtime to enhance the operation of the robot.

The survey is centered on action-related conceptualizations, with the objective of finding the theoretical foundations and relevant applications of ontological frameworks within the field of robotics. For this reason, and especially given that most of the articles do not provide source files for the ontology, we focus on the concepts, and discussions regarding scalability, computational cost, and efficiency are not within the scope of our study.

## 4.3 Review audience

This survey is directed towards a specific audience that is not seeking introductory knowledge about ontologies or their basic usage. That content can be found in resources such as Staab and Studer (2009). For an introduction from the robotics perspective, refer to Chapter 10 in Russell and Norvig (2021). Our target audience consists of people interested in understanding

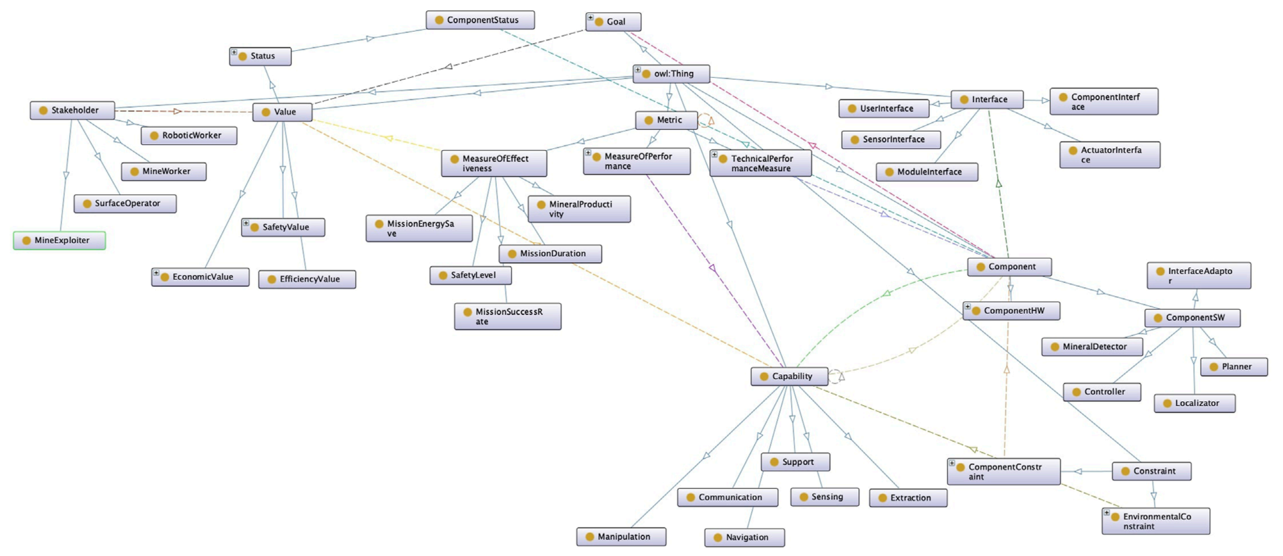


FIGURE 3
Example of an ontology to formalize robotics system models and their runtime deployment to select the most appropriate reconfiguration strategy in response to unexpected circumstances. This ontology represents the robot components, capabilities, and mission goals. The objective of the ontology is to find adaptation solutions that are not predefined or *ad hoc*; rather, they are computed during operation based on explicit engineering knowledge from the design phase.

the practical application of concepts related to robot autonomy during robot task execution. We aim to cater to researchers, practitioners, and academics who already possess a foundational understanding of ontologies and are now seeking to explore how these concepts intersect with and enhance the execution of tasks in robotic systems.

## 5 A survey of applications using ontologies for robot autonomy

The following sections review and compare some projects to find their specific contributions to the field of robotic ontologies. We have organized the analysis of the projects under review based on the application domain—manipulation, navigation, social, and industrial—in which they have been deployed. However, most of the work has a broader perspective and may be applied to other domains. Each analyzed work includes a brief description and a discussion of the above elements. Finally, each domain provides a comparative table with the most relevant aspects of each capability.

### 5.1 Manipulation domain

In the domain of robotics, manipulation refers to the control and coordination of robotic arms, grippers, and other mechanical systems to interact with objects in the physical world. This includes a wide range of applications, from industrial automation to tasks in unstructured environments, such as household chores or healthcare assistance. In this section, most of the work under analysis focuses on domestic applications for manipulation.

#### 5.1.1 KnowRob and KnowRob-based approaches

KnowRob[13] is a framework that provides a KB and a processing system to perform complex manipulation tasks (Tenorth and Beetz, 2009; Tenorth and Beetz, 2013). KnowRob2[14] (Beetz et al., 2018) represents the second generation of the framework and serves as a bridge between vague instructions and specific motion commands required for task execution.

KnowRob2's primary objective is to determine the appropriate action parametrization based on the required motion and identify the physical effects that must be achieved or avoided. For example, if the robot is asked to pick up a cup and pour out its contents, the KB retrieves the necessary action of pouring, which includes a sub-task to grasp the source container. Subsequently, the framework queries for pre-grasp and grasp poses, along with grasp force, to establish the required motion parameters.

KnowRob ontology includes a spectrum of concepts related to robots, including information about their body parts, connections, sensing and action capabilities, tasks, actions, and behavior. Objects are represented with their parts, functionalities, and configuration, while context and environment are also taken into account. Additionally, the ontology incorporates temporal predicates based on event logic and time-dependent relations.

While KnowRob's initial ontology was based on Cyc (Lenat, 1995), which was designed to understand how the world works by representing implicit knowledge and performing human-like reasoning, Cyc has remained proprietary. OpenCyc, a public initiative related to Cyc, is no longer available. The KnowRob framework later transitioned to DUL, which was chosen for its

13 http://KnowRob.org

14 https://github.com/KnowRob/KnowRob

compatibility with concepts for autonomous robots. KnowRob uses Prolog as a query and assert interface, but all perception, navigation, and manipulation actions are encoded in plans rather than Prolog queries or rules.

One of the main expansions of KnowRob is RoboEarth, a worldwide open-source platform that allows any robot with a network connection to generate, share, and reuse data (Waibel et al., 2011). It uses the principle of linked data connections through web and cloud services to speed up robot learning and adaptation in complex tasks.

RoboEarth uses an ontology to encode concepts and relations in maps and objects and a SLAM map that provides the geometry of the scene and the locations of objects with respect to the robot (Riazuelo et al., 2015). Each robot has a self-model that describes its kinematic structure and a semantic model to provide meaning to robot parts, such as a set of joints that form a gripper. This model also includes actions, their parameters, and software components, such as object recognition systems (Tenorth et al., 2012).

Another example of a KnowRob-based application is Crespo et al. (2018). In this case, the focus is on using semantic concepts to annotate a SLAM map with additional conceptualizations. This application diverges somewhat from KnowRob's initial emphasis on robot manipulators. The work models the environment utilizing semantic concepts but specifically captures the relationships between rooms, objects, object interactions, and the utility of objects.

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* KnowRob and RoboEarth incorporate inference mechanisms to abstract sensing information, particularly in the context of object recognition (Beetz et al., 2018; Crespo et al., 2018). As highlighted in Tenorth and Beetz (2013), inference processes information perceived from the external environment and abstracts it to the most appropriate level while retaining the connection to the original percepts. These ontologies serve as shared conceptualizations that accommodate various data types and support various forms of reasoning: effectively handling uncertainties arising from sensor noise, limited observability, hallucinated object detection, incomplete knowledge, and unreliable or inaccurate actions (Tenorth and Beetz, 2009).
- *Decision making and planning:* KnowRob places a strong emphasis on determining action parametrizations for successful manipulation, using hybrid reasoning with the goal of *reasoning with eyes and hands* (Beetz et al., 2018).

This approach equips KnowRob with the ability to reason about specific physical effects that can be achieved or avoided through its motion capabilities. Although KnowRob's KB might exhibit redundancy or inconsistency, the reasoning engine computes multiple hypotheses, subjecting them to plausibility and consistency checks and ultimately selecting the most promising parametrization. The planning component of KnowRob2 is tailored for motion planning and solving inverse kinematics problems. Tasks are dynamically assembled on the basis of the robot's situation.

- *Prediction and monitoring:* KnowRob uses its ontology to represent the evolution of the state, facilitating the retrieval of semantic information and reasoning (Beetz et al., 2018). Through these heterogeneous processes, the framework can predict the most appropriate parameters for a given situation. However, the monitoring capabilities within this framework are limited to objects in the environment. Unsuccessful experiences are labeled and stored in the robot's memory, contributing to the selection of action parameters in subsequent scenarios. The framework introduces NEEMs, allowing queries about the robot's actions, their timing, execution details, success outcomes, the robot's observations, and beliefs during each action. This knowledge is used primarily during the learning process.
- *Reasoning:* KnowRob2 incorporates a hybrid reasoning kernel comprising four KBs with their corresponding reasoning engines (Beetz et al., 2018):
- Inner World KB: Contains CAD and mesh models of objects positioned with accurate 6D poses, enhanced with a physics simulation.
- Virtual KB: Computed on demand from the data structures of the control system.
- Logic KB: Comprises abstracted symbolic sensor and action data enriched with logical axioms and inference mechanisms. This type of reasoning is the focus of our discussion in this article.
- Episodic Memories KB: Stores experiences of the robotic agent.
- *Execution:* KnowRob execution is driven by competency questions to bridge the gap between undetermined instructions and action. The framework incorporates the Cognitive Robot Abstract Machine (CRAM), where one of its key functionalities is the execution of the plan. The framework provides a plan language to articulate flexible, reliable, and sensor-guided robot behavior. The executor then updates the KB with information about perception and action results, facilitating the inference of new data to make real-time control decisions (Beetz et al., 2010).

RoboEarth and earlier versions of KnowRob rely on action recipes for execution. Before executing an action recipe, the system verifies the availability of the skills necessary for the task and orders each action to satisfy the constraints. In cases where the robot encounters difficulties in executing a recipe, it downloads additional information to enhance its capabilities (Tenorth et al., 2012). Once the plan is established, the system links robot perceptions with the abstract task description given by the action recipe. RoboEarth ensures the execution of reliable actions by actively monitoring the link between robot perceptions and actions (Waibel et al., 2011).

- *Communication and coordination:* RoboEarth (Tenorth et al., 2012) uses a communication module to facilitate the exchange of information with the web. This involves making web requests to upload and download data, allowing the construction and updating of the KB.
- *Learning:* KnowRob includes learning as part of its framework. It focuses on acquiring generalized models that capture how the physical effects of actions vary depending on the motion parametrization (Beetz et al., 2018). The learning process involves abstracting action models from the data, either by identifying a class structure among a set of entities or by

grouping the observed manipulation instances according to a specific property (Tenorth and Beetz, 2009). KnowRob2 extends its capabilities with the Open-EASE knowledge service (Beetz et al., 2015), offering a platform to upload, access, and analyze NEEMs of robots involved in manipulation tasks. NEEMs use descriptions of events, objects, time, and low-level control data to establish correlations between various executions, facilitating the learning-from-experience process.

#### 5.1.2 Perception and Manipulation Knowledge

The Perception and Manipulation Knowledge (PMK)[15] framework is designed for autonomous robots, with a specific focus on complex manipulation tasks that provide semantic information about objects, types, geometrical constraints, and functionalities. In concrete terms, this framework uses knowledge to support task and motion planning (TAMP) capabilities in the manipulation domain (Diab et al., 2019).

PMK ontology is grounded in IEEE Standard 1872.2 (IEEE SA, 2022) for knowledge representation in the robotic domain, extending it to the manipulation domain by incorporating sensor-related knowledge. This extension facilitates the link between low-level perception data and high-level knowledge for comprehensive situation analysis in planning tasks. The ontological structure of PMK comprises a meta-ontology for representing generic information, an ontology schema for domain-specific knowledge, and an ontology instance to store information about objects. These layers are organized into seven classes: *feature*, *workspace*, *object*, *actor*, *sensor*, *context reasoning*, and *actions*. This structure is inspired by OUR-K (Lim et al., 2011), which is further described in Section 5.3).

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* The PMK ontology incorporates RFID sensors and 2D cameras to facilitate object localization, explicitly grouping sensors to define equivalent sensing strategies. This design enables the system to dynamically select the most appropriate sensor based on the current situation. PMK establishes relationships between classes such as *feature*, *sensor*, and *action*. For example, it stores poses, colors, and IDs of objects obtained from images.
- *Decision making and planning:* PMK augments TAMP parameters with data from its KB. The KB contains information on action feasibility considering object features, robot capabilities, and object states. The TAMP module utilizes this information, combining a fast-forward task planner with physics-based motion planning to determine a feasible sequence of actions. This helps determine where the robot should place an object and the associated constraints.
- *Reasoning:* PMK reasoning targets potential manipulation actions employing description logic's inference for real-time information, such as object positions through spatial reasoning and relationships between entities in different classes. Inference mechanisms assess robot capabilities, action constraints, feasibility, and manipulation behaviors, facilitating the integration of TAMP with the perception module. This process yields information about constraints such as interaction parameters (e.g., friction, slip, maximum force), spatial relationships (e.g., inside, right, left), feasibility of actions (e.g., arm reachability, collisions), and action constraints related to object interactions (e.g., graspable from the handle, pushable from the body).
- *Interaction and design:* PMK represents interaction as manipulation constraints, specifying, for example, which part of an object is interactable, such as a handle. It also considers interaction parameters such as friction coefficient, slip, or maximum force.

#### 5.1.3 Failure Interpretation and Recovery in Planning and Execution

Failure Interpretation and Recovery in Planning and Execution (FailRecOnt)[16] is an ontology-based framework featuring contingency-based task and motion planning. This innovative system empowers robots to handle uncertainty, recover from failures, and engage in effective human–robot interactions. Grounded in the DUL ontology, it addresses failures and recovery strategies, but it also takes some concepts from CORA and SUMO for robotics and ontological foundations, respectively.

The framework identifies failures through the *non-realized situation* concept and proposes corresponding *recovery strategies* for actions. To improve the understanding of failure, the ontology models terms such as *causal mechanism*, *location*, *time*, and *functional considerations*, which facilitates a reasoning-based repair plan (Diab et al., 2020; Diab et al., 2021).

Failure ontology requires a system knowledge model in terms of *tasks*, *roles*, and *object concepts*. Lastly, FailRecOnt has some similarities with KnowRob; both target manipulation tasks are at least partially based on DUL and share some of the authors. Moreover, they propose using PMK as a model of the system (Diab et al., 2021).

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* Perception in FailRecOnt is limited to action detection to detect abnormal events. For geometric information and environment categorization, the framework leverages PMK to abstract perceptual information related to the environment.
- *Decision making and planning:* FailRecOnt is structured into three layers: planning and execution, knowledge, and an assistant low-level layer. The planning and execution layer provides task planning and a task manager module. The assistant layer manages perception and action execution for the specific robot, determining how to sense an action and checking whether a configuration is collision-free. The framework has been evaluated for a task that involves storing an object in a given tray according to its color; it can handle situations such as facing a closed or flipped box and continuing the plan (Diab et al., 2021).
- *Prediction and monitoring:* Monitoring is a crucial aspect of the FailRecOnt framework. It continuously monitors executed actions and signals a failure to the recovery module if an error

15 https://github.com/MohammedDiab1/PMK

16 https://github.com/ease-crc/failrecont

occurs. The reasoning component interprets potential failures and, if possible, triggers a recovery action to repair the plan.

- *Reasoning:* FailRecOnt ontology describes how the perception of actions should be formalized. Reasoning selects an appropriate recovery strategy depending on the kind of failure, why it happened, and if other activities are affected, etc.
- *Execution:* FailRecOnt relies on planning for execution. The task planner generates a sequence of symbolic-level actions without geometric considerations. Geometric reasoning comes into play to establish a feasible path. During action execution, the framework monitors each manipulation action for possible failures by sensing. Reasoning is then applied to interpret potential failures, identify causes, and determine recovery strategies.
- *Communication and coordination:* FailRecOnt incorporates the reasoning for communication failure from Diab et al. (2020) to address failures in scenarios where multiple agents exchange information.

### 5.1.4 Probabilistic Logic Module

Probabilistic Logic Module (PLM) offers a framework that integrates semantic and geometric reasoning for robotic grasping (Antanas et al., 2019). Specific details about the KB, such as the source files, are not publicly available, so the information provided here is derived from articles about the framework.

The primary focus of this work is on an ontology that generalizes similar object parts to semantically reason about the most probable part of an object to grasp, considering object properties and task constraints. This information is used to reduce the search space for possible final gripper poses. This acquired knowledge can also be transferred to objects within the same category.

The object ontology comprises specific objects such as *cup* or *hammer*, along with supercategories based on functionality, such as *kitchen container* or *tool*. The task ontology encodes grasping tasks with objects, such as *pick and place right* or *pour in*. Additionally, a third ontology conceptualizes object-task affordances, considering the manipulation capabilities of a two-finger gripper and the associated probability of success.

In using the ontology, high-level knowledge is combined with low-level learning based on visual shape features to enhance object categorization. Subsequently, high-level knowledge utilizes probabilistic logic to translate low-level visual perception into a more promising grasp planning strategy.

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* This approach is based on visual perception, employing vision to identify the most suitable part of an object for grasping. The framework utilizes a low-level perception module to label visual data with semantic object parts, such as detecting the top, middle, and bottom areas of a cup and its handle. The probabilistic logic module combines this information with the affordances model.
- *Decision making and planning:* PLM uses ontologies to support grasp planning. It integrates ontological knowledge with probabilistic logic to translate low-level visual perception into an effective grasp planning strategy. Once an object is categorized and its affordances are inferred, the task ontology determines the most likely object part to be grasped, thereby reducing the search space for possible final gripper poses. Subsequently, a low-level shape-based planner generates a trajectory for the end effector.
- *Prediction and monitoring:* Although this framework does not specifically predict or monitor robot actions, it uses task prediction to select among alternative tasks based on their probability of success but does not actively monitor them.
- *Reasoning:* PLM uses semantic reasoning to grasp. It selects the best grasping task based on object affordances, addressing the uncertainty of visual perception through a probabilistic approach.
- *Learning:* Learning techniques are used to identify the visual characteristics of the shape, which are then categorized. The robot utilizes the acquired knowledge for grasp planning.

Table 1 provides a concise comparison of the four frameworks analyzed. Although all of them address perception and categorization, decision making and planning, and reasoning, the specific aspects involved depend on the perspective. Only half of the frameworks explicitly utilize their KB for execution, prediction and monitoring, communication and coordination, and learning. In particular, integration and design are addressed exclusively by the PMK framework.

## 5.2 Navigation domain

The navigation domain describes the challenges and techniques involved in enabling robots to autonomously move around their environment. This involves the processes of guidance, navigation, and control (GNC), incorporating elements such as computer vision and sensor fusion for perception and localization, as well as control systems and artificial intelligence for mapping, path planning, or obstacle avoidance.

### 5.2.1 Teleological and Ontological Model for Autonomous Systems

Teleological and Ontological Model for Autonomous Systems (TOMASys)[17] is a metamodel designed to consider the functional knowledge of autonomous systems, incorporating both teleological and ontological dimensions. The teleological aspect includes engineering knowledge, which represents the intentions and purposes of system designers. The ontological dimension categorizes the structure and behavior of the system.

TOMASys serves as a metamodel to ensure robust operation, focusing on mission-level resilience (Hernández et al., 2018). This metamodel relies on a functional ontology derived from the Ontology for Autonomous Systems (OASys) (Bermejo-Alonso et al., 2010), establishing connections between the robot's architecture and its mission. The core concepts in TOMASys include functions, objectives, components, and configurations. However, it operates as a metamodel and intentionally avoids representing specific features of the operational environment, such as objects, maps, etc.

17 https://github.com/meta-control/mc_mdl_tomasys

TABLE 1 Use of ontologies in the manipulation domain for perception and categorization (P/C), decision making and planning (DM/P), prediction and monitoring (P/M), reasoning (R), execution (E), communication and coordination (C/C), interaction and design (I/D) and learning (L).

| | KnowRob (Tenorth and Beetz, 2009; Beetz et al., 2018; Tenorth and Beetz, 2013; Waibel et al., 2011; Tenorth et al., 2012; Crespo et al., 2018; Riazuelo et al., 2015) | PMK (Diab et al., 2019) | FailRecOnt (Diab et al., 2021; Diab et al., 2020) | PLM (Antanas et al., 2019) |
|---|---|---|---|---|
| P/C | Integrate data types and object recognition | Dynamically select the most appropriate sensor | Action sensing to detect abnormal events | Identify the most suitable part of an object for grasping |
| DM/P | Action parametrization for successful manipulation | Task and motion planning feasibility | How to sense an action and check if a configuration is collision-free | Translate low-level visual perception into effective grasp planning |
| P/M | Predict the most appropriate parameters. Monitoring objects in the environment | - | Failure detection and identification of failed recovery attempts | Task prediction to select among alternatives |
| R | Hybrid, symbolic reasoning for sensor and action data | Integration of task and motion planning with the perception module | How the perception of actions should be formalized | Select the best grasping task based on object affordances |
| E | Match undetermined instructions and actions | - | Symbolic-level actions without geometric considerations | - |
| C/C | Web exchange of information | - | Communication failures between multiple agents | - |
| I/D | - | Information about which part of an object is interactable, constraints | - | - |
| L | Use previous descriptions of events, objects, time, and low-level control data | - | - | Categorize visual shape features |

At the core of the TOMASys framework is the metacontroller. While a conventional controller closes a loop to maintain a system component's output close to a set point, the metacontroller closes a control loop on top of a system's functionality. This metacontroller triggers reconfiguration when the system deviates from the functional reference, allowing the robot to adapt and maintain desired behavior in the presence of failures. Explicit knowledge of mission requirements is leveraged for reconfiguration using the system's functional specifications captured in the ontology.

In practice, TOMASys has been applied to various robots and environments, particularly for navigation tasks. Examples include its application to an underwater mine explorer robot (Aguado et al., 2021) and a mobile robot patrolling a university campus (Bozhinoski et al., 2022).

The detailed analysis of comparison criteria follows:

- *Decision making and planning:* In TOMASys, the metacontrol system manages decision making by adjusting parameters and configurations to address contingencies and mission deviations. It assumes the presence of a nominal controller responsible for standard decisions. In the case of failure detection or unmet mission requirements, the metacontroller selects an appropriate configuration. The planning process is integrated into the metacontrol subsystem, where reconfiguration decisions can impact the overall system plan, potentially altering parameters, components, functionalities, or even relaxing mission objectives to ensure task accomplishment.
- *Prediction and monitoring:* Monitoring is a critical aspect of TOMASys, providing failure models to detect contingencies or faulty components. Reconfiguration is triggered not only in the event of failure but also when mission objectives are not satisfactorily achieved. Observer modules are used to monitor reconfigurable components of the system.
- *Reasoning:* TOMASys uses a DL reasoner for real-time system diagnosis. It propagates component failures to the system level, identifying affected functionalities and available alternatives. This reasoning process helps to select the most promising alternative to fulfill mission objectives.
- *Execution:* The execution in TOMASys follows the monitor-analyze-plan-execute (MAPE-K) loop (IBM Corporation, 2005). It evaluates the mission and system state through monitoring observers, uses ontological reasoners for assessing mission objectives and propagating component failures, decides reconfigurations based on engineering and runtime knowledge, and executes the selected adaptations.
- *Communication and coordination:* TOMASys uses its hierarchical structure to coordinate components working toward a common goal. Components utilize roles that specify

parameters for specific functions, and bindings facilitate communication by connecting component roles with function specifications during execution. Bindings are crucial for detecting component failures or errors. In cases where the metacontroller cannot handle errors, a function design log informs the user.

- *Interaction and design:* TOMASys provides a metamodel that leverages engineering models from design time to runtime. This approach aims to bridge the gap between design and operation, relying on functional and component modeling to map mission requirements to the engineering structure. The explicit dependencies between components, roles, and functions, along with specifications of required component types based on functionality, support the system's adaptability at runtime.

### 5.2.2 Ontology-based multi-layered robot knowledge framework

The Ontology-based Multi-layered Robot Knowledge Framework (OMRKF) aims to integrate high-level knowledge with perception data to enhance the intelligence of a robot in its environment (Suh et al., 2007). Specific details about the KB, such as the source files, are not publicly available, so the information provided here is derived from articles about the framework.

The framework is organized into knowledge boards, each representing four knowledge classes: perception, activity, model, and context. These classes are divided into three knowledge levels (high, middle, and low). Perception knowledge involves visual concepts, visual features, and numerical descriptions. Similarly, activity knowledge is classified into service, task, and behavior, while model knowledge includes space, objects, and their features. The context class is organized into high-level context, temporal context, and spatial context.

At each knowledge level, OMRKF employs three ontology layers: (a) a meta-ontology for generic knowledge, (b) an ontology schema for domain-specific knowledge, and (c) an ontology instance to ground concepts with application-specific data. The framework uses rules to define relationships between ontology layers, knowledge levels, and knowledge classes.

OMRKF facilitates the execution of sequenced behaviors by allowing the specification of high-level services and guiding the robot in recognizing objects even with incomplete knowledge. This capability enables robust object recognition, successful navigation, and inference of localization-related knowledge. Additionally, the framework provides a querying-asking interface through Prolog, enhancing the robot's interaction capabilities.

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* OMRKF includes perception as one of its knowledge classes, specifically addressing the numerical descriptor class in the lower-level layer. Examples of these numerical descriptors are generated by robot sensors and image processing algorithms such as Gabor filter or scale-invariant feature transform (SIFT) (Suh et al., 2007).
- *Decision making and planning:* OMRKF uses an event calculus planner to define the sequence to execute a requested service. The framework relies on query-based reasoning to determine how to achieve a goal. In cases of insufficient knowledge, the goal is recursively subdivided into subgoals, breaking down the task into atomic functions such as *go to*, *turn*, and *extract feature*. Once the calculus planner generates an output, the robot follows a sequence to complete a task, such as the steps involved in a delivery mission.
- *Reasoning:* OMRKF employs axioms, such as the inverse relation of *left* and *right* or *on* and *under*, to infer useful facts using the ontology. The framework uses Horn rules to identify concepts and relations, enhancing its reasoning capabilities.

### 5.2.3 Smart and Networked Underwater Robots in Cooperation Meshes ontology

The Smart and Networked Underwater Robots in Cooperation Meshes (SWARMs) ontology addresses information heterogeneity and facilitates a shared understanding among robots in the context of maritime or underwater missions (Li et al., 2017). Specific details about the KB, such as the source files, are not publicly available, so the information provided here is derived from articles about the ontology.

SWARMs leverages the probabilistic ontology PR-OWL[18] to annotate the uncertainty of the context based on the multi-entity Bayesian network (MEBN) theory (Laskey, 2008). This allows SWARMs to perform hybrid reasoning on (i) the information exchanged between robots and (ii) environmental uncertainty.

SWARMs establishes a core ontology to interrelate several domain-specific ontologies. The core ontology manages entities, objects, and infrastructures. These ontologies include:

- *Mission and Planning Ontology*: Provides a general representation of the entire mission and the associated planning procedures.
- *Robotic Vehicle Ontology*: Captures information on underwater or surface vehicles and robots.
- *Environment Ontology*: Characterizes the environment through recognition and sensing.
- *Communication and Networking Ontology*: Describes the communication links available in SWARMs to interconnect different agents involved in the mission, enabling communication between the underwater segment and the surface.
- *Application Ontology*: Provides information on scenarios and their requirements. PR-OWL is included in this layer to handle uncertainty.

Li et al. (2017) present an example using SWARMa to monitor chemical pollution based on a probability distribution. SWARMs incorporates this model into the ontology and uses MEBN to deduce the emergency level of the polluted sea region.

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* The SWARMs ontology provides a shared framework to represent the underwater environment. For this reason, it contains classes on sensors and the main concepts of the environment to understand it through its properties, such as water salinity, conductivity, temperature,

18 https://www.pr-owl.org

and currents. Uncertainty reasoning is critical for the categorization of sensor information, especially in harsh maritime and underwater environments.

- *Decision making and planning:* SWARMs uses two levels of abstraction. High-level planning allows the user to describe different tasks related to operations without specifying the exact actions that each robotic vehicle must perform. Low-level planning is performed on each robot to generate waypoints, actions, and other similar low-level tasks.
- *Reasoning:* SWARMs uses a hybrid context reasoner that combines ontological rule-based reasoning with MEBN for probabilistic annotations.
- *Communication and coordination:* A main concern in SWARMs is cooperation; robots share tasks, operations, and actions. The ontology provides transparent information sharing to support the heterogeneity of the data. It also provides an abstraction for communication and networking, describing the communication links available from command control stations to vehicles and from vehicles to command control stations.

### 5.2.4 Robot Task Planning Ontology

The Robot Task Planning Ontology (RTPO) Sun et al. (2019b) is an effective knowledge representation framework for robot task planning. Specific details about the KB, such as the source files, are not publicly available, so the information provided here is derived from articles about the ontology.

Designed to accommodate temporal, spatial, continuous, and discrete information, RTPO prioritizes scalability and responsiveness to ensure practicality in task planning. The ontology comprises three main components: robot, environment, and task.

- *Robot Ontology*: Comprises hardware and software details, location information, dynamic data, and more. Sensors are explicitly modeled as a subclass of hardware, specifying the measurable aspects of the environment. In an experimental context, the robot's perception is limited to obstacles.
- *Environment Ontology*: Focuses on location and recognition of humans and objects, the environment map, and information collected from other robots.
- *Task Ontology*: Aims to understand how to decompose high-level tasks into atomic actions and adapt plans when the environment changes.

This multi-ontology approach allows RTPO to capture the details of robot task planning by representing both the robot's internal state and its interactions with the environment.

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* The robot ontology in RTPO considers sensors as a subclass of hardware, specifying measurable aspects of the environment. In the experiment discussed, the robot's perception is limited to detecting obstacles.
- *Decision making and planning:* Task planning is performed using a domain and a problem file, as in the Planning Domain Definition Language (PDDL). The algorithm described by Sun et al. (2019b) generates and uses these files.

The task planning process is realized by matching the execution preconditions of atomic actions and their effects on the environment from the initial state to the goal state. With this approach, the task planning module can also be executed by changing the input tasks while considering the present action resources. The plans generated by the task planning algorithm are added to the ontology to improve the efficiency of task planning if the same task needs to be planned again.

- *Reasoning:* The reasoning process in RTPO involves updating and storing the knowledge within the ontology. The scalability of robot knowledge aims to enhance the efficiency of reasoning. Experimental scenarios involving the addition of elements to the indoor environment and corresponding KB instances demonstrate changes in consumed time, particularly affecting knowledge query speed. The authors show real-time performance in an application that involves 52,000 individuals, although the impact on the planning process is not explicitly detailed.
- *Communication and coordination:* This process is not explicitly explained by (Sun et al. 2019b). However, a scenario with three robots and two humans is described. Communication among the three robots is highlighted, emphasizing knowledge sharing. The relationships between these entities can be defined by users or developers based on their requirements. In situations with various robots mapping different rooms and using various sensors, the ontology facilitates linking and adding knowledge to constraints to maintain coherence.
- *Learning:* RTPO incorporates learning by updating and adding the plans generated by the task planning algorithm into the ontology. This iterative process aims to improve the efficiency of future task planning, especially when the same task is encountered again.

### 5.2.5 Guidance, navigation, and control for planetary rovers

Burroughes and Gao (2016) present an architectural solution to address limitations in autonomous software and GNC structures designed for extraterrestrial planetary exploration rovers. Specific details about the KB, such as the source files, are not publicly available, so the information provided here is derived from articles about the framework.

This framework uses an ontology to facilitate the autonomous reconfiguration of mission goals, software architecture, software components, and the control of hardware components during runtime. To manage complexity, the self-reconfiguration ontology is organized into modules. The base ontology serves as an upper ontology, including modules that delineate logic, numeric aspects, temporality, fuzziness, confidences, processes, and block diagrams. Additional modules describe the functions of software, hardware, and the environment.

The primary focus of this framework is reconfiguration. Once the ontology manager detects an undesired state change through the monitoring process, it activates a safety mode. In this safety mode, the system can execute either a reactive plan or create a new plan based on inferences drawn from the new situation.

Following the replanning process, new mission goals are established, allowing the robot to exit safety mode and resume normal operation. The framework specifically aims to minimize odometry errors and ensure safety during travel. To achieve this objective, the connective architecture and processes, such as navigation, localization, control, mapping, as well as sensors and the locomotion system, can be fine-tuned and optimized according to the environment and faulty hardware conditions. For example, certain methods may prioritize tolerance to sensor noise over absolute accuracy in localization.

The detailed analysis of comparison criteria follows:

- *Decision making and planning:* Burroughes and Gao (2016) integrate a reactive approach with a deliberative layer for quick responses. Precalculated responses are prepared for likely changes, but in the absence of options, the rational agent resorts to deliberative techniques. Ontologies and the PDDL are used for knowledge representation and planning, respectively. Actions correspond to specific configurations of services, with plans defined using the initial state of the world, the goal state, the resources of the system, the safety criteria, and the rules. The approach employs a pruning algorithm to reduce possible actions and planning space.
- *Prediction and monitoring:* The focus is on replanning, requiring monitoring to trigger the process. Generic inspectors perform network, resource, and state checks, and specific inspectors manage tasks like checking camera performance, monitoring, and updating the ontology on the current state of the world. The system can decide the depth of monitoring for each subsystem, balancing computational resources and self-protection.
- *Reasoning:* Reasoning is used to configure the software elements of the rover, update the state of the world, and decide whether the system should re-plan to adapt to changes or use pre-established reactive responses. The ontology checks for knowledge incoherence when adding new information.
- *Execution:* The MAPE-K loop is used for self-reconfiguration. When the monitor detects a change, the ontology provides the knowledge to select how the system should evolve. The configuration of the components is established through planning or reactivity. Navigation and operation components are organized into modular services with self-contained functionality to allow reconfiguration.
- *Communication and coordination:* Communication requirements, such as publication rate, conditions, and effects, are used to establish appropriate communication links between modules, treating communication as a reconfigurable service within the framework.
- *Interaction and design:* While not explicitly detailing the design or requirements, the system reasons in terms of service capacities, considering measures such as accuracy or suitability for sensor noise in sensor processing algorithms. It also incorporates safety criteria to select system changes, providing design and engineering knowledge to select alternative designs based on runtime situations.

### 5.2.6 Collaborative context awareness in search and rescue missions

Chandra and Rocha (2016) present a framework for collaborative context awareness using an ontology that comprises high-level aspects of urban search and rescue (USAR) missions. Specific details about the KB, such as the source files, are not publicly available, so the information provided here is derived from articles about the framework.

The framework serves as an efficient knowledge-sharing platform to represent and correlate various mission concepts, including those related to agents and their capabilities, scenarios, and teams. The ontology facilitates the sharing of homogeneous knowledge among all robots and supports human–robot collaboration by reasoning based on rules provided by humans.

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* The system employs perceived and pre-processed information gathered from its own sensors, as well as from other agents, such as humans and robots, to continually update its KB. Specifically, it focuses on entities within context classes and their relationships, including information about smoke levels, visibility, location, and temperature.
- *Decision making and planning:* The framework offers an efficient knowledge-sharing strategy that improves decision-making processes. For example, it facilitates the management of a global map with shared events and real-time tracking of agent locations.
- *Reasoning:* In this framework, reasoning plays a crucial role in storing and disseminating information between agents. The KB retains essential details such as location, temperature, visibility, battery status, tasks, and detection of victims or fires. Furthermore, reasoning contributes to resilience against communication failures by storing information for future sharing. This process supports coordination, including the incorporation of new team members or the temporary removal and subsequent reintegration of teammates.
- *Communication and coordination:* Building on a foundation of multirobot cooperation, Chandra and Rocha (2016) structure the framework to represent the team and mission within an ontology. This enables the creation of a global map and the selection of suitable candidates for various tasks. Examples of coordination include telepresence and compensating for a teammate's immobility. Furthermore, the framework ensures resilience in the face of communication failures. For example, all events detected by a robot are logged in the local KB and shared across the team to synchronize knowledge. In situations of communication loss or isolation from teammates, a list of unattended events is maintained and shared after communication is restored.

### 5.2.7 Autonomous vehicle situation assessment and decision making

Huang et al. (2019) use ontologies for situation assessment and decision making in the context of autonomous vehicles navigating urban environments. Specific details about the KB, such as the source files, are not publicly available, so the information provided here is derived from articles about the ontologies.

The KB integrates vehicle information, storing details of the permissible *directions* a vehicle can take. Furthermore, the KB includes representations of both static and dynamic obstacles, as well as relevant information on the characteristics of roads, distinguishing between *highways* and *urban roads*. The ontology incorporates specific scenarios that the autonomous vehicle could encounter on the road, such as proximity to an *intersection*, traversing a *bridge*, or executing a *U-turn*. The autonomous driving system leverages this KB to assess the current driving scenario, facilitating well-informed decisions on whether to maintain its current trajectory or initiate a lane change.

The detailed analysis of comparison criteria follows:

- *Decision making and planning:* The decision-making process is tied to situation assessment, employing a methodology that evaluates the safety of the surroundings of the vehicle. This involves characterizing the regions around the car and assigning a binary safety value based on the presence of obstacles.

If a region is considered safe, the vehicle continues in its current lane; otherwise, a lane change is initiated. This decision-making step incorporates statistical indicators that account for velocity. Moreover, the model considers legitimacy by adhering to traffic rules during lane changes and reasonableness by querying the global planning path to identify the next road segment or lane, particularly when approaching intersections. This strategic approach prevents the system from optimizing locally at the expense of compromising the overarching global plan.

- *Reasoning:* The reasoner is used to generate behavioral decisions. It employs rules derived from traffic regulations and driving experiences, taking into account various factors that influence the road, such as speed limits, traffic lights, and the presence of surrounding obstacles.
- *Interaction and design:* Although the framework does not explicitly include concepts as design decisions, it addresses requirements such as legality by incorporating traffic rules and reasonableness. This ensures that the behavior of the lane change aligns with the main goal of global planning, avoiding changes to local optimization that may compromise the broader strategic plan.

#### 5.2.8 Search and rescue scenario

Sun et al. (2019a) introduce an ontology tailored for search and rescue (SAR), enhancing the decision-making capabilities of robots in complex and unpredictable scenarios. Specific details about the KB, such as the source files, are not publicly available, so the information provided here is derived from articles about the ontology. The SAR ontology comprises three interlinked subontologies: an *entity* ontology, an *environment* ontology, and a *task* ontology.

- The *entity* ontology includes various types of robots, including ground, underwater, and air robots. It delineates their constituent parts and specifications in terms of hardware and software aspects.
- The *environment* ontology extends the scope to store most of the elements present in SAR scenarios, including the environment map and objects that shall be recognized. Updated knowledge from the environment can be shared among other robots, although the specific mechanisms are not detailed in the presented use case.
- The *task* ontology encapsulates the task-related knowledge that is necessary for informed decision making. This involves task decomposition and allocation facilitated by a hierarchical structure. The ontology defines four typical tasks: *charge*, *search*, *rescue*, and *recognize.* Additionally, atomic actions are articulated by their effects on the state of the environment. The framework proposes a task planning algorithm that aligns the preconditions of execution with the effects on the environment, utilizing the SAR ontology as a foundational framework.

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* The SAR framework focuses on SLAM and autonomous navigation. The authors use a semantic map, which facilitates the recognition and localization of objects. The acquired information dynamically updates the environment ontology, establishing connections between the SLAM map and the semantic details. Bayesian reasoning enhances the precision of victim positioning, while QR code scanning streamlines the acquisition of semantic information, such as vital signs (e.g., heart rate and blood pressure), in disaster rescue scenarios.
- *Decision making and planning:* decision making in this framework is based on structured queries. It begins by defining tasks and identifying the requisite actions, followed by a comprehensive examination of the action properties, including time constraints, preconditions, and postconditions, to achieve the desired state. The planner then specifies a sequenced set of atomic actions. The program can adopt specific search algorithms for planning or reusing previously established plans.
- *Reasoning:* Before planning, the robot conducts a preliminary assessment to determine the suitability of the task for the current state. If considered appropriate, the planning phase starts, identifying the tasks the robot should execute under the given circumstances.
- *Execution:* The robot systematically executes the sequence of atomic actions. When faced with a previously planned task, the robot can query the task definition, retrieving the corresponding atomic action sequence.

Tables 2 and 3 summarize studies in the navigation domain. All frameworks address decision making, planning, and reasoning, mainly to organize tasks. Most of the works focus on perceiving and categorizing measurable aspects of the environment, along with communication and coordination among different elements and systems. TOMASys, the planetary rover scenario, and the SAR scenario explicitly handle prediction, monitoring, and execution. The use of engineering knowledge—interaction and design—is explicitly addressed in TOMASys, the planetary rover scenario, and the autonomous vehicle scenario. Learning is applied only in RTPO to improve the plan generation process.

### 5.3 Social domain

Social robots usually refer to the interaction between robots and humans in a variety of contexts, such as homes, healthcare, education, entertainment, and public spaces. The goal is to create robots that are not only technically capable but also socially aware

TABLE 2 Part 1: Use of ontologies in the navigation domain for perception and categorization (P/C), decision making and planning (DM/P), prediction and monitoring (P/M), reasoning (R), execution (E), communication and coordination (C/C), interaction and design (I/D) and learning (L).

| | TOMASys (Hernández et al., 2018;Bozhinoski et al., 2022; Aguado et al., 2021) | OMRKF (Suh et al., 2007) | SWARMs (Li et al., 2017) | RTPO (Sun et al., 2019b) |
|---|---|---|---|---|
| P/C | - | Lower-lever perception features such as numerical descriptors | Represent underwater environment properties and sensors | Specification of measurable aspects of the environment |
| DM/P | Adjust parameters and configurations to address mission contingencies | Calculus planner to accomplish goals and subgoals | Task-level planning and low-level planning to generate waypoints, actions, etc. | Generate and use domains and problem files to generate plans |
| P/M | Failure models to detect faulty components and goals not achieved | - | - | - |
| R | Propagate failures to system level, identify affected functionalities and available alternatives | Geometrical relationships between objects | Hybrid context reasoner: rule-based reasoning and probabilistic annotations | Evaluation of real-time performance in reasoning with 52,000 elements of indoor environments |
| E | MAPE-K loop to evaluate mission status | - | - | - |
| C/C | Component coordination | - | Robots share tasks, operations, and actions | Knowledge sharing between three robots and two humans |
| I/D | Use of metamodels to bridge the gap between design and operation | - | - | - |
| L | - | - | - | Update and add plans generated by the task planning algorithm back into the ontology |

and able to interact with humans in a manner that is natural, intuitive, and socially acceptable.

### 5.3.1 Ontology-based unified robot knowledge

The Ontology-based Unified Robot Knowledge (OUR-K) framework for service robots (Lim et al., 2011) consists of five knowledge classes: features, objects, spaces, contexts, and actions. It takes OMRKF, the concept of layer division, from its predecessor. Although OMRKF was originally evaluated in navigation domains, OUR-K extends its application to social robots operating within domestic environments. Specific details about the KB, such as the source files, are not publicly available, so the information provided here is derived from articles about the framework.

A notable feature of OUR-K is its ability to perform tasks even when provided with incomplete information. The framework incorporates a knowledge description of both the robot and its environment, employing algorithms for knowledge association. This involves logic, Bayesian inference, and heuristics; however, logical inference in OUR-K is specifically limited to performing associations between classes and ontological levels, with no indication of alternative inference mechanisms in the literature.

OUR-K includes mechanisms for object recognition, context modeling, task planning, space representation, and navigation. Regarding action representation, its descriptions are simpler and do not contemplate processes as did its predecessor, OMRKF.

Diab et al. (2018) extend OUR-K with a physics-based manipulation ontology to address the challenges that a motion planner might encounter. An actor class is introduced within the knowledge classes to describe the working constraints of the robot, enhancing the planner's capability to handle interaction dynamics. In addition, the authors propose a prediction mechanism for the entire OUR-K framework. Instead of relying on inferences, a semantic map is generated for categorizing and assigning manipulation constraints, using reasoning based on logical axioms. This approach is evaluated in the context of a specific manipulation task, where a robot serves a liquid drink contained in a can.

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* OUR-K follows the same approach as OMRKF. Perception is abstracted and stored in the feature, space, and object classes. The feature class defines the same knowledge level as the OMRKF. The space class defines a topological map in the middle level and a semantic map in the higher one. The object class middle level includes the object name and function, whereas the top layer defines generic information and relationships among objects; for example, a cup is a type of container.
- *Decision making and planning:* The OMRKF successor, OUR-K (Lim et al., 2011), uses the same structure based on the abductive event calculus planner to reach a hierarchical abstraction of space elements and behaviors. High-level tasks,

TABLE 3 Part 2: Use of ontologies in the navigation domain for perception and categorization (P/C), decision making and planning (DM/P), prediction and monitoring (P/M), reasoning (R), execution (E), communication and coordination (C/C), interaction and design (I/D) and learning (L).

| | Planetary rover scenario (Burroughes and Gao, 2016) | USAR scenario (Chandra and Rocha, 2016) | Autonomous vehicle scenario (Huang et al., 2019) | SAR scenario (Sun et al., 2019a) |
|---|---|---|---|---|
| P/C | - | Pre-processed information from sensors and other intelligent agents | - | Semantic map for recognizing and localizing objects |
| DM/P | Precalculated solutions for likely changes, combined with deliberative techniques | Management of a global map with shared events and real-time tracking of agent locations | Evaluate the safety of the vehicle's surroundings with statistical indicators | Definition of tasks and identification of requisite actions and their properties |
| P/M | Monitor resources and state checks to trigger replanning | - | - | - |
| R | Configure SW elements, update world state, and decide on replanning | Store and distribute information among agents | Apply rules derived from traffic regulations and driving experiences | Evaluate the suitability of the task for the current state |
| E | MAPE-K loop for reconfiguration | - | - | Query task definition to retrieve the corresponding atomic action sequence |
| C/C | Communication requirements between modules | Multirobot cooperation, global map, and selection of suitable candidates for each task | - | - |
| I/D | Explicit service capacities | - | Addresses requirements such as legality by incorporating traffic rules and reasonableness | - |
| L | - | - | - | - |

such as *delivery*, are decomposed into mid-level sub-tasks, such as *go to goal space*, *find object*, or *generate context.* These sub-tasks are further planned as sequences of primitive behaviors, such as *go to* or *recognize object.* The approach used in Diab et al. (2018) based on OUR-K also integrates the three levels of the action class for planning. The planner consults the topological map to determine which object (or robot) occupies which space, and then the semantic map is used to extract object and robot constraints concerning the action. This results in a sequence of actions that may consider contextual information, particularly at the temporal level.

- *Prediction and monitoring:* OUR-K includes rules for navigation monitoring and missing object recognition tasks. However, monitoring is not treated as a distinct process in this framework; instead, it is represented as rules that use several knowledge classes.
- *Reasoning:* Like OMRKF, OUR-K relies on logical inference to associate classes and ontological levels. The bidirectional links between low-level data and high-level knowledge enable both frameworks to fill in missing information, contributing to mission accomplishment.
- *Execution:* OUR-K execution is plan based, where the robot follows the plan and executes a sequence of actions. However, action knowledge is coupled with all other concepts to represent world environments using features such as sensory-motor coordination, object action complex, and affordances. This approach allows robots to perform actions without explicit planning, potentially enabling reactive behavior when necessary (Lim et al., 2011).
- *Interaction and design:* The OUR-K extension presented by Diab et al. (2018) introduces dynamic interaction, focusing on how robotic controllers should adapt to runtime situations. This extension provides information on how a motion planner should handle dynamic forces when manipulating objects, enhancing the framework's capability to deal with real-time interactions.

### 5.3.2 OpenRobots Common Sense Ontology

The OpenRobots Common Sense Ontology (ORO) (Lemaignan et al., 2010) establishes a knowledge processing framework and a common sense ontology tailored for facilitating semantic-rich human–robot interaction environments[19]. This approach focuses on conceptualization but offers flexibility for implementing other cognitive functions simultaneously, such as object recognition, task planning, or reasoning. Cooperation is a crucial aspect in ORO, as it targets human–robot interactions.

ORO ontology is based on OpenCyc. The authors specify in the project Wiki[20] that it shares most of its concepts with the first version of the KnowRob ontology. It includes categories, such as *spatial thing* or *action,* and more concrete concepts, such as *event* or

19 http://kb.openrobots.org/

20 https://www.openrobots.org/wiki/oro-ontology

*book*. The authors evaluated their approach by showing robot objects and asking humans about their properties.

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* The authors use several algorithms, such as common ancestors or calculating the best discriminant to categorize perceptions. Discrimination is an important element in human–robot collaboration. For example, if a user asks the robot to bring the bottle and two bottles are available, it can use its ability to discriminate to select the best option.
- *Reasoning:* ORO provides ontological reasoning, as well as external modules that trigger when an event occurs. External modules are used to provide reactive responses. For example, when a human asks the robot to bring an object, the robot creates an instance of this desire. Statements are stored in different memory profiles, such as long-term and short-term memory. Each profile is characterized by a lifetime that is assigned to the stored facts; when a lifetime ends, the ontology removes the fact.
- *Execution:* The framework integrates CRAM (Beetz et al., 2010) to automatically update the ORO server when an object enters or exits the field of view. Execution is query-based, involving combinations of patterns like *is the bottle on the table?* or filters such as *weight < 150.*
- *Communication and coordination:* ORO is designed as an intelligent blackboard that allows various modules to push or pull knowledge to and from a central repository. This facilitates knowledge sharing among agents. Examples of heterogeneity with shared information include event registration, categorization capabilities, and the existence of different memory profiles.

Each agent has an alternative cognitive model for the other agents with whom it has interacted. When ORO identifies a new agent, it automatically creates a new separate model that can be shared with others. This feature enables the storage and reasoning of different and potentially globally inconsistent models of the world.

### 5.3.3 Ontology for Collaborative Robotics and Adaptation

Ontology for Collaborative Robotics and Adaptation (OCRA)[21], as described by Olivares-Alarcos et al. (2022), is a specialized ontology designed to represent relevant knowledge in collaborative scenarios, facilitating plan adaptation. Based on KnowRob (Tenorth and Beetz, 2009; Beetz et al., 2018) and its upper ontology, DUL, OCRA shares the support of its predecessor for the temporal history of the KB through episodic memories.

This work primarily employs FOL definitions to establish a foundation for reliable, collaborative robots, with the aim of enhancing the interoperability and reusability of terminology within this domain. The ontology serves as a tool for the robot to address competency questions, such as identifying ongoing collaborations, understanding current plans and goals, determining the agents involved, and assessing plans before and after adaptation.

This ontology has been qualitatively validated for human–robot cooperation, sharing the task of filling the compartments of a tray. The main asset of OCRA, from our perspective, is its explicit representation of collaboration requirements, including safety considerations and a measure of risks, with the objective of effective plan adaptation.

The detailed analysis of comparison criteria follows:

- *Decision making and planning:* OCRA uses ontological knowledge for dynamic plan adaptation during runtime. For example, if the robot had a plan to fill a certain compartment and it is not empty, it adapts to fill another empty compartment. The ontology also stores the adaptation triggers.
- *Reasoning:* As stated above, this work focuses on answering competency questions related to agents involved in collaboration, their plans, goals, and the adaptation of plans when required.
- *Communication and coordination:* OCRA focuses on collaboration with humans, providing a mechanism to explain its plan adaptation through the ontology. This collaboration is enabled through coordination in terms of the plan to solve the task. Specifically, the robot changes its plan according to variations in the environment, such as an already-filled compartment.
- *Learning:* While not explicitly implemented in the current work, OCRA mentions future plans to incorporate episodic memories for learning tasks. Examples include modeling the preferences of different users or learning the structure of tasks to generalize to new ones.

### 5.3.4 Intelligent Service Robot Ontology

Intelligent Service Robot Ontology (ISRO) (Chang et al., 2020) introduces an ontology-based model tailored for human–robot interaction. The practical application of this approach is demonstrated through the implementation of a social robot that functions as a medical receptionist in a hospital setting. Specific details about the KB, such as the source files, are not publicly available, so the information provided here is derived from articles about the ontology.

ISRO serves as an abstract knowledge management system designed to comprehend information about agents, encompassing both users and robots, as well as their environment. The ontology establishes connections between user knowledge and the robot's actions and behaviors, incorporating considerations for the spatial and temporal environment. This integration is facilitated through the Artificial Robot Brain Intelligence (ARBI) framework, which includes a task planner, a context reasoner, and a knowledge manager.

ISRO is not limited to a specific domain because it provides a high-level scheme for dynamic generation and management of information. The ontology is divided into four sub-models:

- *User* ontology to store profile information.
- *Robot* ontology to define the robot type, components, and capabilities.

21 https://github.com/albertoOA/know_cra

- *Perception* and *environment* to define objects and their attributes, maps, places, temporal events, and relations such as before, after, etc.
- *Action* ontology to define the actions required to perform a task and the expected events in such situations.

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* The framework uses sensor information to recognize the user involved, their face, gender, and age, and recognize their state, that is, the user's expression, in its application. It also senses the robot's pose to guide the user to the medical department. Additionally, the framework is also prepared to handle information related to spaces and objects that was not defined in the experiment.
- *Decision making and planning:* The ARBI framework uses path planning to move the robot, specifically, to guide patients to the correct medical department. It is based on JAM architecture (Huber, 1999), which defines goals and plans using the belief, desire, and intention (BDI) agent (Chang et al., 2020). The path is produced using semantic spatial knowledge, defining the spaces the robot shall traverse and relationships between spaces such as *connected to.* The semantic path is translated into topological waypoints using knowledge about objects and a map. This path is stored in the KB and shared between robots.
- *Reasoning:* The knowledge manager infers spatial and temporal information, such as the relationships of currently recognized objects and the time intervals between events. It also characterizes users. It uses Prolog to recognize dynamically generated information. However, only information considered important or critical is stored in the KB as static information.

### 5.3.5 Service robot scenario

Ji et al. (2012) provide a flexible framework for service robots using the automatic planner Stanford Research Institute Problem Solver (STRIPS). Specific details about the KB, such as the source files, are not publicly available, so the information provided here is derived from articles about the framework.

The ontology represents two main types of information, environmental description and primitive robot actions, which handle spatial uncertainties of particular objects and the primitive actions that the robot can perform. Actions include four main attributes to comply with STRIPS: precondition, postcondition, input, and result. The planning process is optimized using a recursive back-trace search method and knowledge information to limit the search space. The framework is applied to the Care-O-Bot agent in a scenario in which it shall get a milk box.

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* Symbol grounding is crucial for Ji et al. (2012), as it bridges the gap between abstract planning and actual robot sensing and actuation. For example, a task of moving a table involves moving the robot near the table. The symbol grounding specifies exactly where *"near the table"* is.
- *Decision making and planning:* This framework uses recursive backtracing search for action planning in dynamic environments. The use of semantic maps can improve the search for an object by limiting the search space using semantic inference.
- *Reasoning:* Ji et al. (2012) use reasoning to retrieve information about spatial objects, such as the location of the table where milk is stored. Specific actions, such as *workspace of*, are defined to support reasoning and enable the retrieval of spatial information about an object. For example, a milk box could provide a result of *above the table* or *in the refrigerator*, as it is a perishable product. This framework also provides a likelihood estimation of possible locations of objects.
- *Execution:* Execution uses a central controller to propagate the action to the task planner, the user interface, and the low-level robot modules. Each action is represented as a state machine that is executed in real time. After the action is finished, it updates the models based on the result and the action postcondition.

Table 4 depicts the main aspects of each capability in the social applications reviewed. All capabilities incorporate reasoning using knowledge from the ontology, with the majority also utilizing it for perception and categorization, decision making and planning, and execution. Communication and coordination are only present in two of the five analyzed works. Prediction and monitoring, interaction and design, and learning are the least addressed capabilities across these applications.

## 5.4 Industrial domain

Robots in industrial domains aim to increase efficiency and precision in a variety of tasks. These are often related to manufacturing, production, and other industrial processes. Ontologies in this domain aim to increase flexibility, reduce maintenance, or enhance control and inspection processes.

### 5.4.1 Robot control for Skilled Execution of Tasks

Robot control for Skilled Execution of Tasks (ROSETTA) constitutes an ontology for robots performing manufacturing tasks[22]. Its origins can be traced to the European projects SIARAS, RoSta, and ROSETTA (Stenmark and Malec, 2015). The core of the ontology is mostly focused on robot devices and their skills. It relies on a component called the knowledge integration framework (KIF) to provide services for robotic ontologies and data repositories (Stenmark and Malec, 2015). Note that this should not be confused with the knowledge interchange format, which is a syntax for FOL. KIF acts as an interface to users. They can specify the task by partially ordering the subgoals in an assembly tree; the framework then establishes the action planning and its schedule with limited resources. KIF also provides an execution structure that generates state machines from skill descriptions and their constraints.

Hoebert et al. (2021) use ROSETTA ontology to control the assembly process of a variety of electronic devices. This work focuses on a world model that represents robotic devices and the skills of ROSETTA. It also relies on the

22 https://github.com/jacekmalec/Rosetta_ontology

TABLE 4 Use of ontologies in the social domain for perception and categorization (P/C), decision making and planning (DM/P), prediction and monitoring (P/M), reasoning (R), execution (E), communication and coordination (C/C), interaction and design (I/D) and learning (L).

| | OUR-K (Lim et al., 2011, Diab et al., 2018) | ORO (Lemaignan et al., 2010) | OCRA (Olivares-Alarcos et al., 2022) | ISRO (Chang et al., 2020) | Service robot scenario (Ji et al., 2012) |
|---|---|---|---|---|---|
| P/C | Same approach as OMRKF | Use common ancestors or the best discriminant to categorize | - | Sensor information to recognize the user involved | Abstract planning information |
| DM/P | Abductive event calculus planner for hierarchical abstraction | - | Dynamic plan adaptation during runtime | Path planning, semantic maps | Recursive back-trace searching |
| P/M | Navigation monitoring and missing object recognition tasks | - | - | - | - |
| R | Bidirectional links between low and high levels to fill in missing information | Ontological reasoning combined with external modules triggered when an event occurs | Answering competency questions for collaboration, plans, goals, and adaptations | Infer spatial and temporal information and user characterization | Information about spatial objects |
| E | Action knowledge is coupled with all other concepts of the knowledge of models and features | Query-based integrated CRAM to automatically update the ontology | - | - | Central controller to propagate the action to planner, user interface, and lower levels |
| C/C | - | Intelligent blackboard for knowledge sharing among agents | Collaboration with humans, explainability | - | - |
| I/D | Dynamic interaction, runtime controller adaptation | - | - | - | - |
| L | - | - | Envisioned as future work | - | - |

boundary representation (BREP) ontology (Perzylo et al., 2015) to semantically encode geometric entities. This work uses the ontology to conceptualize objects and their properties in the environment. It defines a product as a hierarchy of sub-assemblies and parts. Requirements are also included to determine the correct automated manufacturing.

Merdan et al. (2019) also employ the ROSETTA ontology to control an industrial assembly process, in particular, a pallet transport system and the control of an industrial robot. In this case, it uses the ROSETTA and BREP ontologies to conceptualize the robotic system (e.g., skills, properties, constraints, etc.), the product model (e.g., parts, geometries, assembly orientation, etc.), and the manufacturing infrastructure (e.g., product, storage, sensors, etc.). Rosetta has been defined in OWL.

The detailed analysis of comparison criteria follows:

*Perception and categorization:* The ROSETTA application by Hoebert et al. (2021) provides an object recognition module that links perception data with geometric features represented in the KB.

*Decision making and planning:* The KIF executor of the ROSETTA ontology serves as the planning mechanism. It transforms an assembly graph into a sequence of operations with preconditions and postconditions, subsequently translated into a task state machine. Both Merdan et al. (2019) and Hoebert et al. (2021) approach decision making as a plan generator, utilizing PDDL. Hoebert et al. (2021) provide a generator that extracts information from the ontology to produce the required PDDL files for planning. Merdan et al. (2019) also translate the semantic model, that is, states and actions, into domain and problem files through templates.

*Reasoning:* ROSETTA reasoning is enabled by the KIF server (Stenmark and Malec, 2015). It allows the user to download and upload libraries with object descriptions, task specifications, and skills. The KIF reasoner assists robot programming by retrieving information about tools, sensors, objects, object properties, etc. Hoebert et al. (2021) take advantage of the ROSETTA ontology to select the individual actions and the equipment required to manufacture a product part.

*Execution:* KIF uses state machines to execute the skills defined in the ROSETTA ontology (Stenmark and Malec, 2015). However, Hoebert et al. (2021) and Merdan et al. (2019) use the execution of atomic actions through PDDL commands. With this approach, as discussed by Hoebert et al. (2021), the missing standardization of the meta-level concepts can cause difficulties when integrating existing ontologies and reduce the scalability of the approach. A comparison of the performance using several PDDL planners is discussed in this article. In terms of scalability, Merdan et al. (2019) use a central database server to support a high-performance integration of multiple homogeneous data sources to integrate the ontology with other knowledge sources.

*Communication and coordination:* The ROSETTA-based framework by Merdan et al. (2019) aims to communicate between robots and entities in the external production environment. Additionally, being a component-based approach, it provides infrastructure for intercomponent communication.

*Interaction and design:* ROSETTA provides an engineering specification of workspace objects, skills, and tasks from libraries (Stenmark and Malec, 2015). It acts as a database for all the information present in the object and in the environment. Hoebert et al. (2021) use these design models to establish and verify the manufacturing process, tailoring the requirements to parameters such as assembly operation type, manufacturing constraints, material used, and piece dimensions.

### 5.4.2 Industrial robotic scenario

Bernardo et al. (2018) define an ontology-based approach for an industrial robotic application focused on inserting up to 56 small pins (sealants) into a harness box terminal specifically tailored for the automotive industry. Specific details about the KB, such as the source files, are not publicly available, so the information provided here is derived from articles about the approach.

The KB is built upon CORA (Prestes et al., 2013) and provides specifications for the robot, the machine vision system, and the tasks required for seamless operation. Task sequences are used to execute the plan, but the machine vision system also uses them to inspect whether the sealants were correctly inserted into the connecting boxes. If not, the system prioritizes applying the sealant in the faulty position(s) in the connecting box.

The detailed analysis of comparison criteria follows:

- *Perception and categorization:* This approach uses vision to inspect robot operations, specifically to verify the correct insertion of sealants in connecting boxes. Image processing is applied for this inspection, although the information is not categorized in the ontology. The framework also utilizes proximity, position, and fiber optic sensors to facilitate robot operation.
- *Decision making and planning:* The framework incorporates visual inspection to validate the precision of the insertion of sealants into the connecting boxes. Additionally, it features a re-planning process that modifies production orders when an error calculus task detects errors. Ontological knowledge is used to establish a sequence of tasks, and in the event of an error, a priority task is immediately added to determine the position of the sealant that requires attention. Once this high-priority task is completed successfully, the robotic system resumes the original production order. The authors define adding this priority task as a re-plan, but we think it corresponds to a planned repair as it returns to the original plan afterward.
- *Reasoning:* This framework uses DL reasoning to acquire valuable data for production and maintenance at the factory level. This reasoning includes information about sensors and actuators and their purposes in various tasks. The knowledge obtained is then utilized to plan and respond to queries, improving the overall decision-making process.

### 5.4.3 Adaptive agents for manufacturing domains

Borgo et al. (2019) extend the DOLCE ontology from a broad perspective. They validated their approach using a real pilot plant, a reconfigurable manufacturing system designed for recycling printed circuit boards (PCBs). Specific details about the KB, such as the source files, are not publicly available, so the information provided here is derived from articles about the ontology. The ontological framework aims to store knowledge about fundamental assumptions and identify the current scenario, including the presence and location of objects, executed actions, responsible agents, changes occurring, etc.

The KB created by this approach seeks to integrate various perspectives within the enterprise, covering intelligent agents, engineering activities, and management activities. The framework is structured in a deliberative layer to synthesize the actions needed to achieve a goal, an executive and monitoring layer to verify the actions, and a mechatronic system that determines the capabilities of the agent.

- *Decision making and planning:* The updated KB uses an abstraction of the device to be controlled, the environment parameters, and the execution constraints to generate a timeline-based planning model. This model provides the atomic operations that a transportation module can perform, depending on its components and the available collaborators. The timeline incorporates temporal flexibility, allowing for relaxed start and end times. The resulting plan represents the potential evolution of the relevant feature. The framework also supports replanning in case of plan execution failure, generating a new plan based on the current state of the mechatronic system.
- *Prediction and monitoring:* Execution is monitored by comparing action outcomes with respect to the expected state of the system and the environment. The monitor process receives signals about the (either positive or negative) outcome of the execution, for example, from the transportation module, and checks whether the actual status of the mechatronic system complies with the plan.
- *Reasoning:* Two types of reasoning are used in this approach. Low-level reasoning infers information about the internal and local contexts of transport modules, identifying system components and available collaborating agents. High-level reasoning utilizes low-level inferences to extract knowledge about the transportation machine's functional capabilities, deducing specific internal and local contexts. This knowledge is then used to generate the plan and its control model.
- *Execution:* The execution is based on a knowledge-based control loop (KBCL) presented in Borgo et al. (2019). This loop facilitates monitoring by dynamically representing the robot's capabilities, internal status, and environmental situation to infer the available functionalities. A reconfiguration phase is activated in case of failure or when new capabilities are added, updating the KB and initiating a new iteration of the overall loop. The executor receives sensor signals and feedback from the transportation module, issuing action commands based on the plan.
- *Communication and coordination:* The framework includes the exchange of information through ontologies, employing commands for sending and receiving interactions with other

TABLE 5 Use of ontologies in the industrial domain for perception and categorization (P/C), decision making and planning (DM/P), prediction and monitoring (P/M), reasoning (R), execution (E), communication and coordination (C/C), interaction and design (I/D) and learning (L).

| | ROSETTA (Stenmark and Malec, 2015; Hoebert et al., 2021; Merdan et al., 2019) | Industrial robotic scenario (Bernardo et al., 2018) | Adaptive agent for manufacturing domains (Borgo et al., 2019) |
|---|---|---|---|
| P/C | Object recognition linking perception data with geometric features | Vision to inspect robot operations and proximity, position, and fiber optic sensors | - |
| DM/P | Transform an assembly graph into a sequence of operations with preconditions and postconditions to create a task-state machine | Replanning process that modifies production orders when an error calculus task detects them | Generate a timeline-based planning model with atomic operations |
| P/M | - | - | Compare action outcomes with respect to the expected status of the system and the environment |
| R | Retrieve information about tools, sensors, objects, object properties, etc. | Reasoning to acquire valuable data for production and maintenance at the factory level | Low-level reasoning for local context and high-level reasoning for the transportation machine's functional capabilities |
| E | State machines to execute skills | - | Reconfiguration phase is activated in case of failure or when new capabilities are added |
| C/C | Communication between robot and external production environment entities and infrastructure for intercomponent communication | - | Commands for sending and receiving interactions with other entities |
| I/D | Design models to establish and verify the manufacturing process, tailoring requirements to parameters | - | Model engineering and management activities |
| L | - | - | - |

entities. The concept of *collaborators* represents relationships between the agent (transport module) and any connected entities, such as other transport modules or machines.

- *Interaction and design:* The ontology also models engineering and management activities. The authors use engineering approaches to identify high-level functions to be executed to reach a given goal, explore the difference between the actual state and the desired state, and isolate the changes to be made. They include information on operand integrity, operand qualities, quality relationships, etc. These concepts make explicit engineering facts that are usually not included in robotic approaches that complement knowledge about robot capacities and contexts.

Table 5 provides a summary of three frameworks applied to industrial scenarios. ROSETTA encompasses all capabilities except prediction, monitoring, and learning. Similarly, the adaptive agent for manufacturing domains includes all capabilities except perception, categorization, and learning. Lastly, the remaining industrial scenario only covers decision making, planning, and reasoning, which were required to be included in the review, and perception and categorization.

## 5.5 Discussion

In this section, we summarize the most relevant results on the reviewed projects as a whole, independent of the discipline in which they were evaluated, as most aim to be generally applicable to any domain. We also discuss other relevant information on the work reviewed, such as the use of temporal information, the encoding language, or the main application and domain.

As far as *perception and categorization* are concerned, most of the projects focus on situation assessment, providing information about the environment to handle the situation and make decisions accordingly. This part is directly related to reasoning because frameworks such as KnowRob (Tenorth and Beetz, 2009; Beetz et al., 2018), SWARMs (Li et al., 2017), or PLM (Antanas et al., 2019) use probability reasoning to generate knowledge about the environment and its affordances.

All works except ORO (Lemaignan et al., 2010) and OCRA (Olivares-Alarcos et al., 2022) use explicit knowledge for *decision making and planning*. These two focus on answering competency questions related to actions or planning for actions in future developments. Geometric and grasp planning are widely used along with task planners to establish a sequence of atomic actions. PDDL is the most widely used task planner. Replanning is also important for some of these articles: TOMASys (Hernández et al., 2018), RTPO (Sun et al., 2019b), FailRecOnt (Diab et al., 2021), industrial application (Bernardo et al., 2018), adaptive manufacturing application (Borgo et al., 2019), and the planetary rover case (Burroughes and Gao, 2016). In these projects, the objective is to maintain the operation in the presence of faults. Lastly, some works also use ontologies to plan in combination with reactive behaviors, such as OUR-K (Lim et al., 2011), ORO (Lemaignan et al., 2010), and the planetary rover application (Burroughes and Gao, 2016).

*Prediction and monitoring* are less present in the reviewed articles. This activity requires a fully operational KB used during runtime and a deeper understanding of the situation and the autonomous robot. Most of the works monitor only the environment, such as KnowRob (Tenorth and Beetz, 2009; Beetz et al., 2018) and OUR-K (Lim et al., 2011). Others only assess the robot state, such as FailRecOnt (Diab et al., 2021), the manipulation application (Borgo et al., 2019), and the planetary rover application (Burroughes and Gao, 2016). Lastly, TOMASys (Hernández et al., 2018) assesses the robot's state and mission status.

*Reasoning* is addressed in all the works. All use ontologies to at least answer queries and verify ontological consistency, as this was an inclusion requirement for the review. Ontological information is used at runtime to recover from failure in some frameworks such as FailRecOnt (Diab et al., 2021), TOMASys (Hernández et al., 2018), and the planetary rover application (Burroughes and Gao, 2016). Reasoners can download additional required information, such as RoboEarth (Tenorth et al., 2012), which uses web and cloud services, or ROSETTA (Stenmark and Malec, 2015), which uses the KIF server to download and upload libraries.

SWARMs (Li et al., 2017) uses a hybrid context reasoner, combining ontological rule-based reasoning with MEBN theory (Laskey, 2008) for probabilistic annotations. PLM (Antanas et al., 2019) uses a probabilistic logic module for grasping, which combines semantic reasoning with object affordances. KnowRob2 (Beetz et al., 2018) takes a further step and provides a hybrid reasoning kernel that enables physics-based reasoning, flexible data structure reasoning, and a detailed robotic agent experience. However, this use comes at a higher cost, as its computation may yield inconsistencies or inefficient reasoning. For this reason, RTPO (Sun et al., 2019b) targets knowledge scalability and efficient reasoning by conducting a study on the performance of real-time reasoning with 52,000 individuals. However, the authors do not describe how this approach affects the planning process.

Not all frameworks use explicit knowledge during *execution;* some of them only perform the action sequence defined in the plan. Others drive its execution to answer competency questions, such as OCRA (Olivares-Alarcos et al., 2022) and KnowRob (Tenorth and Beetz, 2009; Beetz et al., 2018). KnowRob also uses the CRAM (Beetz et al., 2010) executor to update the KB with information about perception and action results, inferring new data to make control decisions at runtime. ORO (Lemaignan et al., 2010) also uses CRAM to update the server when new objects are detected.

Other approaches use state machines, such as ROSETTA (Stenmark and Malec, 2015), or the MAPE-K loop (IBM Corporation, 2005), such as TOMASys (Hernández et al., 2018) and the planetary rover application (Burroughes and Gao, 2016). Similarly, the manufacturing application of Borgo et al. (2019) presents a knowledge-based control loop to combine execution with monitoring.

Other important processes for autonomous systems are *communication and coordination*. All ontological systems can be used easily to answer queries from a human operator. However, this is not sufficient for reliable autonomy; the ontology can provide transparent information to complete the knowledge. SWARMs (Li et al., 2017) enables an abstraction for communication, networking, and information sharing of heterogeneous data. For this, ORO (Lemaignan et al., 2010) uses an intelligent blackboard that allows other modules to push or pull knowledge to a central repository. RoboEarth (Riazuelo et al., 2015) and ROSETTA (Stenmark and Malec, 2015) obtain knowledge from other sources. RoboEarth defines a communication module for uploading and downloading information from the web, and ROSETTA uses the KIF server to download and upload libraries.

Coordination between agents and components of the agent is critical for autonomous operation. ROSETTA provides an infrastructure for intercomponent communication. Component coordination is explicitly addressed in TOMASys (Hernández et al., 2018), the planetary rover from Burroughes and Gao (2016), and the adaptive manufacturing from Borgo et al. (2019). The USAR application from Chandra and Rocha (2016) targets multirobot cooperation for both information sharing and task coordination with other agents. OCRA (Olivares-Alarcos et al., 2022) also handles task coordination and plan adaptation but only concerns human–robot collaboration.

*Interaction and design* are often not explicitly handled during robot operation. However, awareness of interaction allows the robot to step back from action execution and understand the sources of failure. TOMASys (Hernández et al., 2018) provides a metamodel that benefits from the engineering models used during design time. ROSETTA also uses the engineering specification of workspace objects, skills, and tasks as part of the KB. The planetary rover (Burroughes and Gao, 2016) does not explicitly include any information on design or requirements; however, it reasons in terms of the capacity of the services onboard, using some sort of operational requirement to select among alternatives. Similarly, the autonomous vehicle framework (Huang et al., 2019) handles requirements such as legality using traffic rules or reasonableness to support decision making. The adaptive manufacturing application from Borgo et al. (2019) takes a further step in modeling engineering and management activities. Lastly, with regard to dynamical interactions, the OUR-K extension from Diab et al. (2018) includes this knowledge to support the adaptation of robotic controllers. PMK (Diab et al., 2019) represents the interaction as manipulation constraints.

Given the availability of general learning methods, all the processes described above can improve their effectiveness through *learning*. However, this process is less present in the review. Most articles use a form of case-based learning when storing previous successful plans or using episodic memories to recall past situations, such as CORA (Olivares-Alarcos et al., 2022). Other frameworks, such as PMK (Diab et al., 2019) and PLM (Antanas et al., 2019), use learning as part of a situational assessment to categorize perceptual information.

KnowRob provides the most complete learning mechanics. It learns class structures of entities and identifies manipulation places; in addition, it generates generalized models of the physical effects of actions. KnowRob2 (Beetz et al., 2018) makes the learned information available through the Open-EASE knowledge service (Beetz et al., 2015). This service enables any user to upload, access, and analyze episodic memories of robots performing manipulation tasks.

### 5.5.1 Other aspects

Table 6 depicts other aspects of the frameworks studied, such as the languages used, the use of temporal conceptualizations, and the foundational ontologies.

Almost all works use OWL or a combination of OWL and Prolog as the KB language. For planning, most of them

TABLE 6 Summary of other aspects of the framework: concrete encoding languages used, incorporation of temporal conceptualizations, and whether the framework is built upon other works or utilizes an upper-level ontology.

| Framework | Encoding Lang | Temporal | Upper-level Ont |
|---|---|---|---|
| KnowRob (Tenorth and Beetz, 2009; Beetz et al., 2018; Tenorth and Beetz, 2013; Waibel et al., 2011; Tenorth et al., 2012; Crespo et al., 2018; Riazuelo et al., 2015) | OWL, Prolog | Intervals | DUL |
| PMK (Diab et al., 2019) | OWL, Prolog | - | IEEE-1872.2 Std |
| FailRecOnt (Diab et al., 2021; Diab et al., 2020) | OWL | - | DUL |
| PLM (Antanas et al., 2019) | OWL | - | - |
| TOMASys (Hernández et al., 2018; Bozhinoski et al., 2022; Aguado et al., 2021) | OWL | - | OASys |
| OMRKF (Suh et al., 2007) | Prolog | Context | - |
| SWARMs (Li et al., 2017) | OWL | - | PR-OWL |
| RTPO (Sun et al., 2019b) | OWL, Prolog | - | - |
| Planetary Rovers Scenario (Burroughes and Gao, 2016) | PDDL, OWL | Time Concept | - |
| USAR Scenario (Chandra and Rocha, 2016) | OWL | - | - |
| Autonomous Vehicles Scenario (Huang et al., 2019) | Prolog | - | - |
| SAR Scenario (Sun et al., 2019a) | OWL | - | - |
| OUR-K (Lim et al., 2011; Diab et al., 2018) | OWL | Context | OMRKF |
| ORO (Lemaignan et al., 2010) | OWL | - | - |
| OCRA (Olivares-Alarcos et al., 2022) | OWL | Intervals | DUL, KnowRob |
| ISRO (Chang et al., 2020) | OWL, Prolog | Time Concept | OpenCyc |
| Service Robot Scenario (Ji et al., 2012) | OWL, STRIPS | - | - |
| ROSETTA (Stenmark and Malec, 2015; Hoebert et al., 2021; Merdan et al., 2019) | PDDL, OWL | - | - |
| Industrial Robotic Scenario (Bernardo et al., 2018) | OWL | - | CORA |
| Adaptive Agents (Borgo et al., 2019) | OWL | Intervals | DOLCE |

use geometric planners, such as PLM (Antanas et al., 2019), along with task planners. In some works, task planners are custom made, such as the service robot from Ji et al. (2012). However, in general, task planners are based on ontological queries, such as the SAR scenario (Sun et al., 2019a), the industrial application (Bernardo et al., 2018), and the adaptive manufacturing of Borgo et al. (2019). The planetary rover (Burroughes and Gao, 2016) and ROSETTA (Stenmark and Malec, 2015) use PDDL directly, while RTPO (Sun et al., 2019b) and SWARMs (Li et al., 2017) use a custom approach similar to PDDL.

Some studies use temporal conceptualizations to address planning dynamics. Most of them rely on an ordered sequence of actions. However, KnowRob (Tenorth and Beetz, 2009; Beetz et al., 2018), OCRA (Olivares-Alarcos et al., 2022), and the adaptive manufacturing approach use time intervals to recall a time frame in which the action is executed. OMRKF (Suh et al., 2007) and OUR-K (Lim et al., 2011) include time as part of the context ontology, and the planetary rover (Burroughes and Gao, 2016) and ISRO (Chang et al., 2020) conceptualize time in their KBs.

## 6 Research directions and conclusion

Considering the projects surveyed, we believe that ontologies are a valuable asset that supports robot autonomy. The frameworks discussed provide an advance towards mission-level dependability, increasing robot reliability and availability. However, there are both unsolved issues and valuable possibilities for further research in this domain.

An unsolved issue is the limited dissemination and convergence of the different ontological approaches. There is still insufficient reuse of existing ontologies and interchangeability or interoperability of the different realization frameworks. In this direction, KnowRob (Tenorth and Beetz, 2009; Beetz et al.,

2018) is the most documented and impactful project, as it proves its influence on other projects such as ORO (Lemaignan et al., 2010), FailRecOnt (Diab et al., 2021), and OCRA (Olivares-Alarcos et al., 2022). Ontology convergence is always a challenge that is exacerbated by the variety of ontologies, both vertical—the levels of abstraction—and horizontal—the domains of application. However, efforts must be made to ensure this harmonization, given the fact that robots are not isolated entities but are always part of systems-of-systems that share specific forms of knowledge (Sanz et al., 2021).

We believe that future engineering-grade, knowledge-driven autonomous robot software platforms should also be capable of providing three characteristics: explainability, reusability, and scalability. In the articles under review:

- *Explainability* is enabled in most cases—at least in a shallow form—as most frameworks include query/answer interfaces to provide information on robot operation. However, this explainability is tightly bound to the capability of the human user, who is often a robot operator, to fully grasp the explanation. More broad-spectrum, human-aligned ontologies are needed to support a wider spectrum of users.
- *Reusability* is intrinsically present, given the explicitness of declarative knowledge and the sharing of common backgrounds, some of which are built on previous works or upper-layer ontologies. Efforts should made to harmonize and integrate conceptualizations to effectively reuse ontologies—especially in heterogeneous systems.
- *Scalability* and information handling are also partially addressed in some of the realizations, such as reasoning in RTPO (Sun et al., 2019b) and KnowRob (Tenorth and Beetz, 2009; Beetz et al., 2018). However, the problems of scalability remain, such as when dealing with complex systems-of-systems, when addressing geographical or temporal extensive missions, or when knowledge-based collaboration is required, as is the case of cognitive multirobot systems or human–robot teams.

Therefore, we propose a shared concern in future works in this field to improve the capabilities of robots and contribute to the community in these three aspects. There are already some initiatives, such as CRAM from KnowRob authors, a software toolbox for the design, implementation, and deployment of manipulation activities (Beetz et al., 2010). This toolbox supports planning, beliefs, and KnowRob reasoners and has been used in other frameworks such as ORO (Lemaignan et al., 2010). The same authors provide the package *rosprolog*, a bidirectional interface between SWI-Prolog and ROS, to make this logic language accessible to the main robotics middleware. However, these tools are tailored for KnowRob-based environments and are sometimes not well documented or maintained.

The conclusions we extract from this survey are that most of the work is focused on categorization, decision making, and planning. However, *monitoring and coordination* are critical processes with respect to robot control, especially for robust and reliable operation. These kinds of processes are difficult to assess using reusable implementations because of the enormous variability between applications, context, and intervening agents. We propose the use of *explicit engineering models* to facilitate these processes. They are already partially included in TOMASys (Hernández et al., 2018), ROSETTA (Stenmark and Malec, 2015), the planetary rover scenario (Burroughes and Gao, 2016), and the adaptive manufacturing application from Borgo et al. (2019). We believe that the combination of runtime information with design knowledge can be used during decision making and to drive adaptation to bridge the differences between the expected results of robotic users and the actual robot performance.

## 7 Conclusion

In this article, we have presented an overview of projects that use ontologies to enable robot autonomy. We have systematically searched for works that propose or extend an ontology and use that knowledge to select, adapt, or plan robot actions. We have compared approaches in terms of how the processes that support autonomous operation use and update the supporting ontology. We also briefly discussed applications, languages, and time representation in those works.

Conceptualization provides robots with a path to understanding, which is a powerful tool for achieving robustness and resiliency. We have found that most frameworks do not include explicit engineering knowledge about the robot system itself. Information about robot components and their interaction, design requirements, and alternatives enables both explainability and self-adaptation. Robots are far from meeting user and owner expectations, especially in terms of dependability, efficiency, and efficacy. We have found that task and goal specifications usually lack explicit knowledge of the user-expected performance. This is necessary information to create robots that a user can trust because trust depends on the reliable provision of some user-expected result. For example, this information could include user phenomenological aspects, required safety levels, or energy thresholds that make a task unprofitable.

In conclusion, we believe that explicit knowledge can support autonomous robot operations in unstructured environments. There are still open issues with respect to reliability, safety, and explainability to meet the expectations of researchers and the industry. However, the steps taken by these projects in a variety of domains to enhance autonomy using ontologies have proven how promising this approach is. A strong effort in convergence and harmonization is needed, but this is maximally difficult because the necessary concepts are situated in the realm of the always elusive cognitive robot mind.

## Data availability statement

The raw data supporting the conclusion of this article will be made available by the authors, without undue reservation.

## Author contributions

EA: conceptualization, methodology, writing–original draft, writing–review and editing. VG: conceptualization, writing–review and editing. MH: methodology, writing–review and editing. CR: conceptualization, formal analysis, writing–review and editing.

RS: conceptualization, formal analysis, project administration, writing–review and editing.

## Funding

The author(s) declare that financial support was received for the research, authorship, and/or publication of this article. This work was partially supported by the ROBOMINERS project with funding from the European Union's Horizon 2020 research and innovation programme (Grant Agreement No. 820971), by the CORESENSE project with funding from the European Union's Horizon Europe research and innovation programme (Grant Agreement No. 101070254), and by a grant from the *Programa Propio* supported by the Universidad Politécnica de Madrid.

## Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## Publisher's note

## References

Abbott, R. J. (1990). Resourceful systems for fault tolerance, reliability, and safety. *ACM Comput. Surv.* 22, 35–68. doi:10.1145/78949.78951

Aguado, E., Gómez, V., Hernando, M., Rossi, C., and Sanz, R. (2024). "Category theory for autonomous robots: the marathon 2 use case," in *Robot 2023: sixth iberian robotics conference*. Editors L. Marques, C. Santos, J. L. Lima, D. Tardioli, and M. Ferre (Cham: Springer Nature Switzerland), 39–52.

Aguado, E., Milosevic, Z., Hernández, C., Sanz, R., Garzon, M., Bozhinoski, D., et al. (2021). Functional self-awareness and metacontrol for underwater robot autonomy. *Sensors* 21, 1210–1228. doi:10.3390/s21041210

Antanas, L., Moreno, P., Neumann, M., de Figueiredo, R. P., Kersting, K., Santos-Victor, J., et al. (2019). Semantic and geometric reasoning for robotic grasping: a probabilistic logic approach. *Aut. Robots* 43, 1393–1418. doi:10.1007/s10514-018-9784-8

Arp, R., Smith, B., and Spear, A. D. (2015) *Building ontologies with basic formal ontology*. The MIT Press. doi:10.7551/mitpress/9780262527811.001.0001

Avižienis, A., Laprie, J.-C., and Randell, B. (2004). "Dependability and its threats: a taxonomy," in *Building the information society*. Editor R. Jacquart (Boston, MA: Springer US), 91–120.

Balakirsky, S., Schlenoff, C., Fiorini, S., Redfield, S., Barreto, M., Nakawala, H., et al. (2017). "Towards a robot task ontology standard," in *Proceedings of the manufacturing science and engineering conference (MSEC)* (Los Angeles, CA: US).

Beer, J. M., Fisk, A. D., and Rogers, W. A. (2014). Toward a framework for levels of robot autonomy in human-robot interaction. *J. Hum.-Robot Interact.* 3, 74–99. doi:10.5898/JHRI.3.2.Beer

Beetz, M., Beßler, D., Haidu, A., Pomarlan, M., Bozcuoğlu, A. K., and Bartels, G. (2018). "Know rob 2.0 — a 2nd generation knowledge processing framework for cognition-enabled robotic agents," in *2018 IEEE international conference on robotics and automation (ICRA)*, 512–519. doi:10.1109/ICRA.2018.8460964

Beetz, M., Mösenlechner, L., and Tenorth, M. (2010). "CRAM — a Cognitive Robot Abstract Machine for everyday manipulation in human environments," in *2010 IEEE/RSJ international conference on intelligent robots and systems*, 1012–1017. doi:10.1109/IROS.2010.5650146

Beetz, M., Tenorth, M., and Winkler, J. (2015). "Open-EASE – a knowledge processing service for robots and robotics/ai researchers," in *IEEE international conference on robotics and automation (ICRA) (seattle, Washington, USA). Finalist for the best cognitive robotics paper award.*

Bermejo-Alonso, J., Sanz, R., Rodríguez, M., and Hernández, C. (2010). "An ontology–based approach for autonomous systems' description and engineering," in *Knowledge-based and intelligent information and engineering systems*. Editors R. Setchi, I. Jordanov, R. J. Howlett, and L. C. Jain (Berlin, Heidelberg: Springer Berlin Heidelberg), 522–531.

Bernardo, R., Farinha, R., and Gonçalves, P. J. S. (2018). "Knowledge and tasks representation for an industrial robotic application," in *Robot 2017: third iberian robotics conference*. Editors A. Ollero, A. Sanfeliu, L. Montano, N. Lau, and C. Cardeira (Cham: Springer International Publishing), 441–451.

Beßler, D., Porzel, R., Pomarlan, M., Vyas, A., Höffner, S., Beetz, M., et al. (2021). "Foundations of the socio-physical model of activities (SOMA) for autonomous robotic agents," in *Formal ontology in information systems* (IOS Press), 159–174.

Borgo, S., Cesta, A., Orlandini, A., and Umbrico, A. (2019). Knowledge-based adaptive agents for manufacturing domains. *Eng. Comput.* 35, 755–779. doi:10.1007/s00366-018-0630-6

Bozhinoski, D., Oviedo, M. G., Garcia, N. H., Deshpande, H., van der Hoorn, G., Tjerngren, J., et al. (2022). MROS: runtime adaptation for robot control architectures. *Adv. Robot.* 36, 502–518. doi:10.1080/01691864.2022.2039761

Brachman, R. (2002). Systems that know what they're doing. *IEEE Intell. Syst.* 17, 67–71. doi:10.1109/MIS.2002.1134363

Brown, C. E., Pease, A., and Urban, J. (2023). "Translating SUMO-K to higher-order set theory," in *Frontiers of combining systems (FroCoS), to appear*.

Bunge, M. (1977) *Treatise on basic philosophy: volume 3: ontology I: the furniture of the world*. Boston, MA: Reidel.

Burroughes, G., and Gao, Y. (2016). Ontology-based self-reconfiguring guidance, navigation, and control for planetary rovers. *J. Aerosp. Inf. Syst.* 13, 316–328. doi:10.2514/1.I010378

Chandra, R., and Rocha, R. P. (2016). "Knowledge-based framework for human-robots collaborative context awareness in usar missions," in *2016 international conference on autonomous robot systems and competitions (ICARSC)*, 335–340. doi:10.1109/ICARSC.2016.50

Chang, D. S., Cho, G. H., and Choi, Y. S. (2020). "Ontology-based knowledge model for human-robot interactive services," in *Proceedings of the 35th annual ACM symposium on applied computing* (New York, NY, USA: Association for Computing Machinery), 2029–2038. doi:10.1145/3341105.3373977

Cornejo-Lupa, M., Ticona-Herrera, R., Cardinale, Y., and Barrios-Aranibar, D. (2020). A survey of ontologies for simultaneous localization and mapping in mobile robots. *ACM Comput. Surv.* 53, 1–26. doi:10.1145/3408316

Crespo, J., Barber, R., Mozos, O. M., BeBler, D., and Beetz, M. (2018). "Reasoning systems for semantic navigation in mobile robots," in *2018 IEEE/RSJ international conference on intelligent robots and systems*, 5654–5659. doi:10.1109/IROS.2018.8594271

Diab, M., Akbari, A., and Rosell, J. (2018). "An ontology framework for physics-based manipulation planning," in *Robot 2017: third iberian robotics conference*. Editors A. Ollero, A. Sanfeliu, L. Montano, N. Lau, and C. Cardeira (Cham: Springer International Publishing), 452–464.

Diab, M., Akbari, A., Ud Din, M., and Rosell, J. (2019). PMK—a knowledge processing framework for autonomous robotics perception and manipulation. *Sensors* 19, 1166. doi:10.3390/s19051166

Diab, M., Pomarlan, M., Beßler, D., Akbari, A., Rosell, J., Bateman, J., et al. (2020). "An ontology for failure interpretation in automated planning and execution," in *Robot 2019: fourth iberian robotics conference*. Editors M. F. Silva, J. Luís Lima, L. P. Reis, A. Sanfeliu, and D. Tardioli (Cham: Springer International Publishing), 381–390.

Diab, M., Pomarlan, M., Borgo, S., Beßler, D., Rosell, J., Bateman, J., et al. (2021). "FailRecOnt – An Ontology-Based Framework for Failure Interpretation and Recovery in Planning and Execution," in *Proceedings of the Joint Ontology Workshops co-located with the Bolzano Summer of Knowledge (BOSK 2021)*, Virtual & Bozen-Bolzano, Italy, September 13–17, 2021, CEUR-WS.org, 2021.

Fiorini, S. R., Bermejo-Alonso, J., Gonçalves, P., Pignaton de Freitas, E., Olivares Alarcos, A., Olszewska, J. I., et al. (2017). A suite of ontologies for robotics and automation [industrial activities]. *IEEE Robotics Automation Mag.* 24, 8–11. doi:10.1109/MRA.2016.2645444

Gangemi, A., Guarino, N., Masolo, C., Oltramari, A., Oltramari, R., and Schneider, L. (2002). Sweetening ontologies with DOLCE. In *Proceedings of the 13th European conference on knowledge engineering and knowledge management (EKAW)*, 166–181.

Gayathri, R., and Uma, V. (2018). Ontology based knowledge representation technique, domain modeling languages and planners for robotic path planning: a survey. *ICT Express* 4, 69–74. doi:10.1016/j.icte.2018.04.008

Gayathri, R., and Uma, V. (2019). A review of description logic-based techniques for robot task planning. *Stud. Comput. Intell.* 771, 101–107. doi:10.1007/978-981-10-8797-4_11

Guarino, N. (1998) *Formal ontology in information systems: proceedings of the 1st international conference june 6-8, 1998, trento, Italy*. Netherlands: IOS Press.

Guiochet, J., Machin, M., and Waeselynck, H. (2017). Safety-critical advanced robots: a survey. *Robotics Aut. Syst.* 94, 43–52. doi:10.1016/j.robot.2017.04.004

Hepp, M., Bachlechner, D., and Siorpaes, K. (2006). "Ontowiki: community-driven ontology engineering and ontology usage based on wikis," in *Proceedings of the 2006 international symposium on wikis* (New York, NY, USA: Association for Computing Machinery), 143–144. doi:10.1145/1149453.1149487

Hernández, C., Bermejo-Alonso, J., and Sanz, R. (2018). A self-adaptation framework based on functional knowledge for augmented autonomy in robots. *Integr. Computer-Aided Eng.* 25, 157–172. doi:10.3233/ICA-180565

Hoebert, T., Lepuschitz, W., Vincze, M., and Merdan, M. (2021). Knowledge-driven framework for industrial robotic systems. *J. Intelligent Manuf.* doi:10.1007/s10845-021-01826-8

Huang, L., Liang, H., Yu, B., Li, B., and Zhu, H. (2019). "Ontology-based driving scene modeling, situation assessment and decision making for autonomous vehicles," in *2019 4th asia-pacific conference on intelligent robot systems (ACIRS)*, 57–62. doi:10.1109/ACIRS.2019.8935984

Huber, M. J. (1999). "Jam: a bdi-theoretic mobile agent architecture," in *Proceedings of the third annual conference on autonomous agents* (New York, NY, USA: Association for Computing Machinery), 236–243. doi:10.1145/301136.301202

IBM Corporation (2005). An architectural blueprint for autonomic computing. *Tech. Rep.* White Paper. doi:10.1109/ICAC.2004.1301340

Ieee, S. A. (2015). IEEE standard ontologies for robotics and automation. *IEEE Std* 1872-2015, 1–60doi. doi:10.1109/IEEESTD.2015.7084073

Ieee, S. A. (2022). "IEEE standard for autonomous robotics (AuR) ontology," in *Standard IEEE std 1872* (IEEE), 2–2021. doi:10.1109/IEEESTD.2022.9774339

Ji, Z., Qiu, R., Noyvirt, A., Soroka, A., Packianather, M., Setchi, R., et al. (2012). "Towards automated task planning for service robots using semantic knowledge representation," in *IEEE 10th international conference on industrial informatics*, 1194–1201. doi:10.1109/INDIN.2012.6301131

Lamy, J.-B. (2017). Owlready: ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. *Artif. Intell. Med.* 80, 11–28. doi:10.1016/j.artmed.2017.07.002

Langley, P., Laird, J. E., and Rogers, S. (2009). Cognitive architectures: research issues and challenges. *Cognitive Syst. Res.* 10, 141–160. doi:10.1016/j.cogsys.2006.07.004

Laskey, K. B. (2008). MEBN: a language for first-order Bayesian knowledge bases. *Artif. Intell.* 172, 140–178. doi:10.1016/j.artint.2007.09.006

Lemaignan, S., Ros, R., Mösenlechner, L., Alami, R., and Beetz, M. (2010). "Oro, a knowledge management platform for cognitive architectures in robotics," in *2010 IEEE/RSJ international conference on intelligent robots and systems*, 3548–3553. doi:10.1109/IROS.2010.5649547

Lenat, D. B. (1995). Cyc: a large-scale investment in knowledge infrastructure. *Commun. ACM* 38, 33–38. doi:10.1145/219717.219745

Li, X., Bilbao, S., Martín-Wanton, T., Bastos, J., and Rodriguez, J. (2017). SWARMs ontology: a common information model for the cooperation of underwater robots. *Sensors* 17, 569. doi:10.3390/s17030569

Lim, G. H., Suh, I. H., and Suh, H. (2011). Ontology-based unified robot knowledge for service robots in indoor environments. *IEEE Trans. Syst. Man, Cybern. - Part A Syst. Humans* 41, 492–509. doi:10.1109/TSMCA.2010.2076404

Lukyanenko, R., Storey, V. C., and Pastor, O. (2021). Foundations of information technology based on bunge's systemist philosophy of reality. *Softw. Syst. Model.* 20, 921–938. doi:10.1007/s10270-021-00862-5

Manzoor, S., Rocha, Y., Joo, S.-H., Bae, S.-H., Kim, E.-J., Joo, K.-J., et al. (2021). Ontology-based knowledge representation in robotic systems: a survey oriented toward applications. *Appl. Sci. Switz.* 11, 4324. doi:10.3390/app11104324

Mascardi, V., Cordì, V., and Rosso, P. (2006) *A comparison of upper ontologies (technical report DISI-TR-06-21)*. Tech. rep., Dipartimento di Informatica e Scienze dell'Informazione (DISI) and Universidad Politécnica de Valencia.

Merdan, M., Hoebert, T., List, E., and Lepuschitz, W. (2019). Knowledge-based cyber-physical systems for assembly automation. *Prod. Manuf. Res.* 7, 223–254. doi:10.1080/21693277.2019.1618746

Musen, M. A. (2015). The protégé project: a look back and a look forward. *AI Matters* 1, 4–12. doi:10.1145/2757001.2757003

Niles, I., and Pease, A. (2001). "Toward a standard upper ontology," in *Proceedings of the 2nd international conference on formal ontology in information systems (FOIS-2001)*. Editors C. Welty, and B. Smith, 2–9.

Olivares-Alarcos, A., Beßler, D., Khamis, A., Goncalves, P., Habib, M., Bermejo-Alonso, J., et al. (2019). A review and comparison of ontology-based approaches to robot autonomy. *Knowl. Eng. Rev.* 34, e29. doi:10.1017/S0269888919000237

Olivares-Alarcos, A., Foix, S., Borgo, S., and Alenyà, G. (2022). OCRA – an ontology for collaborative robotics and adaptation. *Comput. Industry* 138, 103627. doi:10.1016/j.compind.2022.103627

Page, M. J., McKenzie, J. E., Bossuyt, P. M., Boutron, I., Hoffmann, T. C., Mulrow, C. D., et al. (2021). The PRISMA 2020 statement: an updated guideline for reporting systematic reviews. *BMJ* 372, n71. doi:10.1136/bmj.n71

Pease, A. (2011) *Ontology: a practical guide*. Angwin, CA: Articulate Software Press.

Perzylo, A., Somani, N., Rickert, M., and Knoll, A. (2015). "An ontology for cad data and geometric constraints as a link between product models and semantic robot task descriptions," in *2015 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, 4197–4203. doi:10.1109/IROS.2015.7353971

Prestes, E., Carbonera, J. L., Rama FioriniJorge, S. M. V. A., Abel, M., Madhavan, R., et al. (2013). Towards a core ontology for robotics and automation. *Robotics Aut. Syst.* 61, 1193–1204. doi:10.1016/j.robot.2013.04.005

Riazuelo, L., Tenorth, M., Di Marco, D., Salas, M., Gálvez-López, D., Mösenlechner, L., et al. (2015). Roboearth semantic mapping: a cloud enabled knowledge-based approach. *IEEE Trans. Automation Sci. Eng.* 12, 432–443. doi:10.1109/TASE.2014.2377791

Russell, S. J., and Norvig, P. (2021) *Artificial Intelligence: a modern approach*. 4 edn. England: Pearson.

Sanz, R., Alarcon, I., Segarra, M., Clavijo, J. A., and de Antonio, A. (1999). Progressive domain focalization in intelligent control systems. *Control Eng. Pract.* 7, 665–671. doi:10.1016/S0967-0661(99)00012-X

Sanz, R., Bermejo, J., Rodríguez, M., and Aguado, E. (2021). The role of knowledge in cyber-physical systems of systems. *TASK Q.* 25, 355–373.

Sanz, R., Matia, F., and Galan, S. (2000). "Fridges, elephants, and the meaning of autonomy and intelligence," in *IEEE international symposium on intelligent control - proceedings (patras, Greece)*, 217–222. doi:10.1109/isic.2000.882926

SEBoK Editorial Board (2023). The guide to the systems engineering body of knowledge (*SEBoK*), Available at: www.sebokwiki.org. (Accessed August 2023).

Shapiro, S. C. (2003) *Encyclopedia of cognitive science*. Macmillan Publishers Ltd, 671–680. Knowledge Representation.

Staab, S., and Studer, R. (2009) *Handbook on ontologies*. 2nd edn. Incorporated: Springer Publishing Company.

Stenmark, M., and Malec, J. (2015). Knowledge-based instruction of manipulation tasks for industrial robotics. *Robotics Computer-Integrated Manuf. Special Issue Knowl. Driven Robotics Manuf.* 33, 56–67. doi:10.1016/j.rcim.2014.07.004

Suh, I. H., Lim, G. H., Hwang, W., Suh, H., Choi, J.-H., and Park, Y.-T. (2007). "Ontology-based multi-layered robot knowledge framework (omrkf) for robot intelligence," in *2007 IEEE/RSJ international conference on intelligent robots and systems*, 429–436. doi:10.1109/IROS.2007.4399082

Sun, X., Zhang, Y., and Chen, J. (2019a). High-level smart decision making of a robot based on ontology in a search and rescue scenario. *Future Internet* 11, 230. doi:10.3390/fi11110230

Sun, X., Zhang, Y., and Chen, J. (2019b). RTPO: a domain knowledge base for robot task planning. *Electronics* 8, 1105. doi:10.3390/electronics8101105

Tenorth, M., and Beetz, M. (2009). Knowrob — knowledge processing for autonomous personal robots. in *2009 IEEE/RSJ international conference on intelligent robots and systems*, 4261–4266. doi:10.1109/IROS.2009.5354602

Tenorth, M., and Beetz, M. (2013). Knowrob: a knowledge processing infrastructure for cognition-enabled robots. *Int. J. Robotics Res.* 32, 566–590. doi:10.1177/0278364913481635

Tenorth, M., Clifford Perzylo, A., Lafrenz, R., and Beetz, M. (2012). "The roboearth language: representing and exchanging knowledge about actions, objects, and environments," in *2012 IEEE international conference on robotics and automation*, 1284–1289. doi:10.1109/ICRA.2012.6224812

Vernon, D. (2014) *Artificial cognitive systems: a primer*. The MIT Press.

W3C OWL Working Group (2012) *OWL 2 web ontology language document overview*. World Wide Web Consortium. Tech. rep.

Waibel, M., Beetz, M., Civera, J., D'Andrea, R., Elfring, J., Gálvez-López, D., et al. (2011). Roboearth. *IEEE Robotics Automation Mag.* 18, 69–82. doi:10.1109/MRA.2011.941632

Wand, Y., and Weber, R. (1993). On the ontological expressiveness of information systems analysis and design grammars. *Inf. Syst. J.* 3, 217–237. doi:10.1111/j.1365-2575.1993.tb00127.x

Frontiers in Robotics and AI

TYPE Original Research
PUBLISHED 02 August 2024
DOI 10.3389/frobt.2024.1363281

Check for updates

OPEN ACCESS

EDITED BY
Ivano Malavolta,
VU Amsterdam, Netherlands

REVIEWED BY
Federico Ciccozzi,
Mälardalen University, Sweden
Michel Albonico,
Federal University of Technology, Brazil

*CORRESPONDENCE
Argentina Ortega,
argentina.ortega@uni-bremen.de

†These authors have contributed equally to this work and share first authorship

RECEIVED 30 December 2023
ACCEPTED 04 July 2024
PUBLISHED 02 August 2024

CITATION
Ortega A, Parra S, Schneider S and Hochgeschwender N (2024), Composable and executable scenarios for simulation-based testing of mobile robots. *Front. Robot. AI* 11:1363281.
doi: 10.3389/frobt.2024.1363281

# Composable and executable scenarios for simulation-based testing of mobile robots

Argentina Ortega[1,2*†], Samuel Parra[3†], Sven Schneider[3] and Nico Hochgeschwender[1]

[1]SECORO Group, Department of Computer Science, University of Bremen, Bremen, Germany, [2]Intelligent Software Systems Engineering Lab (ISSELab), Department of Computer Science, Ruhr University Bochum, Bochum, Germany, [3]Institute for AI and Autonomous Systems, Department of Computer Science, Hochschule Bonn-Rhein-Sieg, Sankt Augustin, Germany

Few mobile robot developers already test their software on simulated robots in virtual environments or sceneries. However, the majority still shy away from simulation-based test campaigns because it remains challenging to specify and execute suitable testing *scenarios*, that is, models of the environment *and* the robots' tasks. Through developer interviews, we identified that managing the enormous variability of testing scenarios is a major barrier to the application of simulation-based testing in robotics. Furthermore, traditional CAD or 3D-modelling tools such as SolidWorks, 3ds Max, or Blender are not suitable for specifying sceneries that vary significantly and serve different testing objectives. For some testing campaigns, it is required that the scenery replicates the dynamic (e.g., opening doors) and static features of real-world environments, whereas for others, simplified scenery is sufficient. Similarly, the task and mission specifications used for simulation-based testing range from simple point-to-point navigation tasks to more elaborate tasks that require advanced deliberation and decision-making. We propose the concept of *composable and executable scenarios* and associated tooling to support developers in specifying, reusing, and executing scenarios for the simulation-based testing of robotic systems. Our approach differs from traditional approaches in that it offers a means of creating scenarios that allow the addition of new semantics (e.g., dynamic elements such as doors or varying task specifications) to existing models without altering them. Thus, we can systematically construct richer scenarios that remain manageable. We evaluated our approach in a small simulation-based testing campaign, with scenarios defined around the navigation stack of a mobile robot. The scenarios gradually increased in complexity, composing new features into the scenery of previous scenarios. Our evaluation demonstrated how our approach can facilitate the reuse of models and revealed the presence of errors in the configuration of the publicly available navigation stack of our SUT, which had gone unnoticed despite its frequent use.

KEYWORDS

**verification and validation, software testing, simulation-based testing, scenario-based testing, robot software engineering, model-based development, mobile robot, navigation**

## 1 Introduction

The responsible deployment of autonomous mobile robots in everyday environments (e.g., warehouses, hospitals, and museums) relies on extensive testing to ensure that robots achieve their expected performance and can cope with failures to avoid safety risks during their operational lifetime. The two major types of testing–in simulations and the real world–have complementary properties. The former allows robots to be exposed to a wide range of situations early in the development cycle at a limited cost (Sotiropoulos et al., 2017; Timperley et al., 2018), whereas the latter offers more realistic situations and failure cases in later stages of the development cycle. Often, developers forego simulation-based testing, even if they are aware of its benefits, and expose their robots exclusively to the real world (Ortega et al., 2022). This often requires more time to set up than a simulator, and reduces coverage because it is difficult to change the real world, for example, by deliberately injecting failure-inducing situations. Both approaches can be employed for black- and white-box testing at various levels of abstraction (e.g., system vs. component tests).

In our previous study (Parra et al., 2023), we obtained a better understanding of why robot software engineers opt out of simulation-based testing by conducting in-depth interviews with 14 domain experts in the field of mobile robot navigation in indoor environments. One key insight we identified is that creating scenery or virtual environments in which simulated robots are deployed and tested remains challenging for developers. The use of traditional Computer Aided Design (CAD) and three-dimensional (3D) modelling tools is time-consuming because they require an additional skill set. To make simulation-based testing more attractive to developers, we designed and implemented two domain-specific languages (DSLs), namely, the FloorPlan DSL and the Variation DSL. We demonstrated how these DSLs enable developers to specify and automatically generate varying yet testable environments, and how testing robots in different simulated worlds overcomes the false sense of confidence (Hauer et al., 2020). Furthermore, our tooling helped discover real-world dormant bugs in the well-known ROS navigation stack (Parra et al., 2023).

Even though providing tool support for specifying testing scenery is a crucial element to make simulation-based testing of robot software more attractive, it is not sufficient. Additional models that express robot tasks and missions, robot capabilities, interactions among agents, and temporal evolution of actions and events are required to make simulation-based testing campaigns more realistic. In the field of autonomous driving, these models are known as *scenarios* (Tang et al., 2023). In the context of this study, we broadly define scenarios entailing both *mission-relevant* and *mission-plausible* information. On the one hand, by mission-relevant information we refer to, among others, the environment and its dynamics, time and events, objects (e.g., rooms) and subjects (e.g., human operators), and their potential behaviour. On the other hand, the mission-plausible information describes acceptance criteria that enable the verification and validation of the robotic requirements.

As we will show in this article, the interviews revealed that testing scenarios are characterized by a large amount of variability that results in varying, heterogeneous models expressing all too often implicitly in an ad-hoc way the robots' environment and task, as well as the developers' testing objectives, means to execute scenario models in simulations, and hints on how to collect and interpret test results. Therefore, simulation-based robot testing remains limited to carefully designed testing campaigns in which developers have control over a few scenario features and parameters, such as the type of robot task and the characteristics of the environment. Thus, reusing scenarios in the context of other testing campaigns is limited and a major barrier to achieving a higher level of test automation.

To improve this situation, we propose the concept of *composable and executable scenarios* and developed associated tooling to support robot software engineers in specifying, reusing, and executing scenarios in (semi-)automated simulation-based testing campaigns of robotic systems. To this end, we revisit and further analyse the corpus obtained by in-depth interviews conducted and briefly presented in our previous work (Parra et al., 2023), with the objective of deriving a domain model of scenario-based testing through simulation in robotics. As a result, we identified the common and variable parts of simulation-based testing and represented them in a feature model for scenarios of mobile robots. These features are selected to design or reuse the composable models needed for a particular scenario. Our composable modelling approach enables the addition of new semantics to existing scenarios, without altering them. This approach allows the development of new extensions and tools to support new use cases for the FloorPlan models previously introduced. To summarize, our contributions are:

- a domain model with common and variable features for simulation-based testing scenarios of mobile robots,
- a composable modelling approach to specify and execute scenarios,
- a dynamic-objects extension to the FloorPlan DSL that allows to model scenery objects and their locations using JSON-LD and facilitates the reuse and exploitation of environment models in simulation-based test design and semi-automated model-based scenario generation,
- three gazebo plugins that exploit the scenery information and integrate with the simulation to define the (initial) poses of objects (initial pose plugin), and actuate their joints on a time (dynamic joint plugin) or distance basis (distance-to-trigger plugin),
- a proof-of-concept tool to exploit scenery information and features from the FloorPlan DSL models to generate task specifications,
- and we demonstrate how one can use our approach to systematically run a simulation-based testing campaign with scenarios of varying complexity.

## 2 Domain analysis

To develop our composable and modelling approach, we perform a domain analysis using the corpus we obtained in (Parra et al., 2023). In this section, we describe the methodology we followed and the domain model we derived based on our insights.

## 2.1 Methodology

Semi-structured interviews were conducted (Hove and Anda, 2005), which involved interviews with specific questions to set the theme for the discussion, but allowed for exploration of the topic through open-ended questions. This allowed for a flexibly guided discussion. We designed a questionnaire that covered experts' experience with software for mobile wheeled robots (specifically indoor navigation stacks for mapping, motion planning, and obstacle avoidance), their real-world challenges, and the challenges of simulation in the context of testing. An internal pre-study was conducted to improve the questionnaire. We then recruited participants for the study by reaching out to professionals in academia and the industry. A list of potential candidates was obtained from our professional network.

We conducted 14 interviews with a pool of experts with diverse academic and professional backgrounds, as well as multiple years of experience in the field. The table summarizing the interviewee's demographics can be found in the Supplementary Material of this paper. The interviews were conducted through an online meeting, recorded, and transcribed into protocols, which were later separated into fragments. Interview participants signed a written informed consent and their participation was voluntary. All the interview data were anonymized by IDs, which only participants have and can use to withdraw their participation.

To analyse the fragments, we used qualitative coding (Saldaña, 2021), which consists of assigning one or multiple "codes" to the fragments[1]. For instance, the fragment "*One metric to measure map quality is to see how many tasks can be completed with it. How useful it is to solve certain kind of tasks.*" has the codes *Environment Representation* and *Performance*. We selected a list of codes before the start of the coding and allowed for expansion if necessary. We performed two rounds of coding: an initial round and a review in a second round. We used 37 themes to code the interviews. The distribution of references per individual code is available in the Supplementary Material of this paper. Once all the fragments were coded, patterns were identified in the data.

## 2.2 Domain model

Based on the identified patterns, we derived a domain model for scenario-based testing in robotics in the form of a feature model, as illustrated in Figure 1. Here, we employed a standard feature model notation (Kang et al., 1990) to express the mandatory and optional features of the scenarios. Our scenario domain model is composed of four main features: the *System under Test (SUT)*, which is tested, assessed, and evaluated in the context of varying scenarios; the *testing objective* of the scenario; the *scenery*, which is a description of the environment or virtual world in which the SUT is embedded; and a specification of the *mission* that the SUT is expected to execute. In the following paragraphs, we explain the domain model by referencing some representative quotes from the interviews shown in Table 1 as excerpts E1-E10. Note that the abstract feature model in Figure 1 is not exhaustive; its abstraction levels were chosen to allow the addition of new features (e.g., planning and scheduling features under the mission feature) in future extensions.

The interviews revealed that the roles and activities of the developer influenced the type and scope of the tests they performed during the development process. Most interviewees considered themselves to be integrators and/or robot application developers in various fields, such as logistics or healthcare, where robots (cf. E4) perform missions characterized by navigation tasks and where an action is associated with one or more waypoints (e.g., the waypoints of racks to be visited in a logistic mission) (cf. E5).

Stakeholders mentioned a number of testing goals that influence their design decisions for their tests. Developers often build systems by composing readily available components (cf. E2), some of which are well-tested software packages developed and maintained by a third party such as an open-source community. Often, the components chosen for building the system are highly tailorable, which requires tuning parameters for an optimized performance (cf. E2). For integrators, interest in testing focuses on the capabilities and performance of the integrated system. These tests verified that all components were integrated correctly and validated the parameter values, and often involve multiple components and algorithms, instead of focusing on a single component. They are also more likely to require execution in a robotic simulator, and therefore, a scenery. However, other testing objectives such as safety and robustness, functional correctness, etc., are also present (cf. E1, E2, E3).

Generally, simulation is seen as a valuable tool, but it can be challenging to fully utilize it. The setup process for simulation execution can be a time-consuming task, meaning that smaller developer teams often opt to perform tests exclusively in the real world. One participant states, "*[using a simulator] depends on whether creating the simulation was going to add a long term benefit. In most cases, the answer was not. It required too much effort*".

Creating or specifying the scenery, or environment model, for the simulation is often mentioned as one of the big challenges. Although simulating a scenario requires several types of models, such as robotic platform models, simulation capabilities for sensors and actuators, and a model of the environment (the scenery), the former two are often provided by their respective manufacturers, but the latter must often be created by application developers. We identified that scenery can be broadly divided into two main features, namely, the environment dynamics and the environmental features that are present. If the target environment contains particular features such as double doors, rails, or columns, it is useful to include them in the simulated environment to observe the behaviour of the system when it is exposed. (cf. E7) The simulation environment can also be application dependent. One participant stated, "*You need to describe the elements you want to be robust against. You do not want to describe all aspects.*" Modelling the 3D scenery for simulation is often the reason developers refrained from employing simulators. The experts see the modelling task as time-consuming, as one participant asserts "*The environment is very large and modelling is time-consuming*". The effort necessary to model the environment depends on the scale and level of granularity that

1 The codes and interview fragments are available online at https://github.com/secorolab/floorplan-dsl-interviews.

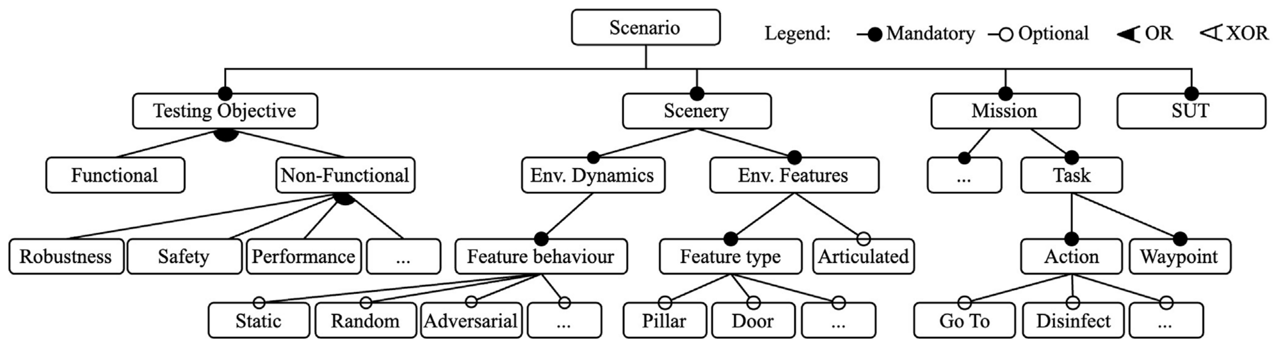


FIGURE 1
Abstract scenario feature model derived from the codes in participant interviews.

**TABLE 1 Representative interview excerpts and their relation to top-level features of our domain model.**

| Feature | ID | Excerpt |
|---|---|---|
| Testing Objective | E1 | *I would measure [robustness] by trying challenging scenarios, maybe introducing different test. Create a simple environment for a test, such as a static environment, and make it more complex by adding dynamic obstacles* |
| | E2 | *Another of my projects was regarding robot collisions, so a lot of tests also focused on that. The test were performed to optimize parameters and try to make the stack work* |
| | E3 | *There is an impact, if there is a discrepancy between what you see in the real world and the map this will degrade the performance* |
| SUT | E4 | *My goal was to integrate the platform and the navigation stack, so my tests had that objective* |
| Mission | E5 | *We also deployed robots in industrial spaces, and there the setting was an industrial warehouse with many racks* |
| Scenery | E6 | *We try to replicate the real environment, but is very limited. We have only a static world with the walls and objects that make the environment* |
| | E7 | *Another challenge are dynamic obstacles, and understanding the environment. i.e., understanding that a piece of furniture is not fixed but also that it does not move often* |
| | E8 | *Interacting with objects such as doors and chairs is also challenging* |
| | E9 | *Lighting is one of the main issues if you wanna use VSLAM map, model of human agents are difficult in our standard simulator. I also wanted to model doors that open and close. It is interesting to simulate if the robot can get through certain doors* |
| | E10 | *When dynamic come into play, i.e., everything that makes the map to change significantly, this can lead to localization and navigation failures* |

the test requires. The same participant states: "*Depending on the application, I would also like to see the models have either a lot of detail or be very simple*". This refers to the levels of granularity, i.e., how much correspondence there is between the real world entity and its model (Hauer et al., 2020). Because modelling using traditional tools is time-consuming, and the 3D modelling tools have a steep learning curve, when developers create scenery models for simulation these tend to be low in granularity; i.e., they mostly consist of a set of walls.

The experts are also interested in re-creating environments for simulation. Two-thirds of the participants have tried to replicate the real world in a simulation. When real-world environments are re-created, most of the features of the environment were not modelled. The result is a scenery that consist of a set of walls that replicates the geometric shape of the original environment, with some cases adding objects such as furniture.

In summary, developers usually test their robots in scenery resembling static and primitive environmental features, such as walls and rooms (cf. E6) of the known and unknown target environment. These simple sceneries are incrementally enriched through additional and not necessarily dynamic features such as static obstacles (cf. E2) until the point of including dynamic elements, such as other agents, obstacles, and lighting conditions (cf. E9, E10), and actuated environmental features such as drawers, windows, and doors (cf. E8) to gain confidence in the tests. One can infer that developers of real-world robot applications would like to further exploit simulation-based testing of robotic systems, but that the current tools for specifying and executing actual test scenarios are limited. They do not allow the creation of scenery models in a flexible and incremental manner in which new concepts and associated semantics (e.g., a door and how it moves) can be added without altering the existing model.

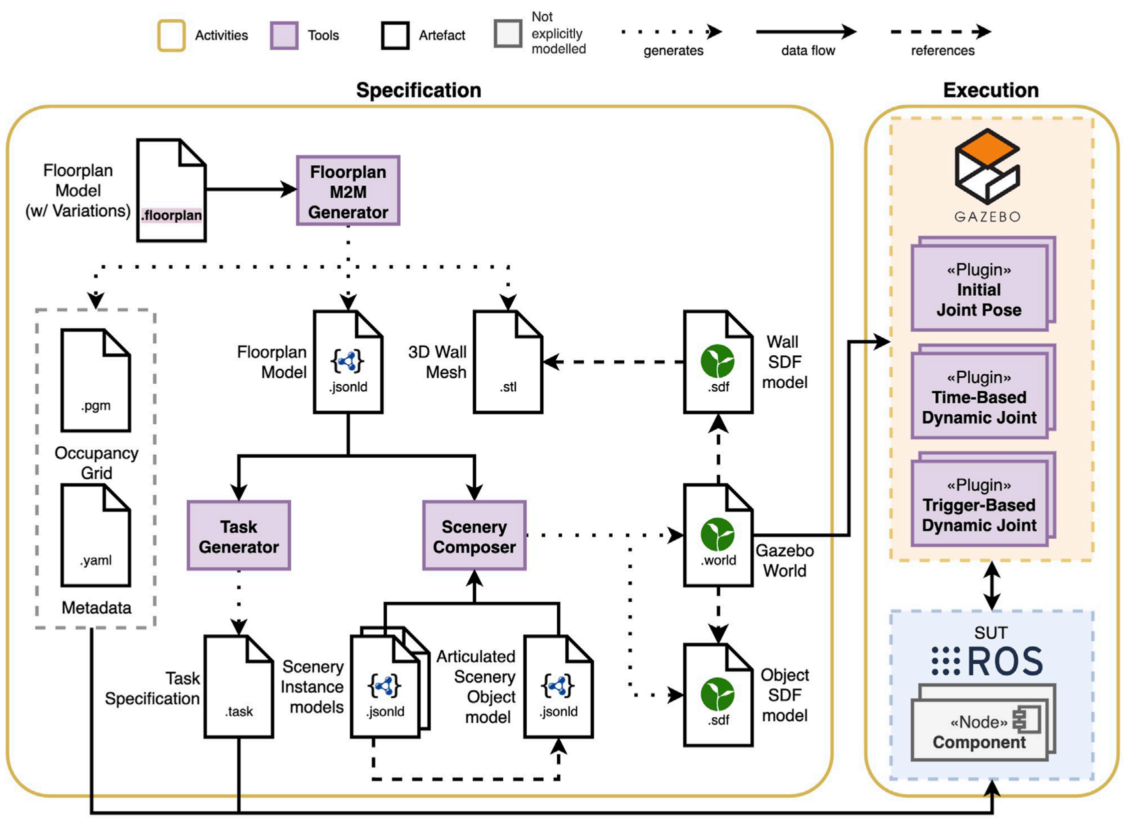


FIGURE 2
Composable and executable scenario pipeline showing the models, the tools discussed in this paper (purple) and the execution artefacts resulting from their composition and transformation.

## 3 Composable and executable scenarios

One of the goals of our composable and executable scenario approach is to enable engineers in specifying, reusing and executing scenarios in simulation. Before specifying a scenario, examining the design space of the scenarios (cf. Figure 1) with the testing objective and the application requirements in mind results in the choice of the scenario features. Next, the corresponding models for those features must be specified (or potentially reused from other scenarios). Finally, these models are composed and transformed into software artefacts that can be used in simulation. The remainder of this section looks at these three main steps in more detail. Figure 2 shows an overview of the scenario specification and execution activities and tooling.

### 3.1 Scenario design

Let us start by examining the design dimensions of a scenario, and how the design decisions have an impact in the effectiveness and efficiency of the scenario.

The first design dimension to be determined should be the testing objective, as all the other design decisions for the scenario will depend on the objective and the scope of the test. There are numerous objectives that engineers can have in mind when designing a test, among others, examples include:

- Performance: Optimization of the configuration parameters for a particular behaviour in an exploratory way, identification of the effect of changes in the performance, or measurement of the efficiency of a given configuration.
- Robustness: Determination of the reaction or handling unexpected environmental changes, or calculating through experimentation the failure rates of the hardware or software components.
- Safety: Validation of conformance to internal or external standards, or identification of hazards and failures in the robot capabilities.
- Functional: Validation of the correctness of a component by validating that its performance is within the required or specified tolerances.

The task specification–*what* the SUT should do during a test–is one of the inputs that must be defined or adapted to support the test objective of the scenario and the scope of the SUT. Usually, the task for a fully-integrated system is determined by the application, but given the environmental complexity different variations of tasks, e.g., in scale or choice of locations, can be chosen to fit the SUT, and the scope and objective of the scenario. For instance, functional tests require tasks that are designed to be successfully completed in nominal conditions. Note that in unstructured environments, even for nominal conditions, larger scale or more complex tasks can reveal unexpected behaviours, due to the increasing amount of time the robot interacts with its surroundings. For other types of tests like safety, engineers can specify tasks informed by the test objective, e.g., choosing actions or constraints that could produce a failure. In short, the task specification is a test input that describes the workload the robot is expected to execute.

The scenery features include the features to be modelled in the floor plan, including the types of objects the robot interacts with, and the behaviour of those features, particularly if they are dynamic. The choices depend on the objective of the test (e.g., narrow hallways, moving obstacles), the SUT (e.g., minimum width of doorways for it to pass through). Functional tests require scenery that represents expected operating conditions, while sceneries that are used or designed for robustness tests must include features that represent invalid inputs or stressful conditions, such as dynamic obstacles. For testing the conformance to safety features, the scenery design should focus on including features that introduce hazards, increasing the risk of a critical failure. Consider the following examples of sceneries for different types of tests:

- Functional navigation tests: To assess if the robot is able to complete navigation tasks of varying complexity, validate if the robot is able to reach the target poses. A passing test means the robot reaches all the waypoints in its task. The complexity of the mission is determined by several factors: how many waypoints must be visited, the distance between waypoints, and the reachability from one waypoint to another. The distance between waypoints can be chosen from an existing scenery, or a new scenery can be created to test in larger environments. The reachability is constrained by the geometry of the space and by the pose of obstacles. The ideal simulation scenery for testing localization components includes many static features in the floor plan, but a reduction of these features can also increase the complexity.
- Robustness testing for obstacle avoidance: To validate how well the local planner adapts to sudden changes in the scenery, the robot can be tasked to perform navigation tasks of varying complexity in scenery where there are dynamic changes. The changes should be sufficient to trigger a re-planning of the planned path by the local planner, but not enough to worsen the localization performance. For this objective, it is sufficient that the changes occur at random times, where the complexity increases with the frequency and number of changes. This type of simulation scenery could also be used to perform a functional test of the trajectory planning component.
- Safety conformance by negative testing: One way to test the conformance to safety and functional requirements is to design a test case where the robot is expected to fail. Rather than random changes to the scenery, the changes can be adversarial to the robot, where especial conditions trigger changes in the simulation scenery. For instance, when the robot is less than 1 m away from the door, the door closes.

The test oracle–the mechanism to compare the expected result of a test with the observed output–is directly related to all the scenario features. Although they are usually derived from or influenced by the application requirements, they must be defined taking into account the objective of the test (e.g., what metrics to observe), the task (e.g., waypoints, specified tolerances or constraints), SUT (e.g., configuration) and scenery (e.g., free space, objects, obstacles). For example, the specified tolerance for the performance of a component can be the difference between a passed and failed test.

## 3.2 Scenario specification

A scenario specification is a composition of multiple models, with each individual model targeting a different dimension of the scenario. To form a complete specification of the scenario, we use composable models. A model is "composable" if the entities of the model can refer to each other via identifiers. New entities from a new model are composed by referencing the entities in the existing models. A model can now be a domain-specific artefact, that with composition can create a full specification. In previous work (Schneider et al., 2023), the use of JSON-LD as a representation for composable models was introduced, as well as many metamodels that are used in the scenario specifications presented in this work.

Composabiliy enables a modular approach for the re-use of models in multiple specifications. This is used in our approach as a way to systematically and gradually introduce complexity, and allows the creation of scenarios that are more challenging based on simpler scenarios without modifying the existing models. For instance, a scenario can start with a static scenery that just contains walls, and a new scenario reuses the floorplan model and composes a new simulation scenery with obstacles in the environment. The next scenario reuses these specifications and composes some dynamic behaviour to the obstacles, and so forth.

The floor plan model is the starting point of the scenery specification. We make a distinction between "user-facing" or "front-end" models and "machine-readable" or "back-end" models. Models written using DSLs are "front-end" models, as they are written using syntax and semantics meant for human understanding. On the other hand, the composable models are meant to be created and understood by computers. While it is possible to create these models by hand, it is complex and error-prone. A better approach is to transform the "front-end" models into "back-end" models.

The FloorPlan DSL, introduced in our previous work (Parra et al., 2023), is the base of the front-end environment specification. It enables developers to describe concrete indoor environments using a pseudo-code-like representation. The language is implemented with TextX (Dejanović et al., 2017), a Python-based language workbench for defining the metamodel and language syntax. The language is declarative and designed to be easy to understand. Using keywords such as Space, Column, or Entryway followed by an identifier, common elements of an indoor environment can be specified and referred to.

To compose objects, such as doors with hinges or elevator doors, into the scenery, their models must be specified in a composable way. We do so based on the kinematic chain metamodel described in previous work (Schneider et al., 2023) and represent them also in JSON-LD (as there is no front-end language currently available). Two types of scenery models are needed to represent an object: An object model describes their geometry and instance models that specifies the pose of its articulated joint using a selection of frames of reference.

## 3.3 Scenario execution

Executing the scenario involves the composition and transformation of the models into software artefacts for their execution in simulation. The execution of an indoor scenario requires multiple software artefacts: a simulation scenery (3D mesh) representing the walls of the environment for the simulator, an occupancy grid map representative of the environment for the navigation stack, and a task to complete in a format supported by the system.

Although the majority of the artefacts generated by the tools are simulator-independent, engineers will also need simulator-specific artefacts to run the tests; our current version of the tooling supports the generation of the artefacts required by the Robot Operating System (ROS) and the Gazebo simulator[2]. Previously, when the FloorPlan M2M Generator was executed, it used the manually-specified FloorPlan model to generate the occupancy grid maps and the 3D meshes that would be referred to by manually-specified Gazebo models and worlds. These sceneries could only represent static environments.

In this paper, we introduce an extension to the FloorPlan M2M Generator that generates a Composable FloorPlan model (represented in JSON-LD) to enable its composition with other scenery models. Now, it also generates the Composable FloorPlan models, where each entity has an identifier that other entities can refer to. The transformation and composition process, illustrated in Figure 3, links all the entities from the different models in a singular graph though their identifiers. Using this graph, we can make queries about environmental concepts and features, and generate artefacts or new models. The transformation and composition engine is implemented by using rdflib[3] to query the singular graph, and, similar to textX generators, filling out a jinja template[4] for the corresponding artefact. In our case, it allows us to model and compose objects into the scenery, and to define their dynamic behaviour.

Scenery composition refers to the composition of the static scenery models with the dynamic scenery objects to generate the simulation scenery. We developed the Scenery Composer tool to add articulated dynamic objects to the static simulation scenery from the FloorPlan models. The composable models enable the specification of the objects and their location in the environment, and the Scenery Composer tool creates a single scenery model that refers to all the different models together. The format of this model will depend on the simulator. In the case of Gazebo, this format is known as the Simulation Description Format (SDF)[5], and referred to as the "world" file. At the time of writing, the Scenery Composer targets only Gazebo, and generates all the required models in SDF.

At runtime, three Gazebo plugins are responsible for the behaviour of the dynamic scenery objects. The three plugins require that the scenery object is articulated, i.e., has at least a revolute or prismatic joint. All plugins are able to set a joint pose, but differ when and how the changes occur. The Initial Joint Pose plugin is used to assign to a joint a position at the start of the simulation, which will fixed throughout the entire run. In contrast, the Time-Based Dynamic Joint plugin can change the position of the joint at specified time stamps; for example, closing a door after 30 s of simulation time. The Trigger-Based Dynamic Joint plugin can change the joint position from an initial state to an end state if the robot ever gets closer than a specified distance.

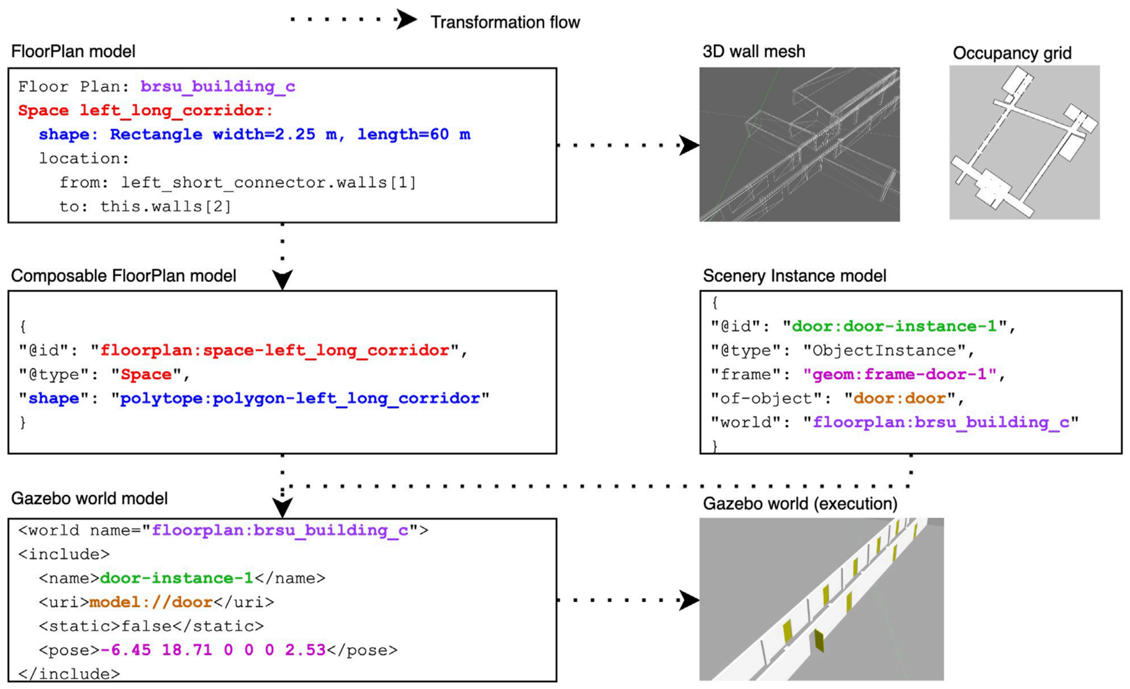
Finally, a task specification can be generated using the composable approach based on the scenery models. We opted for generating the task specification in our approach to take advantage of existing mission and task DSLs that meet application and domain-specific requirements for the specification, which would be hard to generalize. As a proof-of-concept, our tool generates navigation tasks tailored to our SUT, but can be easily adapted to generate specifications in other formats. We refer to the tool as the Task Generator, which exploits the geometric information in the FloorPlan model to generate a series of waypoints that form a smaller contour based on the inset of each space in the environment. The tool uses the FloorPlan composable models to extract the free space information, and a configuration file to determine the distance between the room contour and the inset contour. The current prototype generates a list of waypoints using YAML syntax, which is used by the navigation stack.

## 4 Evaluation

To evaluate our approach, we designed three scenarios that demonstrate how to exploit different properties of the scenery for a given test objective. For each scenario we describe the test objective, i.e., the motivation for the test and chosen from the examples in Section 3.1, and the features selected for the test based on the test objective. Note that although testing is context-dependent and the scenarios discussed here take into account a specific System Under Test (SUT), our focus is on *how* to test robot software, not the design or development of a particular navigation algorithm or robot platform. In particular, the goal of the scenarios described in this section is to exemplify how one would use composable scenarios to execute tests to validate the software of a mobile robot. Thus, we mainly focus on the models being used and/or designed as described in Section 3.2.

Our SUT consists of the KELO Robile platform, a mobile robot platform with four active wheels and a 2D laser for navigation. The robot is 0.466 m wide and 0.699 m long. Its software is based on the Robot Operating System (ROS), and uses its navigation stack[6]. This includes the default map_server; move_base to send navigation goals to the robot; the Navfn global planner, the Dynamic Window Approach (DWA) local planner with global and local costmaps; and the Adaptive Monte Carlo Localization (AMCL) algorithm for its pose estimation.

2 https://gazebosim.org/
3 https://rdflib.readthedocs.io/
4 https://jinja.palletsprojects.com/
5 http://sdformat.org/
6 http://wiki.ros.org/navigation



```
Floor Plan: brsu_building_c
Space left_long_corridor:
  shape: Rectangle width=2.25 m, length=60 m
  location:
    from: left_short_connector.walls[1]
    to: this.walls[2]
```

```
{
"@id": "floorplan:space-left_long_corridor",
"@type": "Space",
"shape": "polytope:polygon-left_long_corridor"
}
```

```
{
"@id": "door:door-instance-1",
"@type": "ObjectInstance",
"frame": "geom:frame-door-1",
"of-object": "door:door",
"world": "floorplan:brsu_building_c"
}
```

```
<world name="floorplan:brsu_building_c">
<include>
  <name>door-instance-1</name>
  <uri>model://door</uri>
  <static>false</static>
  <pose>-6.45 18.71 0 0 0 2.53</pose>
</include>
```

FIGURE 3
Scenery models are composed using their identifiers and then transformed into execution artefacts.

The scenery where the tests are performed is the ground floor of a university building and was modelled using the FloorPlan DSL, as detailed in (Parra et al., 2023). The recreation of the building's corridors and rooms was achieved by performing measurements of occupancy grid maps captured in the real world. Then the measurements were used to specify the FloorPlan DSL model, and used to generate the occupancy grid and wall mesh.

The test scenarios also exploit the composable models presented in this paper to generate the tasks to be performed and the variations in scenery in which they are executed. First, using the Composable FloorPlan Model, the Task Generator creates task specifications for each room and hallway in the scenarios. Second, we specify an Articulated Scenery Object model that describes the door geometry and joints. Finally, in each scenario we compose this door model with specific scenery instance models into a Gazebo world model that supports the scenario's testing objective, as will be detailed later.

## 4.1 Scenario 1: functional testing for navigation

### 4.1.1 Testing objective

The objective of this functional test is to ensure that all components of the navigation stack are correctly integrated and configured[7]. The goal is for the robot to successfully navigate from the starting position to a series of waypoints. In addition, for this paper, we chose to observe the localization component as it is one of the components in the navigation stack that relies on the correct integration with the other components. In this scenario, a successful navigation test means that the robot meets the following functional requirements: (a) reaches all the waypoints, (b) the localization error does not exceed 0.35 m, and (c) the confidence level of the localization component is 95% at minimum.

### 4.1.2 Models

Given that this is a functional test, we select the features shown in Figure 4 and treat this scenario as a way to obtain a consistent and reproducible baseline. Therefore, the scenery we chose is a static environment with realistic features. The corridor illustrated in Figure 4 is 60 m long, and is part of the FloorPlan model of the university building. It has several features to aid in localization: doorways and doors, columns, and intersections.

The Gazebo world model with static doors was generated by the Scenery Composer using the Composable FloorPlan Model, the articulated scenery door model, and the scenery instance model for the 17 doors. The instance models allow us to specify the initial pose for each door joint, which was set as "closed" (0 rad) for this scenario. Even though the door models are articulated, the doors will remain static throughout the execution.

The navigation task the robot executes was generated by the Task Generator using the Composable FloorPlan model. Its specification consists of a list of waypoints which must be visited in strict order. The generated task specification was manually updated to close the circuit (i.e., five Go To actions in a sequence, including the return to the first waypoint). In this scenario, the waypoints are the four

7 In the scope of this paper, this is a simplified version of an integration test for the navigation stack. As such, we consider it an example of how to validate the functionality of a subsystem within a system-of-systems.

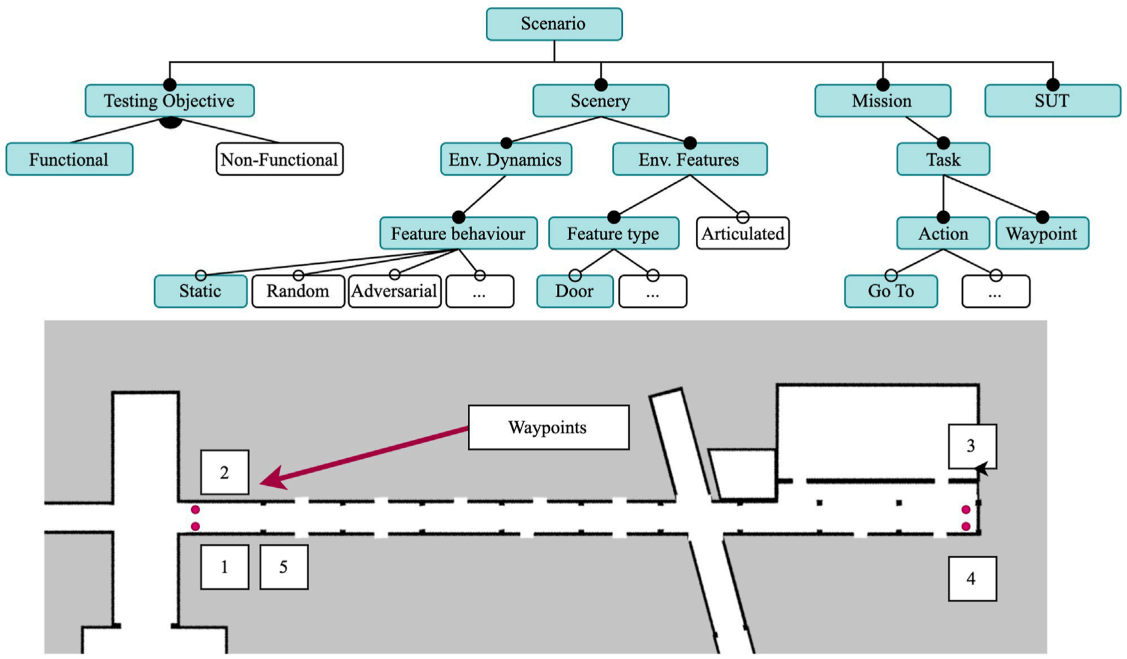


FIGURE 4
Feature model for Scenario 1, with a static scenery and realistic environmental features (e.g., doors). The task consists of five Go To actions for the four waypoints (red dots) for a mobile robot (SUT).

corners of a corridor as shown in Figure 4, and are located at a constant distance of 70 cm from the walls.

### 4.1.3 Test oracle

A passing test must meet the functional requirements listed in Section 4.1.1. For the experiment, we hold the hypothesis that the robot will be able to localize itself successfully, as the environment is static and has numerous features for correcting the estimation. To measure the localization performance, we rely on two metrics: the error $e$ is computed as the difference between the pose estimation and the ground truth pose $p_{gt}$, and the standard deviation of the particle cloud, which we use to validate the confidence level. To compute the latter, we obtain the number of times when the difference between the ground truth and the particle cloud is not statistically significant (i.e., not larger than $2\sigma$). The confidence level is the proportion of those that fulfil Eq. 1,

$$\overline{P}_c - 2\sigma \leq p_{gt} \leq \overline{P}_c + 2\sigma \quad (1)$$

where $\sigma$ is the standard deviation of the particle cloud, and $\overline{P_c}$ is the mean of the particles.

## 4.2 Scenario 2: robustness testing for obstacle avoidance

### 4.2.1 Test objective

The objective of this test is focused on the robustness of the navigation stack, particularly on the ability of the robot to avoid obstacles in a dynamic environment under stressful conditions. This scenario uses a highly dynamic environment where there is a higher risk of collision with moving doors. The task is now performed in a dynamic version of the scenery, where the doors open and close at random intervals. The challenge for the robot is twofold: first, it has to adapt its plan depending on the status of the doors, which change frequently and randomly. Second, they must avoid colliding with the doors, even if they change state when the robot is very close.

### 4.2.2 Models

Using Scenario 1 as a starting point, we increase the complexity to test the robustness of the obstacle avoidance component by making the scenery dynamic, as shown in Figure 5. The scenery for this scenario is mostly the same as the one in Scenario 1, the only difference being the addition of dynamic doors that will open and close at random intervals using the Time-Based Dynamic Joint plugin. Thus, the world file for Scenario 1 is almost identical to the world file for Scenario 2 with the only difference being the use of the plugin at runtime. All positions for all the doors remain the same. Although the task specification is the same, its execution is more complex due to a more challenging environment. When closed, the doors are aligned with the walls of the corridor, keeping the way clear for the robot. When open, the doors are perpendicular to the corridor walls, and partially block the corridor as the doors open towards the inside of the corridor.

The execution models remain mostly the same, the only difference is observed in the world file of the simulation. Dynamic models require a plugin (with its configuration) for them to have a behaviour during the simulation run. For each door, we instantiate a Time-Based Dynamic Joint plugin, which takes as a parameter a JSON file with a sequence of key-frames. The key-frames contain the simulation time at which the joint should move to a particular

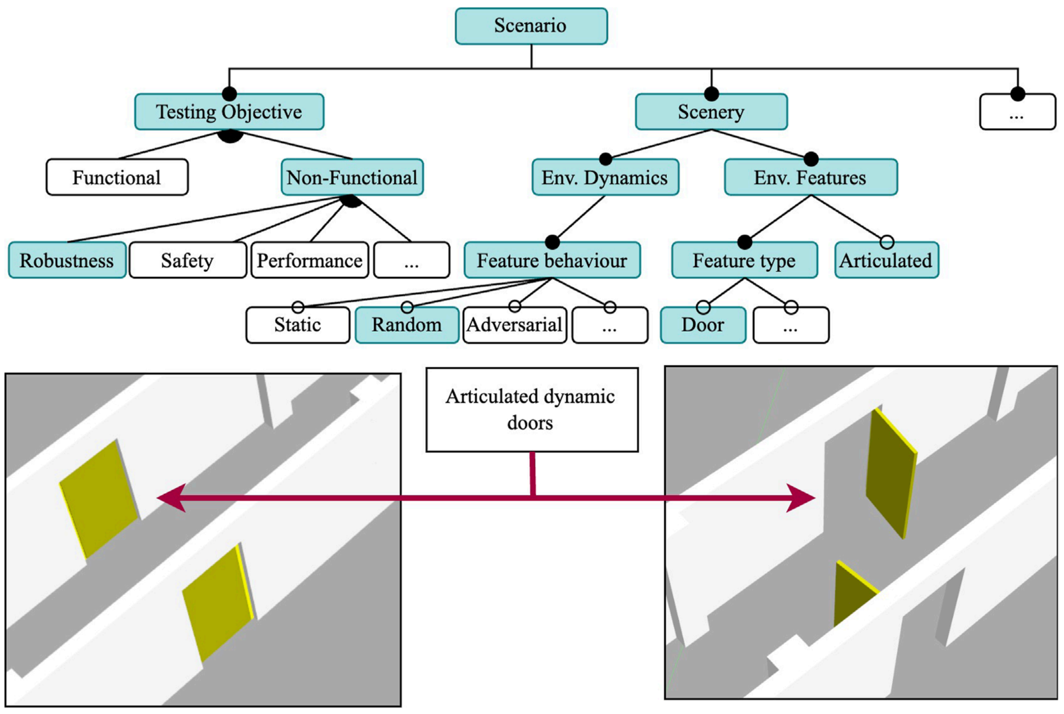


FIGURE 5
Feature model for Scenario 2: A robustness test in a dynamic environment and randomly actuated doors. The mission and SUT features are the same as in Scenario 1. Door models are articulated.

pose. The doors are closed at 0 rad, and open at 1.7 rad. Each door is independent and has a unique opening/closing sequence, with a randomly assigned state change and simulation time. The sequence for each door remains constant throughout the five runs.

### 4.2.3 Test oracle

A successful scenario test run is one where the robot successfully avoids all collisions. To measure the effectiveness of the robot to avoid collisions, we look for the smallest distance to an object with respect to the robot's centre. Given its rectangular shape, a collision occurs when the distance of an object $d_o$ to the centre of the robot is $d_o(x) \leq 0.35$ and $d_o(y) \leq 0.2$. For the simplicity, we discuss $d_o$ using the radius of the robot from its corner, i.e., $\sqrt{0.233^2 + 0.35^2} = 0.42$ although we validate there are no crashes by checking $d_o(x)$ and $d_o(y)$.

## 4.3 Scenario 3: safety conformance in adversarial environment with task variability

### 4.3.1 Test objective

Motivated by the near crashes in Scenario 2 (discussed in Sect. 5.2), the objective in this scenario is to validate the conformance of the SUT to one of its safety requirements, namely, that the robot respects the minimum acceptable distance to obstacles and maintains a safety buffer of $d_s = 0.2$. More concretely, we validate the ability of the navigation stack to complete a navigation task in an adversarial environment where doors close as the robot approaches them. This means the robot should respond to the environmental changes and conform to its minimum safety distance of 0.2 m, i.e., $min(d_o(x)) \geq 0.55$ and $min(d_o(y)) \geq 0.433$.

In this scenario, we also introduce four variations in the task scale that gradually increase the scenario complexity, creating one sub-scenario for each task. The sub-scenarios were executed 5 times each, which amounts to 20 runs in the simulator for this scenario.

## 4.4 Models

To test the safety requirements of the SUT, we select the adversarial behaviour for the dynamic elements of the scenery for this scenario, as can be seen in Figure 6. Following the incremental approach, this scenario will reuse most of the execution models of Scenario 2, but specify an adversarial behaviour for a subset of the doors in the environment. Although we use the same campus re-creation from the previous two environments, our tests are now performed in three rooms rather than one corridor.

Instead of random events as in Scenario 2, in this scenery the two dynamic doors are now triggered when the robot is within a distance threshold from the door using the trigger-based dynamic joint plugin. All other doors in the scenery are set to open with the initial joint pose plugin, and remain static throughout the execution. The relevant doors and their behaviour are illustrated in Figure 7. The trigger-based dynamic joint pluginis configured by describing an initial position, a final position, and a minimum distance for the transition trigger. The force and speed of the closing door is not parameterized. When the robot comes to a distance of 1.3 m or closer, an event to close the door triggers. This distance was

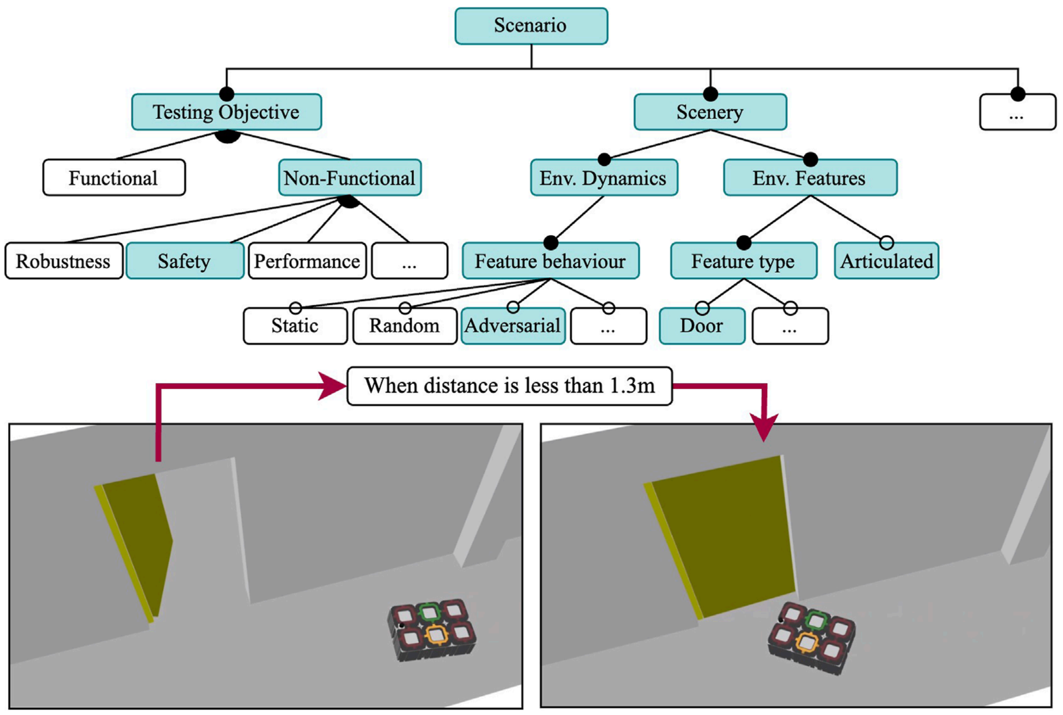


FIGURE 6
Feature model of Scenario 3 where the scenery door models behave adversarially. Shares the same SUT as Scenario 1 and 2.

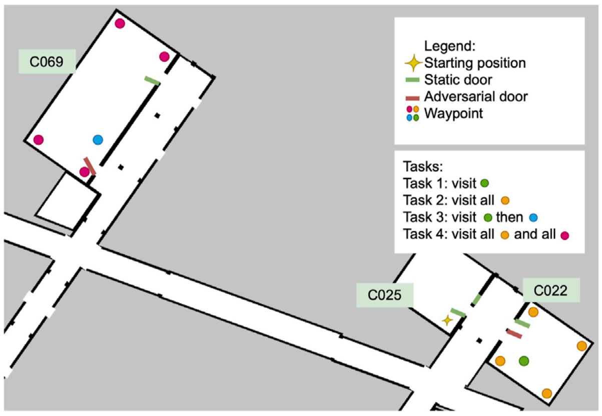


FIGURE 7
The scenery for Scenario 3, with two adversarial doors (door 17 in C022 and door 12 in C069) and four tasks that vary in scale and number of waypoints.

determined after experimenting with different values, and it ensures that the robot can detect the sudden change in the environment without the door hitting the robot. The order and position of the waypoints was intentionally selected in order to force the robot to plan to pass through the adversarial door.

Because we want the robot to attempt to go through the doorway (as it does not expect the door to close), we chose two different rooms to test this, as shown in Figure 7. In the figure, the four different tasks we composed to gradually increase the complexity of the scenario are also shown. On each task, the number of waypoints to visit and the distances between them increases. The tasks vary in number of waypoints to be visited and distance to the next waypoint. All the tasks start in the same pose in room C025. Task 1 and 2 are relatively short and only involve travelling to C022, while in Tasks 3 and 4 the robot must travel first to C022 and then to C069. Tasks 1 and 3 have a single Go To action in each room for a total of one and two waypoints, respectively; while in Tasks 2 and 4 the robot must perform a total of five and ten Go To actions in sequence, respectively. We name the concrete scenarios to match each task, Scenario 3.1 to 3.4.

### 4.4.1 Test oracle

We expect the safety requirement of $min(d_o(x)) \geq 0.55$ and $min(d_o(y)) \geq 0.433$ to be violated, since the doors will only close when the robot is near the door. However, the expected behaviour is that the robot will avoid collisions in all cases and attempt to move away from the obstacles until it reaches a safe distance again. We observe the changes in speed and angular velocity at the moment the door is triggered, as well as the distance to objects $d_o$, to analyse the robot's behaviour in context, e.g., whether it is violating the safety distance but moving slowly.

A passing test also requires the robot to complete the tasks successfully. Our hypothesis for this scenario is that the robot will be able to finish the task, but will take more time, as the adversarial elements will just impede the robot to take the shortest possible route. For this comparison, we create two additional sub-scenarios where the doors remain static and which match the most complex tasks at two scales: Task 2 and Task 4, which we name S3.5 and S3.6, respectively.

To measure how well the robot "recovers" once the door closes on its path, we measure the amount of time the robot has obstacles within its safety buffer $t_{d_s}$. We expect the distributions of the total runtime and the total time the Minimum Safety Distance (MSD) $t_{d_s}$ was violated for scenarios to behave similarly based on the task scale. Finally, we expect that the behaviour of the robot in an adversarial scenery vs. a static one should not differ substantially other than to avoid the effects of the adversarial door being opened. While we expect $t_{d_s}$ to be larger in the adversarial scenery, we expect that the total delay caused by the robot's reaction to the closing door and the detour caused by the closed door not add more than 30 s for door 17 and 60 s for door 12.

## 5 Results

We ran our experiments in an XMG laptop with 16 GB of RAM and an AMD Ryzen 9 5900HX CPU and running Ubuntu 18.04. The SUT described in Section 4 uses ROS1 noetic. Using the generated artefacts, we execute each scenario 5 times in Gazebo and analyse their results. The models and launch files used to run these scenarios can be found in https://github.com/secorolab/frontiers-replication-package.

### 5.1 Scenario 1

In all runs, the robot was able to reach all waypoints and complete the task. The time to complete the task was also consistent, with an average of 696.08 s and a standard deviation of 2.58 s. The behaviour of the robot was consistent across the five runs, with the localization error of 0.1238 m on average, and a maximum value of 0.522 m from run 1. Similarly, the standard deviation of the particle cloud was consistent, as can be seen in Figure 8. As expected, the standard deviation of the particle cloud in $y$ is larger than in $x$ (the direction of travel), and clearly increases whenever the uncertainty about the robot's orientation increases, i.e., when the robot makes turns. The confidence level of the localization component across all runs was 99.8%.

Despite the error being under the acceptable threshold on average, we can see that the localization requirements are violated briefly when the robot makes turns near the entrance to the hallway. Figure 9 shows the run with the largest error in more detail. On the zoomed in area, we see one of the moments at the beginning of the task, where the localization error and the standard deviation of the particle cloud both reached their maximum values in all runs. This area in particular has a lower number of features for the localization component, as no columns are in range for the laser sensors and there is an intersection right before entering the area.

Although all the runs were completed successfully, only three of the runs met the requirements of the localization component. While the confidence level of the localization component was high, meaning that 99.8% of the time the difference between the pose estimate and the ground truth pose is not statistically significant, the error is larger than the acceptable value for this scenario. This threshold was chosen to guarantee that potential errors in the localization would allow the robot to reach its goals without crashing against the walls.

The results reveal that there are areas in the environment that may require further testing, because although the scenery for this scenario is static and represents the nominal operating conditions for the SUT, the localization component does not meet the application requirements.

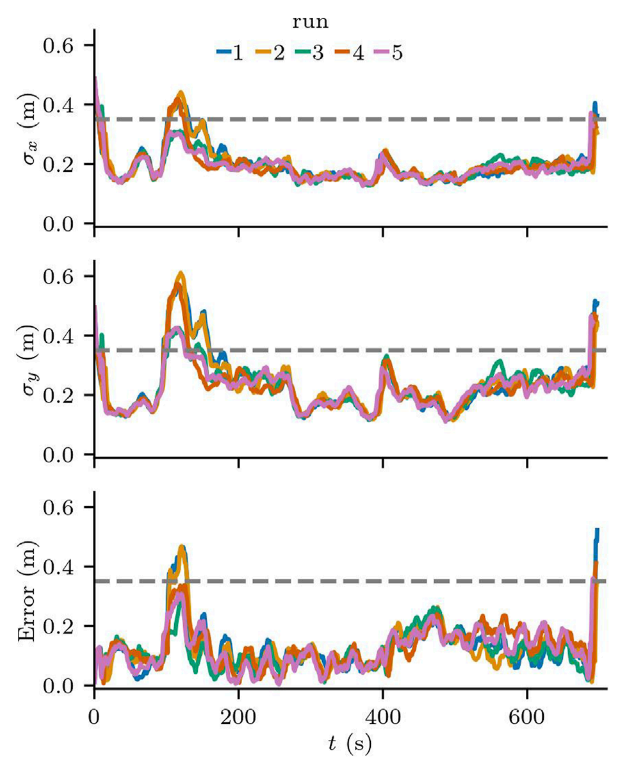


FIGURE 8
Localization error and standard deviation of the particles along the $x$ and $y$ axis in Scenario 1

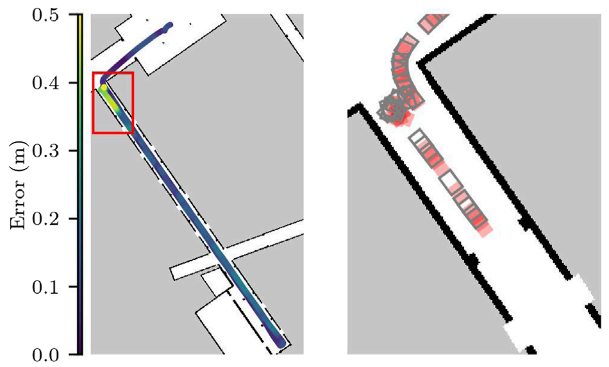


FIGURE 9
Localization error for run 1 in Scenario 1. On the right the zoomed-in version shows the first 130s of the run.

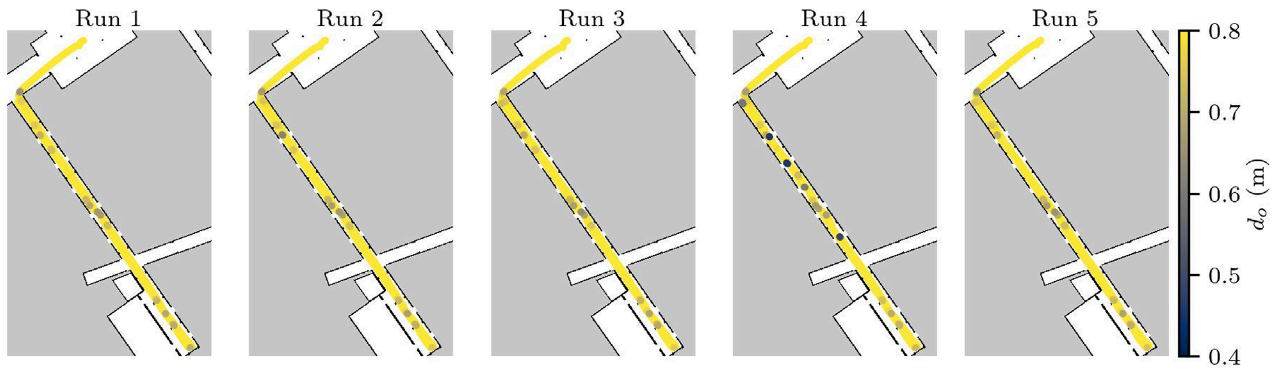


FIGURE 10
Distance from obstacles in Scenario 2. Distances larger than 0.8 m are shown in yellow, and the darker the purple, the closest the robot was to one of the doors in the hallway.

## 5.2 Scenario 2

All test runs for this scenario were successful, as no collisions were detected. The minimum distance to obstacles is shown in Figure 10. One can clearly see that $d_o$ decreases near the doorways, as expected. Although the average $d_o$ is 1.2 m, run 4 was particularly challenging for the robot; and it is noticeable that in a few locations the dynamic doors almost caused collisions. In this run, on two occasions, $d_o$ is less than 5 cm, meaning the robot managed to avoid a collision by merely 3.77 cm and 4.1 cm, respectively.

This scenario builds on top of the static scenery of Scenario 1, and increases the load for the obstacle avoidance component. In a general sense, the dynamic behaviour of the doors in the scenery helps validate the ability of the robot to react to dynamic obstacles in its environment. However, the near misses reveal risks of collision that should be validated against the safety requirements.

## 5.3 Scenarios 3.1–3.6

As a first step to validate the conformance to the safety requirements, we analysed whether there were any obstacles within the 0.2 m safety buffer. To our surprise, we discovered that the SUT struggled with the non-adversarial scenario S3.6, which had one run fail after the robot could not exit C022. Given that the non-adversarial scenery represents the static environment and hence nominal operating conditions, we could immediately conclude that the safety requirements were not being fulfilled. After further inspection, we noticed that the publicly available configuration of the navigation stack[8] (`a2s`) had several errors. The robot's footprint was much smaller and not symmetric around its centre (as can be seen in Figure 11); the laser scan topic used to update the costs for the path planner was using a namespace, i.e., `robile_john/scan_front` instead of `/scan_front`; and, finally, although the platform is omnidirectional and the odometry model used by AMCL was configured as such, the path planner was configured to behave as a differential drive robot.

To validate the safety requirements while trying to deliberately provoke a collision (in S3.1–S3.4), we corrected the configuration errors for the differential configuration (`fro-diff`), and added an omnidirectional configuration (`fro-omni`). Note that our goal is not to find an optimal configuration, but rather we focus on testing if the new configurations fix the problem we observed. The results of the laser measurements that violate the safety buffer for the five runs for each task and configuration can be seen in Figure 11.[9]

Next, we focus on the behaviour of the robot around the two adversarial doors: door 17 and door 12. We see the moment the doors are triggered as vertical dotted lines in Figure 12. We can see that the behaviour of the robot when door 17 is triggered is consistent regardless of the task. For Scenarios S3.3 and S.6, despite some variation on when door 12 is triggered, the behaviour is similarly consistent. Furthermore, Figure 12 shows that violations to the safety buffer do not only occur with the adversarial doors, but any time the robot passes through or near doorways, and that the combination of the task to execute and the state of the doors contribute significantly to the safety violations.

In all the runs, the robot momentarily violates the safety distance on multiple occasions, including the moments where it goes through the non-adversarial doors. Figure 13 shows the distribution for $t_{d_o \leq MSD}$ in all the tasks of Scenario 3. Except for the outlier with the larger $t_{d_s}$ of the `a2s` configuration, the difference in the mean of $t_{d_s}$ between `fro-diff` and `a2s` is not statistically significant. However, both differential configurations had failures on scenario S3.6, while `fro-omni` was the only configuration to successfully finish all tasks; the trade-off seems to be related to the amount of time the MSD is violated and suggests that there are possible improvements to the latter configuration.

8 This configuration has been used for over a year as part of a robotics course by several groups of students. Available at: https://github.com/a2s-institute/robile_navigation/tree/noetic/config

9 We also ran the tests for S1 and S2 with the new configurations, and validated that they perform similar to the original a2s configuration.

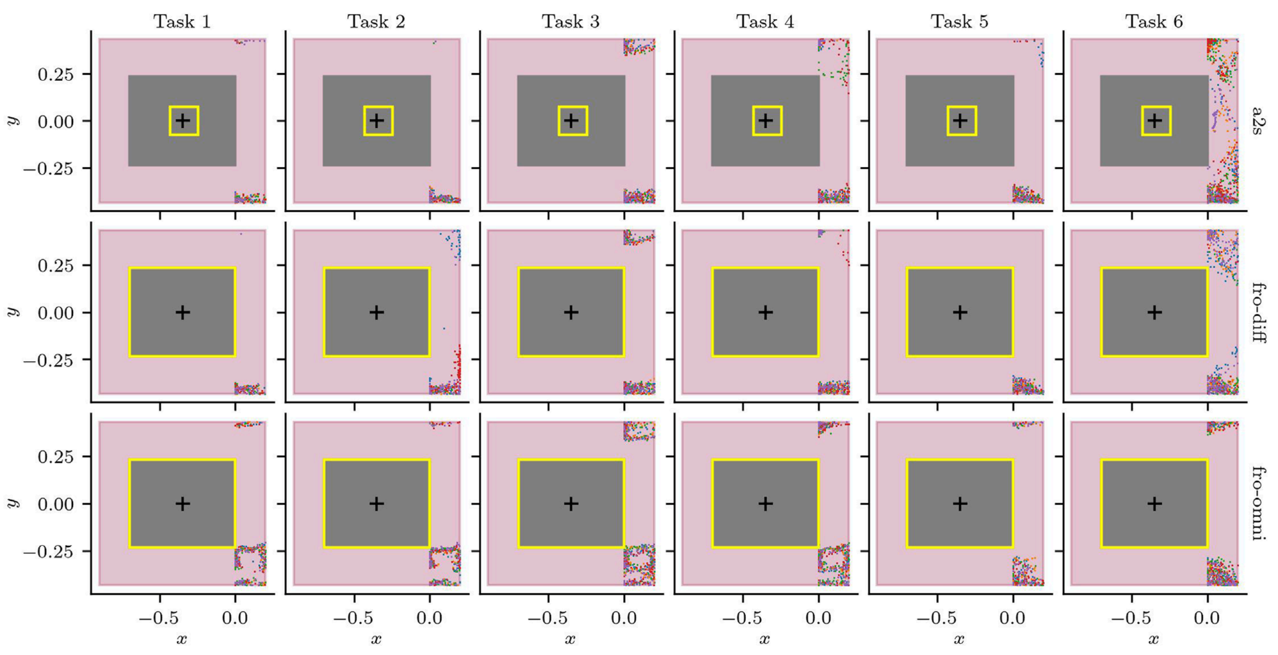


FIGURE 11
Laser measurements within the safety buffer $d_s$. The robot size (gray), the safety buffer (red) and its configured footprint (yellow).

By comparing runs in sceneries with and without adversarial doors, we can see the effects on the speed when the robot slows down as it attempts to avoid a collision, and the angular velocity changes as it turns to follow an alternate path. Figure 14 shows one run of S3.4 and the same task, but without the effects of the adversarial door in S3.6. We have zoomed in to the two moments where the robot reaches the trigger distance of 1.3 m to door 17 and then door 12.

The total $t_{d_s}$ for the different configurations makes the effects of the misconfiguration noticeable. Surprisingly, the original configuration `a2s` requires more time to complete S3.6 than S3.4, on average 8.9 s more (after excluding the outlier). Surprisingly, the fixed configuration `fro-diff` also requires more time for S3.6 than S3.4, although it is about half of the misconfigured SUT at 4.8 s. Finally, the `fro-omni` config behaves as expected, requiring only 1.3 s more for S3.6 than S3.4, which makes these the only successful test runs of scenario 3 in terms of performance.

At a grand scale, these tests reveal that the robot is able to avoid collisions to adversarial obstacles in its environment, attempting to go back to a safe distance as soon as the environmental change is detected. As expected, although the MSD was violated, the robot reacted quickly and the total time MSD was violated did not differ significantly between adversarial and non-adversarial sceneries.

However, upon closer inspection, the test results returned mixed results. Firstly, by using static and dynamic sceneries and a variety of tasks we were able to detect a misconfiguration issue. However, the proposed configurations to fix the issue still do not respect the required safety buffer and need further testing and tuning. Secondly, the tests also revealed that (new) nominal sceneries cause safety violations that still need to be handled, and the effect of the adversarial doors in the performance was overestimated with S3.6 taking longer despite the detour required by the adversarial door in S3.4. Although the scenario tests have met the safety criteria of our oracle regarding $t_{d_s}$, the performance trade-off for the differential configuration was unexpected. Finally, with these examples, we show how the composition plays a key role in the testing process; the interaction between the different features in the scenery and the gradual increase in complexity allowed us to systematically test the SUT and uncover issues that went unnoticed for over a year and in our test runs for S1 and S2.

## 6 Discussion

The results demonstrate that the use of composable and executable scenarios enables the design, specification and execution of tests for a variety of testing objectives. By reusing and composing different scenery models, our approach can gradually increase the complexity of the test scenarios with minimal effort. Due to the context-dependent nature of testing, the results above cannot be generalized to all robots or applications. However, we believe this does not necessarily limit its applicability to other systems due to our focus on the scenery models (which are robot-independent). Furthermore, our evaluation demonstrates how to design, specify and execute test scenarios for systems using the ROS navigation stack (a popular framework used in robotic applications). We now discuss aspects to be considered when applying our approach to a different robot or application.

The transformation and composition process presented in Section 3.3 has a small learning curve. Developers that wish to use our tools, must learn how to specify environments using the FloorPlan DSL and understand the basics of composable models, and making queries on RDF graphs. In its current iteration, because the graph construction relies on the identifiers for each of the

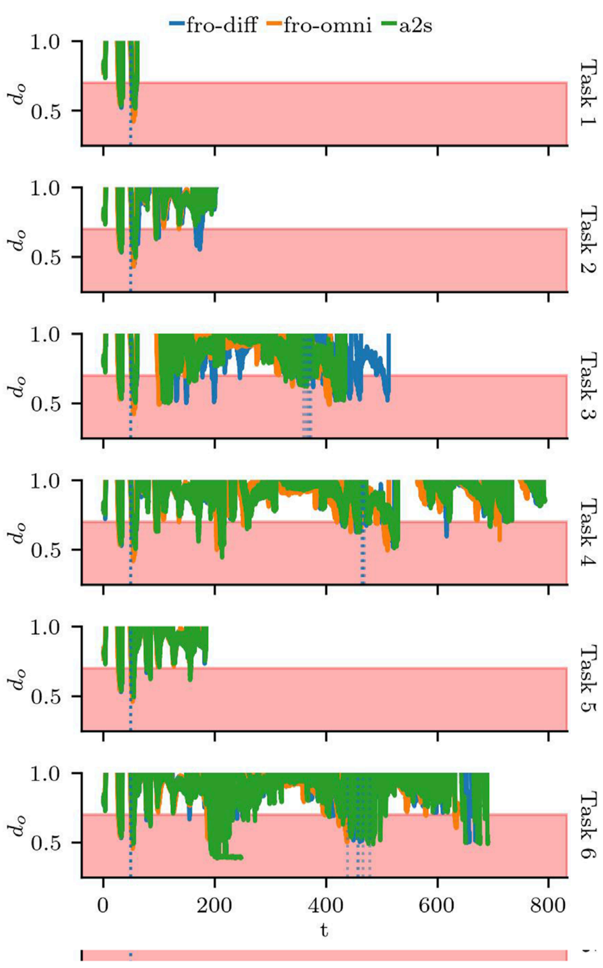


FIGURE 12
Distance to obstacles $d_o$ for Scenario 3. The red area shows the limit at which $d_s$ is violated. The dotted lines show when the doors were triggered.

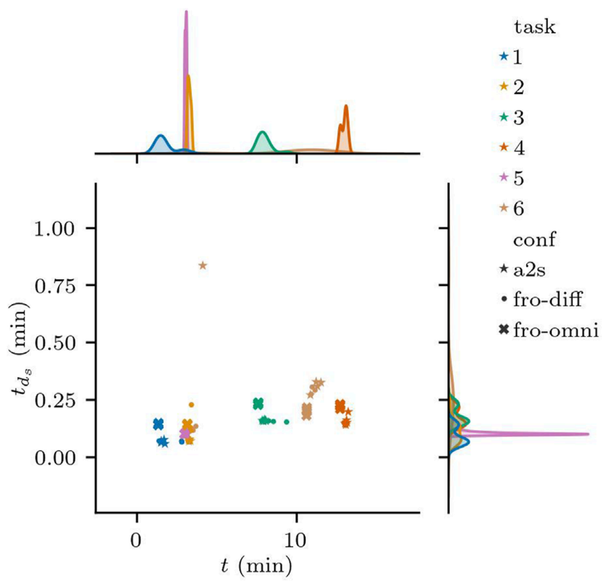


FIGURE 13
Distribution of the total time the Minimum Safety Distance $t_{d_s}$ was violated against the total runtime.

elements to be able to successfully compose and query the singular graph, the lack of front-end models for the scenery objects makes this process error-prone. The objects in this paper were limited to doors with hinges, however the composable metamodels for kinematic chains would also support the specification of sliding doors (or objects) by using prismatic joints. The addition of other types of objects is possible, but suffers the same limitations as the current specifications. We hope to extend our DSLs to be able to specify the scenery objects without having to worry about their composable specification.

Although not the focus of this paper, the validation of the scenery and scenario are another area of future development. In (Parra et al., 2023), we presented experiments on the real2sim gap, and shown how developers could validate that the scenery specification reflects a real-world environment. However, we have not yet implemented validation checks after models have been composed into the graph.

The transformation itself is currently handled in two different parts: The Composable FloorPlan model is generated using the textX infrastructure, while other execution artefacts are defined directly in the transformation engine. Ideally, we would like to define these transformations by using transformation rules and a model transformation language, however, target models (e.g., SDF) do not always have publicly available meta-models. Although this could potentially limit the generalization of the approach, the use of templating engines, such as jinja, provides some flexibility and ease of use for extending the type of artefacts supported and customizing the generated model itself. However, we plan to investigate the possibility of using transformation rules for those models with available meta-models to allow for a more systematic transformation.

There are also opportunities for automating the generation pipeline. The process currently is completely under the control of the developers, and each tool is executed manually and independently. On the one hand, this makes the tool modular and allows customizing which aspects of the scenery are to be generated. On the other, it requires additional effort to keep track of and maintain consistency between the models generated by different tools. The modularity of the tools also makes the integration of external or manually-defined models in a scenario possible, at least to some degree. Because composable models can reference other models by their identifiers, those external models can be referenced in the graph and on the templates of the artefacts that use those artefacts (e.g., the 3D Wall mesh or Gazebo models can be referenced by our generated Gazebo world). However, additional effort must be taken to ensure that the IDs, references, and relevant environmental features are valid and consistent with the FloorPlan models.

## 6.1 Related work

In contrast to the autonomous driving domain (Ren et al., 2022), scenario- and simulation-based testing of autonomous mobile

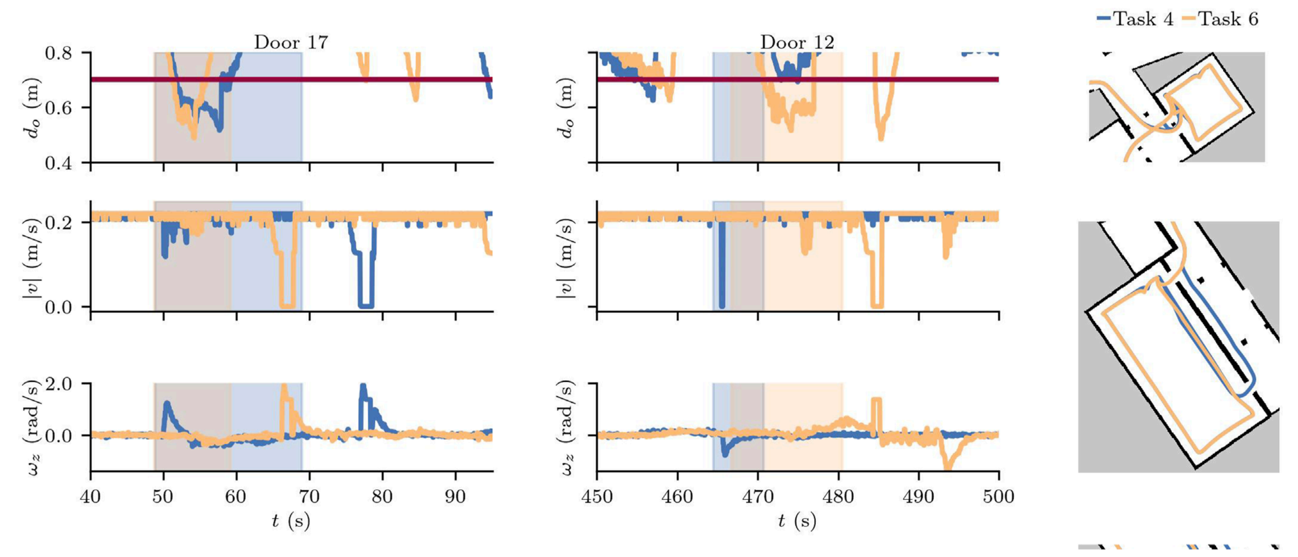


FIGURE 14
Min. safety dist. for a nominal scenario (Task 6) and a scenario with adversarial doors (Task 4) and configuration a2s.

robots is desirable; however, this has not been well established (Afzal et al., 2020; Afzal et al., 2021b). In the autonomous driving domain, scenario standards such as ASAM OpenSCENARIO (ASAM, 2022) are emerging to describe common scenery elements, such as roads, streets, traffic signs, and lanes. However, in robotics, environment and scenery modelling is traditionally supported by CAD tools published by multiple vendors. In the context of indoor robotics, and therefore relevant to our work, is the application of these approaches and tools from the architectural domain, where Building Information Modelling (BIM) has been an established technique to model the geometric information of building structural components (e.g., walls, corridors, and windows), as well as semantic hierarchical information (e.g., about the accessibility and connectivity of rooms) (Borrmann et al., 2018). The composable scenario modelling approach introduced in this work targets robotic experts, where BIM is not as prevalent as in other engineering domains. For example, during the interviews, only a single mention of BIM was made. Even though there are numerous 3D software commercially available that implements the BIM standard, modelling scenery is still considered a time-consuming task by robot application developers, as supported by our interviews. In addition, because BIM models support the full building management lifecycle, they introduce many irrelevant dependencies, such as the latest IFC 4.3. x schema[10] including concepts to define structural building elements (IfcWall, IfcDoor, etc.), but simultaneously introduce concepts for measurements of physical quantities (IfcAbsorbedDoseMeasure, IfcMolecularWeightMeasure, etc.) or building lifecycle management (such as actor roles, including civil engineer or building owner, but also orders, including purchase orders or maintenance work orders).

10 https://github.com/buildingSMART/IFC4.3.x-output/blob/master/IFC.xsd

Furthermore, as pointed out by Hendrikx et al. (Hendrikx et al., 2021), BIM cannot be considered accurate or complete for robotic applications.

Another domain related to our approach is the field of computer graphics, specifically procedural content generation approaches, which focus on synthesizing hundreds of environments separated from a single environment description. In robotics, these approaches are typically employed for machine learning applications, as they require a substantial amount of training and testing data that is arduous for manual production. Different approaches use diverse abstractions as inputs, including constraint graphs (Para et al., 2021), handmade drawings (Camozzato et al., 2015), building contours (Lopes et al., 2010; Mirahmadi and Shami, 2012), and natural language descriptions (Chen et al., 2020). A common theme of these approaches is that the output of the generation step is uncontrollable. Input abstractions deliberately exclude spatial relations to keep the input simple because numerous outputs must often conform to the input model. Thus, these spatial relations are synthesized by algorithms and are not controlled by the user. However, not all procedural generational approaches follow this pattern. Some have rich descriptions that allow for a more structured output, while still enabling the generation of variations. For example, the language presented by Leblanc et al. (2011) is an imperative specification language for building indoor environments by performing space operations. These operations involve complex logic to create variation. However, these approaches are not employed in the context of scenario-based testing of robotic systems, where additional models of other agents, dynamic scenery elements, and task specifications need to be composed.

A closely related approach is the Scenic language (Fremont et al., 2019), a probabilistic programming language for generalized environment specification that targets machine-learning applications. Scenic enables the specification of spatial relationships with concrete and logical values. However, Scenic is just a language, and to consume its models, an extra tool is needed to violate our design ambition of having composable scenario models. GzSCenic Afzal et al. (2021a) is a third-party tool that leverages scenic models to generate scenes using the robot simulator Gazebo (Koenig and Howard, 2004). Although these approaches can generate models that are consumable by simulators, they lack the generation of other artefacts for the direct simulation of navigation tasks, such as the occupancy grid map.

Another modelling approach to describe indoor environments is supported by the indoor tagging schema of Open Street Map, which supports modelling a floor plan with tags such as room, area, wall, corridor, and level (floor). The schema was devised for indoor navigation, but can be consumed by any application. Naik et al., (2019) presented an extension to the schema, which was exploited to generate occupancy grid maps and waypoints for navigation. However, these models were not used to generate the simulation models.

### 6.2 Conclusion and future work

In this study, we presented a domain model for features in simulation-based testing scenarios, which we derived from interviews to 14 domain experts. Based on the insights, we propose a composable modelling approach to specify and execute scenarios. Given that the environment representation is one of the challenges mentioned frequently in the interviews, our focus was on facilitating the specification and reuse of scenery models for testing.

The specification of these scenarios starts with a floor plan model that represents the environment in which the robot operates. This specification is done using the FloorPlan DSL from our previous work (Parra et al., 2023). In this paper, we present an extension to the FloorPlan M2M generator, which takes a floor plan model as input and creates a graph representation of its spaces, geometry and elements in JSON-LD, which we call the composable floor plan model. This representation is key to the composability and reusability of the models. Task specifications are generated by our proof-of-concept tool that uses the composable floor plan model to query the geometric information for a target area, and generate waypoints in free space based on its contours. In addition, objects can be composed into the static scenery of the floor plan by specifying an articulated scenery object model (describing the object geometry and its joints), and scenery instance models for each object. These scenery models are also specified in JSON-LD, and are an input for our scenery composer, which traverses the linked graph to generate the required artefacts for the execution of the scenario in simulation. Finally, at runtime, we introduced three Gazebo plugins which set the joint position of the objects composed into the scenery at the start of the simulation, or using time or event-based triggers.

We demonstrated our approach by performing a small simulation-based testing campaign for a mobile robot in a university building. The scenarios gradually increased their complexity, first focusing on validating the navigation stack with functional tests, then performing robustness tests on a highly dynamic environment, and finally, validating the conformance to the safety requirements. The composable aspect allowed us to reuse the static floor plan scenery specified in Parra et al. (2023), and compose static doors for Scenario 1, randomly opening and closing doors in Scenario 2, and doors that would close as the robot approached them in Scenario 3. Surprisingly, only when we ran the third scenario were we able to find a misconfiguration issue in the publicly available navigation stack of our SUT, which had been undetected for over a year despite being in use by multiple groups of students. Normally, this robot operates autonomously within a single room or hallway, and is teleoperated out of the room for the latter, explaining why this issue was not detected until now. This shows that the variation in the scenario features is essential to expose the robot to situations that may generate failures.

Future work includes creating DSLs to specify the scenery objects and instances, and expanding our proof-of-concept task generator to generate task specifications for existing mission and task DSLs. This is a key step to explore the ability of a fully-automated scenario generation approach, which could exploit the Variation DSL introduced in Parra et al. (2023).

## Data availability statement

The raw data supporting the conclusions of this article will be made available by the authors, without undue reservation.

## Ethics statement

Ethical approval was not required for the studies involving humans because the discussion in this paper only concerns the aggregated data from interviews, which was only used to identify common codes. Interview participants signed an informed consent, and their participation was voluntary. All the data related to the interviews is anonymized, and only the participants have their ID, which they can use to withdraw their participation at any time. The experiments are conducted on our own robot platform and software, minimizing risks or harms to participants and organizations, while enabling them to benefit from the results of our research. The studies were conducted in accordance with the local legislation and institutional requirements. The participants provided their written informed consent to participate in this study. No potentially identifiable images or data are presented in this study.

## Author contributions

AO: Conceptualization, Data curation, Investigation, Methodology, Software, Writing–original draft, Writing–review and editing, Formal Analysis, Validation, Visualization. SP: Conceptualization, Investigation, Methodology, Software, Writing–review and editing, Data curation, Writing–original draft, Formal Analysis, Visualization. NH: Conceptualization, Funding acquisition, Project administration, Supervision, Writing–original draft, Writing–review and editing, Methodology, Validation.

## Funding

The author(s) declare financial support was received for the research, authorship, and/or publication of this article. This wor has partly been supported by the European Union's Horizon 2020 project SESAME (Grant No. 101017258).

## Acknowledgments

We thank the Institute for AI and Autonomous Systems at the Hochschule Bonn-Rhein-Sieg for their support.

## Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## Publisher's note

## Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/frobt.2024.1363281/full#supplementary-material

## References

Afzal, A., Goues, C. L., Hilton, M., and Timperley, C. S. (2020). "A study on challenges of testing robotic systems," in IEEE Intl. Conf. on Software Testing, Verification an Validation (ICST), Porto, Portugal, 24-28 October 2020, 90–107. doi:10.1109/ICST46399.2020.00020

Afzal, A., Goues, C. L., and Timperley, C. S. (2021a). GzScenic: automatic scene generation for gazebo simulator. arXiv:2104.08625 *[Preprint]*. doi:10.48550/arXiv.2104.08625

Afzal, A., Katz, D. S., Le Goues, C., and Timperley, C. S. (2021b). "Simulation for robotics test automation: developer perspectives," in IEEE Intl. Conf. on Software Testing, Verification and Validation (ICST), Porto de Galinhas, Brazil, 12-16 April 2021, 263–274. doi:10.1109/ICST49551.2021.00036

ASAM (2022). *ASAM OpenSCENARIO standard. Association for standardization of automation and measuring systems.* Online; [last accessed 2023-December-28]

Borrmann, A., König, M., Koch, C., and Beetz, J. (2018). "Building information modeling: why? What? How?," in *Building information modeling: Technology foundations and industry practice* (Springer International Publishing), 1–24. doi:10.1007/978-3-319-92862-3_1

Camozzato, D., Dihl, L., Silveira, I., Marson, F., and Musse, S. R. (2015). Procedural floor plan generation from building sketches. *Vis. Comput.* 31, 753–763. doi:10.1007/s00371-015-1102-2

Chen, Q., Wu, Q., Tang, R., Wang, Y., Wang, S., and Tan, M. (2020). "Intelligent home 3D: automatic 3D-house design from linguistic descriptions only," in *Proc. Of the IEEE/CVF conf* (on Computer Vision and Pattern Recognition), 12625–12634. doi:10.1109/CVPR42600.2020.01264

Dejanović, I., Vaderna, R., Milosavljević, G., and Vuković, v. (2017). TextX: a Python tool for domain-specific languages implementation. *Knowledge-Based Syst.* 115, 1–4. doi:10.1016/j.knosys.2016.10.023

Fremont, D. J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A. L., and Seshia, S. A. (2019). "Scenic: a Language for scenario specification and scene generation," in ACM SIGPLAN Conf. on Programming Language Design and Implementation (ACM), Xi'an, China, 30 May 2021 - 05 June 2021, 63–78. doi:10.1145/3314221.3314633

Hauer, F., Pretschner, A., and Holzmüller, B. (2020). Re-Using concrete test scenarios generally is a bad idea. *IEEE Intell. Veh. Symp.* IV, 1305–1310. doi:10.1109/IV47402.2020.9304678

Hendrikx, R. W. M., Pauwels, P., Torta, E., Bruyninckx, H. J., and van de Molengraft, M. J. G. (2021). "Connecting semantic building information models and robotics: an application to 2d lidar-based localization," in IEEE Intl. Conf. on Robot. and Autom. (ICRA), 11654–11660. doi:10.1109/ICRA48506.2021.9561129

Hove, S., and Anda, B. (2005). "Experiences from conducting semi-structured interviews in empirical software engineering research," in IEEE Intl. Software Metrics Symp. (METRICS), Como, Italy, 19-22 September 2005, 10–23. doi:10.1109/METRICS.2005.24

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). Feature-oriented domain analysis (FODA) feasibility study. *Tech. Rep. CMU/SEI-90-TR-021.*

Koenig, N., and Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. *IEEE/RSJ Intl. Conf. Intell. Robots Syst. (IROS)* 3, 2149–2154. doi:10.1109/IROS.2004.1389727

Leblanc, L., Houle, J., and Poulin, P. (2011). Component-based modeling of complete buildings. *Graph. Interface* 2011, 87–94.

Lopes, R., Tutenel, T., Smelik, R. M., De Kraker, K. J., and Bidarra, R. (2010). "A constrained growth method for procedural floor plan generation," in *Proc. Of the int. Conf. Intell. Games simul*, 13–20.

Mirahmadi, M., and Shami, A. (2012). A novel algorithm for real-time procedural generation of building floor plans. arXiv:1211.5842 *[Preprint]*. doi:10.48550/arXiv.1211.5842

Naik, L., Blumenthal, S., Huebel, N., Bruyninckx, H., and Prassler, E. (2019). "Semantic mapping extension for OpenStreetMap applied to indoor robot navigation," in 2019 International Conference on Robotics and Automation (ICRA), Montreal, QC, Canada, 20-24 May 2019, 3839–3845. doi:10.1109/ICRA.2019.8793641

Ortega, A., Hochgeschwender, N., and Berger, T. (2022). "Testing service robots in the field: an experience report," in IEEE/RSJ Intl. Conf. on Intell. Robots and Syst. (IROS), Kyoto, Japan, 23-27 October 2022, 165–172. doi:10.1109/IROS47612.2022.9981789

Para, W., Guerrero, P., Kelly, T., Guibas, L. J., and Wonka, P. (2021). "Generative layout modeling using constraint graphs," in Proc. of the IEEE/CVF Intl. Conf. on Computer Vision, Montreal, QC, Canada, 10-17 October 2021, 6690–6700. doi:10.1109/ICCV48922.2021.00662

Parra, S., Ortega, A., Schneider, S., and Hochgeschwender, N. (2023). "A thousand worlds: scenery specification and generation for simulation-based testing of mobile robot navigation stacks," in *IEEE/RSJ Intl. Conf. On intell. Robots and syst. (IROS)*, 5537–5544. doi:10.1109/IROS55552.2023.10342315

Ren, H., Gao, H., Chen, H., and Liu, G. (2022). "A survey of autonomous driving scenarios and scenario databases," in Intl. Conf. on Dependable Syst. and Their Applications (DSA), Wulumuqi, China, 04-05 August 2022, 754–762. doi:10.1109/DSA56465.2022.00107

Saldaña, J. (2021). The coding manual for qualitative researchers. *Coding Man. Qual. Res.*, 1–440.

Schneider, S., Hochgeschwender, N., and Bruyninckx, H. (2023). "Domain-specific languages for kinematic chains and their solver algorithms: lessons learned for composable models," in 2023 IEEE International Conference on Robotics and Automation (ICRA), London, United Kingdom, 29 May 2023 - 02 June, 9104–9110. doi:10.1109/ICRA48891.2023.10160474

Sotiropoulos, T., Waeselynck, H., Guiochet, J., and Ingrand, F. (2017). "Can robot navigation bugs Be found in simulation? An exploratory study," in IEEE Intl. Conf. on Software Quality, Reliability, and Security (QRS), Prague, Czech Republic, 25-29 July 2017, 150–159. doi:10.1109/QRS.2017.25

Tang, S., Zhang, Z., Zhang, Y., Zhou, J., Guo, Y., Liu, S., et al. (2023). A survey on automated driving system testing: Landscapes and trends. *ACM Trans. Softw. Eng. Methodol.* 32, 1–62. doi:10.1145/3579642

Timperley, C. S., Afzal, A., Katz, D. S., Hernandez, J. M., and Le Goues, C. (2018). Crashing simulated planes is cheap: can simulation detect robotics bugs early? *IEEE Intl. Conf. Softw. Test. Verification Validation (ICST)*, 331–342. doi:10.1109/ICST.2018.00040

TYPE Technology and Code
PUBLISHED 04 September 2024
DOI 10.3389/frobt.2024.1363041

Check for updates

OPEN ACCESS

EDITED BY
Federico Ciccozzi,
Mälardalen University, Sweden

REVIEWED BY
Cezary Zielinski,
Warsaw University of Technology, Poland
Antonio Cicchetti,
Mälardalen University, Sweden
Muhammad Waseem Anwar,
Mälardalen University, Sweden

*CORRESPONDENCE
Maria I. Artigas,
mariaisabel.artigasalfonso@kuleuven.be

RECEIVED 29 December 2023
ACCEPTED 12 August 2024
PUBLISHED 04 September 2024

CITATION
Artigas MI, Rodrigues RT, Vanderseypen L and Bruyninckx H (2024) Software patterns and data structures for the runtime coordination of robots, with a focus on real-time execution performance.
*Front. Robot. AI* 11:1363041.
doi: 10.3389/frobt.2024.1363041

# Software patterns and data structures for the runtime coordination of robots, with a focus on real-time execution performance

Maria I. Artigas[1,2]*, Rômulo T. Rodrigues[1,2], Lars Vanderseypen[1] and Herman Bruyninckx[1,2,3]

[1]Department of Mechanical Engineering, KU Leuven, Leuven, Belgium, [2]Flanders Make, Leuven, Belgium, [3]Department of Mechanical Engineering, TU Eindhoven, Eindhoven, Netherlands

This paper introduces software patterns (registration, acquire-release, and cache awareness) and data structures (Petri net, finite state machine, and protocol flag array) to support the coordinated execution of software activities (also called "components" or "agents"). Moreover, it presents and tests an implementation for Petri nets that supports real-time execution in shared memory for deployment inside one individual robot and separates event firing and handling, enabling distributed deployment between multiple robots. Experimental validation of the introduced patterns and data structures is performed within the context of activities for task execution, control and perception, and decision making for an application on coordinated navigation.

KEYWORDS

**multi-robot, coordination, Petri net, finite state machine, real-time, shared memory**

## 1 Introduction

Society expects "smarter" robotics technology and "higher performance" of the applications and systems that are built with it. A major contribution toward realizing these expectations is improving the capabilities and the predictability of the composition of robotic components into systems. Coordination plays a major role in achieving this predictability: a system has several concurrently active components that require access to "resources" that cannot be shared trivially, such as locations in space or tools and sensors. Application developers must translate user requirements into concrete *coordination specifications*: when and why each of the components in the system must start or stop a particular "behavior." Coordination is triggered by "events" generated by the software component in the system that has the authority to make such decisions, and it is provided with the necessary information by all the components that rely on its coordination. A good (but not necessarily unique) *separation of concerns* (Dijkstra, 1982) approach to ensure coordinated resource sharing with predictable performance and acceptable access policies is to introduce a *dedicated coordination software component* for each shared resource. The contributions of this paper are focused on this coordination design concern.

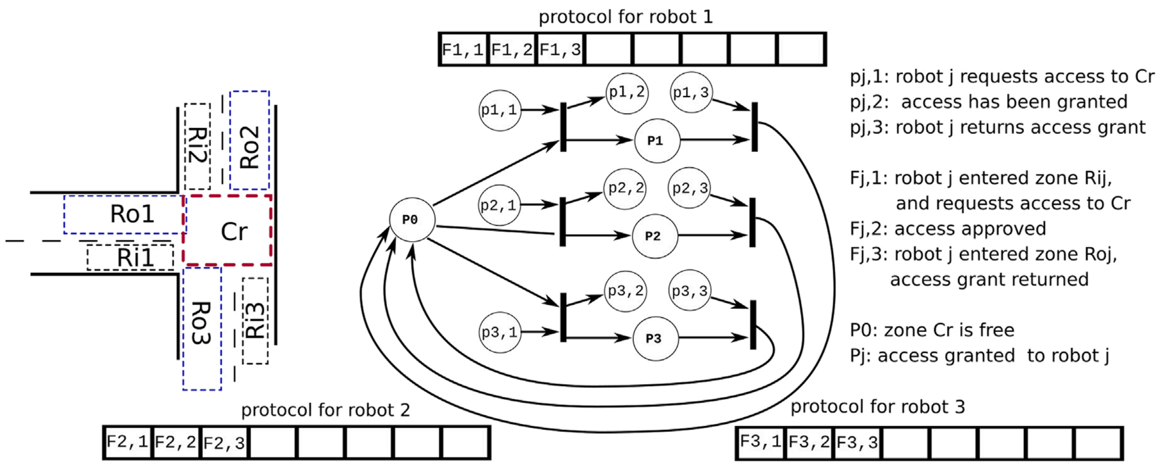

protocol for robot 1
F1,1 F1,2 F1,3
pj,1: robot j requests access to Cr
pj,2: access has been granted
pj,3: robot j returns access grant
Fj,1: robot j entered zone Rij, and requests access to Cr
Fj,2: access approved
Fj,3: robot j entered zone Roj, access grant returned
P0: zone Cr is free
Pj: access granted to robot j
Ri2
Ro2
Ro1
Cr
Ri1
Ro3
Ri3
p1,1 p1,2 p1,3 P1
p2,1 p2,2 p2,3 P2
p3,1 p3,2 p3,3 P3
P0
protocol for robot 2
F2,1 F2,2 F2,3
protocol for robot 3
F3,1 F3,2 F3,3

FIGURE 1
Coordination between concurrently active "agents" in traffic situations, particularly a T-junction. Left: only one "robot" coming from one of the three roads shall be allowed to access the crossing Cr. Right: our design introduces a *mediator* software component to realize such coordination problems. It relies on i) a *Petri net* as a *declarative model* of the coordination's decision making and ii) a *protocol* between the mediator and each of the coordinated robots, via which the latter's own internal decision making is *decoupled* from that of all other robots.

The left-hand side of Figure 1 shows a simple example of the role of coordination in multi-robot systems (Section 1.3 provides an overview of more archetypical coordination-use cases).

- The sketch on the left-hand side represents a "T junction."
- Robots can come from three different roads, each with the timing unknown to the other robots.
- The "crossing area" Cr is the "shared resource" that should be entered by only one robot at a time.

The figure's right-hand side sketches our software design (which is described in detail in the later sections of this paper).

- The crossing area gets its own *mediator* software component (Gamma et al., 1995). The mediator allows robots to navigate the crossing area in a coordinated way. The core data structure of the mediator is a *Petri net* that represents a *declarative model* of the coordination's decision-making.
- The second software component is a *map* data structure that all robots share with the mediator. On that map, they indicate which area they are currently driving in. These areas are given numbers 1, 2, and 3 for each of the three roads "R"; "i" and "o" indicate the "incoming" and "outgoing" lanes.
- The third software component is a *protocol* data structure that is accessed in sequence by the mediator and each of the coordinated robots. The protocol decouples a robot's own internal decision making from that of all other robots.

The *map* is also a shared resource in itself, but its software design presents a different set of coordination challenges, which are beyond the scope of this paper; for further details, refer to Van Baelen et al. (2022).

The following sub-sections introduce and define all the concepts needed in this paper. Section 2 discusses the previous work on which this paper is based and other related work. Section 3 describes the coordination mechanisms introduced in this paper and the complementary communication and configuration mechanisms for its integration. Section 4 introduces the implementation and evaluation of the Petri nets for runtime coordination. Section 5 explains the application of the previously described patterns in a coordinated navigation case. A secondary demonstration is also provided. Section 6 concludes the paper with a discussion of the presented and future work. Supplementary Appendix SA explains the connection between the coordinating and coordinated activities via events.

## 1.1 Component

The terminology "(software) components" has been interpreted several times over the past decades (Brugali and Scandurra, 2009; Brugali and Shakhimardanov, 2010), referring to the software primitive that provides "computational behavior" to a system. The terminology used in this paper to represent complementary types of computational behavior is as follows:

- (robot) Component: each piece of software-controlled hardware that the application identifies as a "robot." It is not to be subdivided further hardware-wise and can be connected to other robot components via mechanical, power, and information connectors to form a larger "composite" robot component.
- Computer: the set of CPUs, each with possibly several computing cores and managed by one operating system. Many robot *components* are built with more than one computer.
- Process and thread: the two well-known *application-independent* computational primitives under the control of an operating system.
- Activity: the smallest concurrently running piece of software that components the need and is deployed in a thread. Typically, each component requires application-centric functionalities implemented in a multitude of activities, all

running *asynchronously* on the same computer or different computers.

- Algorithm: an activity can execute one or more algorithms inside, for which it guarantees the *synchronous* execution context needed to realize the *functionalities* (or "*behavior*") of a component. In other words, the activity is responsible for asynchronous data exchange between activities, making sure that their algorithms only have to access locally stored data structures that are, hence, synchronously processed.

One could have given the name software component to what is called *activity* above. Because activities are designed to be executed concurrently, an appropriate set of asynchronous data exchange mechanisms is needed; these mechanisms should be shared between the activities within the same process memory or use one or more inter-process communication technologies. The challenges of data consistency between concurrently running activities are to be solved at the activity level but not at the thread or algorithm levels. The thread level in a software component design is responsible for scheduling by the operating system. The process level is responsible for managing resources shared between all activities within all process's threads, such as file descriptors, signal handling, and thread priorities.

## 1.2 Coordination

Coordination is all decision making shared between concurrently executing *activities* about which of their *algorithms* ("behaviors") must become "(in)active" at each moment in time in each of the robot components and about how to keep other robot components informed about which behavior(s) are currently "(in)active." A key message of this paper is that all forms of inter-activity coordination can be realized with the following primitives, whose "separation of concerns" roles (Dijkstra, 1982) are illustrated in Figure 1.

- Flag: this represents the "state" of a *Boolean condition* defined over a set of parameters in the behavior(s) of an activity. For example, for mobile robots navigating in the neighborhood of the crossing in Figure 1, flags can indicate areas in which each mobile robot finds itself. (The above-mentioned "map" software component could act as the major source of *flag* information and *event* information introduced below.)
- Event: this represents the change in the Boolean state of a flag. Because Figure 1 is a static "snapshot" of the status of the world, it does not show *events.* They only come in when the time-varying *dynamics* of the coordination problem are considered and they are to be *communicated* between activities.
- Finite state machine [FSM, Hrúz and Zhou (2007)]: each of the activities needs to realize a particular behavior in a particular order. Such an order is represented *declaratively* by a finite state machine data structure and behavior:
  - Each activity can be "in" *one and only one state* at a time.
  - In each state of the finite state machine, the activity executes a particular set of algorithms and communicates a particular set of data structures, including *events.*
  - Transitions between states are triggered by incoming events or events generated internally in the activity.
  - Some of these transitions can also give rise to the *firing* of events that must be communicated to other activities.

This description of the *mechanism* of an FSM corresponds to that of a *Mealy machine* (Mealy, 1955), which is formally represented as a tuple $(S, I, O, \mathcal{T}, \mathcal{O})$, with $S$ representing the finite set of *states*; $I$ representing the finite set of *input* events (or "input symbols"); and $O$ representing the finite set of *output* events (or "output symbols").

- $\mathcal{T}$: the *transition function* $\mathcal{T}{:}S \times I \rightarrow S$ maps the combination of a *state* and an *input* event to a *state.*
- $\mathcal{O}$: the output function $\mathcal{O}{:}S \times I \rightarrow O$ maps the combination of a state and an input event to an output event.

In the actual execution of an FSM, the *policy* must be added to select one of the states as the *initial state* $S_0$.

- Petri net (PN): this is a data structure that keeps track of the (externally exposed) state of a set of activities that need to be coordinated in the coordination *mediator* software component, as shown in Figure 1. Each of these states fills a *place* in the Petri net with a *token.* (This paper uses only the simplest form of Petri nets, sometimes called *safe* Petri nets (Barylska et al., 2017), in which each place can hold only zero or one *token.*) The role of the Petri net is to support decisions about the coordination *between activities* and not about the *internal algorithm* coordination of one single activity. *Semantically*, a Petri net can have more than one of its places *marked* at any given time, while a finite state machine can *be* in only one of its states at any given time.

This *mechanism*of a Petri net is formally represented as a tuple $(P, T, M, \mathcal{F})$, with $P$representing the finite set of *places*; $T$representing the finite set of *transitions*($P$and $T$are always disjoint); $M$representing the set of *markings*of the Petri net, where each marking is a mapping $M{:}P \times \{0,1\}$, indicating whether a *place*is marked or not, that is, it contains a *token*or not; and $\mathcal{F}$representing the *firing function*such that $\mathcal{F}{:}P \times M \rightarrow M$removes the *tokens*in the input *places*of a *transition*whenever all these *places*are *marked*and produces a *token*in each of the *transition*'s output *places.*

In the actual execution of a Petri net, the *policy* must be added to define an *initial marking* $M_0$.

- Protocol: this represents the order in which a particular subset of flags is allowed to be set to "true" in the interaction between the coordination *mediator* and *one* of the coordinated activities. Such an order must be agreed upon *in advance* by all activities participating in the coordination mediation to be able to guarantee temporal constraints between behavior state changes.

For example, the protocols in Figure 1 show that for each robot, the sequence of execution is as follows: 1) the robot requests access, 2) the access is approved, and 3) the robot can enter the area.

Note that the "array" used in Figure 1 to represent a protocol is *always finite*, and flag entries are entered always from the first entry on the left. In other words, it is *not* an endlessly growing "stream" of flag entries. When the protocol ends, for one reason or another, all entries are removed so that the next execution of the protocol starts with an empty array again.

In the simple *workspace sharing* example in Figure 1, the *labeled circles* (called "places") represent *conditions* that can be true or false, and the *solid lines* (called "transitions") represent decision making: if all the input conditions are true, the transition is "fired." The result is that the conditions in the input places are put to false again, and those in the output places become true. The truth values of the "source" places (i.e., those without input transition) are *determined* by the flags in the *protocol* arrays. Similarly, the truth values of the "sink" places (i.e., without output transition) *determine* the value of the corresponding flags in the *protocol*.

The ideal lifetime of an event is "zero": as soon as an event is *fired* by an activity, all the activities that need to react to the event (that is, "to handle" it) will *consume* the event during their reaction. The software architecture of such coordinated components must foresee the *communication* of events between the firing activity and each of the handling activities, which is (one of the reasons) why asynchronous data exchange is needed between these activities.

Figure 1 uses the simplest form of a protocol sequence, namely, an *array*; in general, protocols consist of compositions of more than one such array, representing different allowable "paths" in the coordination. Note the important difference between the very narrow and lean semantics of a "protocol" as needed in this paper and the much wider semantics of "protocol stacks" as used in inter-process communication (Delanote et al., 2008).

### 1.3 Archetypical use cases

The following example set of multi-robot applications, with multi-tasking functionalities for each robot, is representative of the scope of this paper's coordination design contributions:

- Workspace sharing. This involves scenarios where multiple mobile robots (flying, wheeled, and legged) from possibly different vendors (and hence with independently developed software capabilities) need to share the same space in a warehouse or orchard. The same holds for multiple manipulator arms on conveyor belts or at assembly and fruit harvesting stations. In addition, both types of robotic components should also physically interact with each other, like an assembly robot arm that can take parts from a mobile robot that brings the parts from storage.
- Execution protocols. For example, robots must *register* with the "manager" of a shared resource (charging station, parking space, inland waterway lock, and gripper on a fruit harvesting robot) and then follow a *protocol* coordinated by that manager every time they want to use that resource. Being able to coordinate the execution of different robots in a predictable, agreed-upon way is another necessary (but not sufficient) condition for sharing *physical workspace*.
- Task sharing**.** A typical example is two mobile robots in a manufacturing cell that coordinate how to share the same *areas* during the execution of their tasks, such as driving the *routes* through the depicted stations. Other applications requiring robots to share task executions are include carrying or pushing a shared load, covering a whole agriculture field or a surveillance area, closing a control loop around other robots' sensor capabilities, and platooning in traffic. Task sharing is *the* driving *end-user pull* behind having to spend design and implementation efforts on all the archetypical challenges mentioned above.

### 1.4 Scope

This paper focuses on the software design of the **coordination** of *runtime decision making*, including data structures, policies, decision-making functionalities, software patterns, and best practices. An implementation using Petri nets, with the purpose of being used within these coordination patterns, is explained and evaluated. As the final validation, the previous patterns are applied to two coordinated navigation cases.

Subjects outside the scope of this study are the *functional algorithms* that define the *behavior* inside activities, the creation of *maps* and *Petri nets*, the *policies* behind the reasons why the *application* takes these decisions, and the *communication functionalities* via which activities exchange the *data* they need from each other to realize their functional behavior.

### 1.5 Contributions

The *contributions* of the paper are

- The software mechanisms of coordination, which encompasses everything needed to fire and handle events that allow concurrent activities to coordinate their executions. In particular, this includes the complementary roles of *finite state machines* and *Petri nets* by introducing two non-traditional primitives (the *protocol array* and the *event circular buffer*) that help in the *separation of concerns* (Dijkstra, 1982) of the mentioned complementary roles within the presented software design.
- Explicit awareness of the implementation constraints, which are introduced by the distributed, multi-core computer infrastructure common in modern robotics applications. In particular, this includes ensuring event data consistency between concurrent activities via *circular buffers* and optimizing execution efficiency by exploiting *data locality* and *cache awareness.*

## 2 Related work

The coordination of components is only one of the necessary "concerns" that large-scale "cyber–physical" systems must deal with. It fits into the broader context of the "5Cs" approach of making systems-of-systems software architecture (Bruyninckx, 2023; Klotzbücher et al., 2012; Radestock and Eisenbach, 1996; Vanthienen et al., 2014). The five parts of the 5Cs meta model are

- Computation: the functional behavior inside each activity.
- Communication: the data exchange behavior between activities.
- Coordination: the decision making behavior in and between activities.

- Configuration: adapting each activity's behavior to the actual context.
- Composition: the integration of the previous four parts at the "levels" of activity, component, system, and system-of-systems architecture.

Each of the first four "Cs" can, in itself, be a full or partial sub-system of the "5Cs". A very established pattern within the coordination "C" is that of the life cycle state machine (LCSM), responsible for the "top-level" coordination *inside one single activity*: to create, to start up, to execute, to pause, to reconfigure, and to shut down activities (and the resources they manage) in predictable and composable ways. One single robot will have many activities (sensing, control, world modeling, task execution, etc.), each with its own LCSM, and the focus of this paper is to explain how to maintain the coordination between all these LCSMs, which is where the Petri nets come into play.

Petri nets have been widely used for *modeling concurrent activities/processes* (e.g., to analyze the concurrency behavior of several activities with respect to deadlock analysis or reachability analysis), and their *implementations* come in various forms depending on the use case context in which they are deployed. The implementation proposed by Davidrajuh (2010) has been widely used with MATLAB integration for Petri net modeling, simulation, and performance analysis. In the case of generalized stochastic Petri nets, the implementation proposed by Dingle et al. (2009) provides an open-source tool for design and analysis. The TINA toolbox (Berthomieu et al., 2004) offers a broad set of tools for the construction and analysis of Petri nets and timed Petri nets, which has been extensively used in academia. IOPT-Tools (Pereira et al., 2022; Gomes et al., 2010) provide a framework for the automatic generation of controller code from a modeled Petri net. Developments toward the implementation of Petri nets for microcontrollers have been researched by Kučera et al. (2020), providing a framework to model timed interpreted Petri nets to be used in Arduino devices.

While these implementations provide frameworks to work with Petri nets for different purposes, they are not focused on optimization for low-latency execution. This focus is a primary motivator for the research presented in this paper because modern robotic applications must coordinate several activities such as control, perception, world modeling, and task monitoring, many of which expect *real-time determinism* (Abdellatif et al., 2013). Piedrafita and Villarroel (2011) analyzed the execution dynamics of four different Petri net *software* implementation techniques, whose performance is evaluated with the same Petri net models as in this paper.

For robotics applications, Ziparo et al. (2011) used Petri nets as models for multi-body and multi-robot execution and planning. Their modeling within a multi-robot context is analyzed by Costelha and Lima (2007), investigating deadlocks and reachability. Figat et al. (2017) and Figat and Zieliński (2022) focused on, respectively, hierarchical finite state machines and Petri nets. Zhou et al. (2017) used a hierarchical FSM for the control of a navigation base with a manipulator, where one FSM is embedded into a higher FSM. Lacerda and Lima (2019) generated Petri nets for the coordination of a fleet of robots according to the time logic constraints of the coordinated execution.

## 3 Methodology

The focus of this paper is on three of the "5Cs" software concerns:

- *Coordination* : managing the interactions between a (possible large) set of concurrently executing *activities* using flags, events, finite state machines, and Petri nets as the sufficient mechanisms.
- *Configuration*: allowing application developers to steer the *execution efficiency* of their applications: 1) the *pre-processing* of data structures used by the coordination primitives at runtime and 2) the *event firing and handling* mechanisms that each *coordinated activity* needs to interact with the *coordinating activity.*
- *Communication*: facilitating the exchange of *events* between the finite state machines in the *coordinated activities* on the one hand and the *coordinating mediator*'s Petri net on the other hand.

In addition to the *separation of concerns* (Dijkstra, 1982) that already come with the "5Cs" approach, this paper adds other separations of concerns pertaining to the design of the *inside* of the relevant "5Cs" components. More concretely, the design of the *data structures* and *operators* needed to implement the envisaged coordination mechanisms.

### 3.1 Coordination mechanisms

The mechanisms needed for the coordination of activities are conceptually very simple: *flags*, *events*, *Petri nets*, and *finite state machines* (Section 1.2).

A *finite state machine* (Hrúz and Zhou, 2007; Mealy, 1955) models the *discrete behaviors* of one *single activity*. Its four data structures are the *sets* of 1) *states* that the activity can be in, 2) *transitions* that are allowed between states, 3) *events* that can trigger *transitions*, and 4) *flags* whose status is linked with (a subset of) the *events*. The latter is added to the *mathematical* representation of an FSM in Section 1.2 to allow the *interaction* between an FSM and a Petri net. Its functions are 1) to process the list of available *events*, 2) to compute which *transition* each of those events will trigger (when processed in order of arrival), and 3) to adapt the above-mentioned data structures accordingly.

From a *software implementation* point of view (but *not* from a semantics point of view), finite state machines are just a boundary case of Petri nets: the former has a constraint on the number of "tokens," namely, *exactly one* in the whole set of "states." Figure 2 shows an example of the mapping of an FSM to an equivalent Petri net.

So, this paper focuses on the software design of Petri nets because that of finite state machines differs only in the *configuration* of the resulting library and the *naming* of the implementation primitives. A Petri net model shares the four above-mentioned building blocks with a finite state machine model, but it uses the following specific terminology: a *place* that can contain zero or one *token* as a *marking*, a *transition*, and a *directed arc* between them. The constraint on an arc is that its start and end must be either a place or a transition; in other words, places are only connected

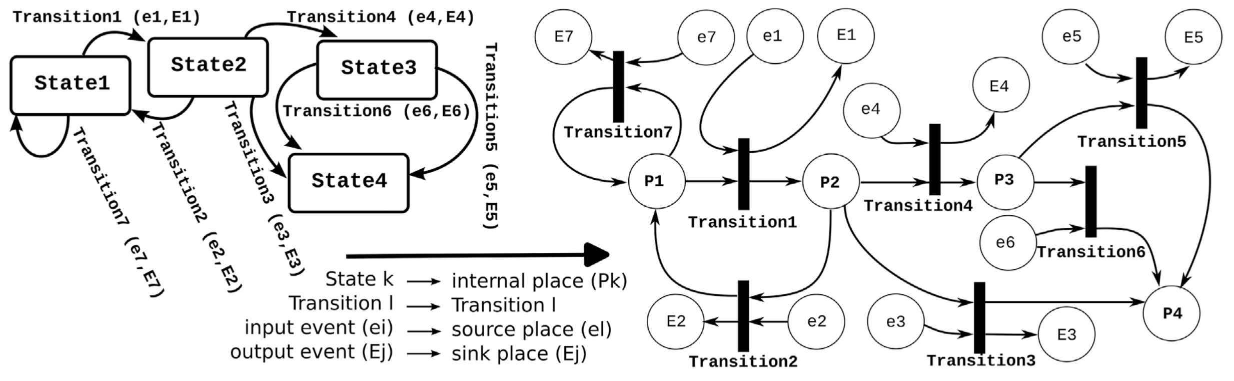


FIGURE 2
A *finite state machine* and the mapping to its equivalent *Petri net*. This mapping constrains the Petri net to have only one connector between any *internal place* and the transitions connected to that place. All other places map to “sink” or “source” events; the “source” places are denoted with small letters, and the “sink” places are denoted with capital letters. A similar typographical convention is used for input and output events in the finite state machine.

to transitions and vice versa. The constraint of a maximum of one token per place is what Murata (1989) referred to as “finite capacity nets of capacity one for all places”; other works of literature call it “safe Petri nets” (Barylska et al., 2017). A *transition* represents a coordination point in the Petri net: its *input places* represent the conditions to be fulfilled for that synchronization to take place; and its *output places* represent the status changes triggered by the coordination.

In addition to the above-described *data structures*, the Petri net mechanism also has some *operators* (“behavior”) on these data structures. If each input place of a particular transition has a token, that transition is *enabled*, and *firing* a transition implies that the tokens in its input places are *removed* and the tokens in its output places are *filled*. The token in the *source places* is to be filled by the processing of an *event* that comes from “somewhere.” Similarly, removing a token from a *sink place* gives rise to sending an event “somewhere.” The links with that “somewhere” are discussed in the following section on “communication.”

Notably, in Figure 2, the FSM and Petri net represent the same process; however, throughout the paper, this is not the case. FSMs are used for the discrete behaviors of single activities, while the Petri nets are used for the coordination across activities. This means there is a match among the FSM states of the coordinated activities and Petri net places of the coordinator; however, they do not present the same process. The latest is illustrated in Figure 3.

## 3.2 Communication mechanisms

The finite state machine in each of the *coordinated activities* exchanges *events* with the *coordinating mediator*’s Petri net (Figure 3). This is reflected in the structure of the Petri net as follows:

- Some input places of transitions do not have any transitions for which they are output places, e.g., p1, p3, and p4 in Figure 3; these are called *source places*. Source places are filled in by the arrival of events to the owner of the Petri net activity. In Figure 3, a token is added to source place p1 when external event 1 (E1) is processed.
- Similarly, *sink places* do not have any transitions for which they are the input places, e.g., P2, P5, and P6 in Figure 3. Sink places trigger the sending of events from the owner of the Petri net activity to other connected activities. In Figure 3, sink place P2 causes the triggering of the internally generated event 4 (e4).

Source and sink places are the locations where the Petri net is connected to *events* from and to the “outside world.” *Internal places* are all other places.

The contribution of this paper with respect to *communication* pertains to the introduction of the *protocol* data structure: it *decouples* the *internals* of the finite state machines and Petri nets from the *communication* of the information they need for their coordination.

The protocol contains information regarding which of the two activities involved in the coordination is expected to set the next *flag* in the protocol. This document uses *arrays* as protocol data structures since they are the simplest approach needed to realize the following goal:

- Only those events that a coordinated activity or the coordinating activity generates or reacts to “end up” in the protocol data structure. These are the events that need to be shared between them.
- The protocol introduces a *hard constraint* in the order in which these events are allowed/expected to be generated; Figure 3 represents this order by the “snake-like” trajectory through the protocol data structure. In order *to guarantee* the correct execution of the coordination, both coordinated parties must satisfy these hard constraints in the *sequence* in which the relevant events are generated or reacted to by the finite state machine and in which the sink and source places are marked in the Petri net.

A flag can be set directly by an activity, or it is the result of processing an event received from that activity. Because of the strict order brought by the protocol, there is no risk that this asynchronous access to the data introduces inconsistency.

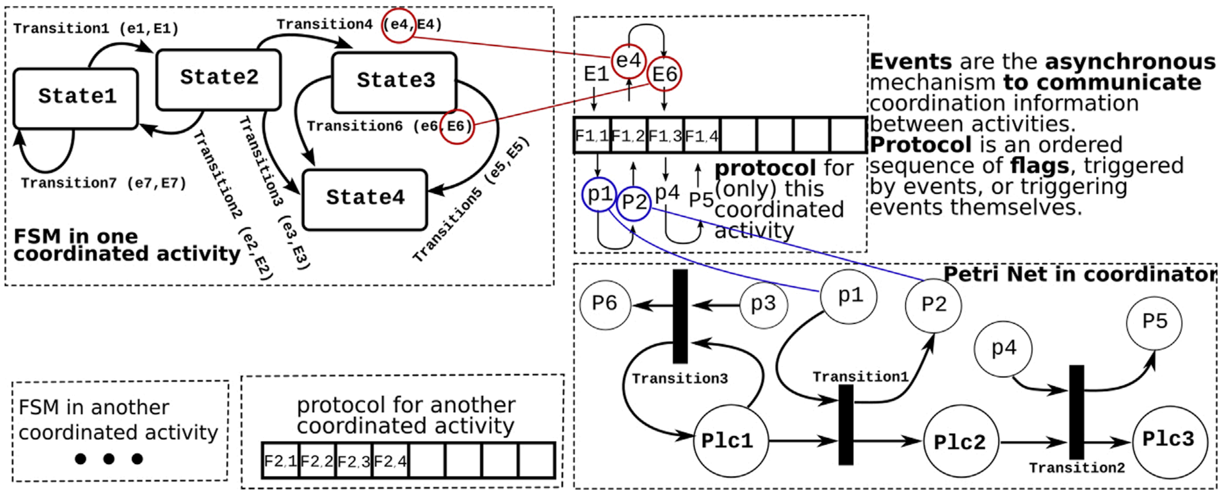

Transition1 (e1,E1)
State1
State2
Transition4 (e4,E4)
State3
Transition6 (e6,E6)
State4
Transition7 (e7,E7)
Transition2 (e2,E2)
Transition3 (e3,E3)
Transition5 (e5,E5)
FSM in one coordinated activity
E1
e4
E6
F1.1 F1.2 F1.3 F1.4
p1 P2 p4 P5
protocol for (only) this coordinated activity
Events are the asynchronous mechanism to communicate coordination information between activities.
Protocol is an ordered sequence of flags, triggered by events, or triggering events themselves.
Petri Net in coordinator
P6
p3
p1
P2
P5
p4
Transition3
Transition1
Plc1
Plc2
Plc3
Transition2
FSM in another coordinated activity
• • •
protocol for another coordinated activity
F2.1 F2.2 F2.3 F2.4

FIGURE 3
Examples of the three software mechanisms needed in *interactivity coordination*: A Petri net inside a *coordinating* activity, a finite state machine inside each of the *coordinated* activities, and (an array of) flags for the bookkeeping of which *coordination "events"* have been communicated between both. Capital letters are used for *output events* in the finite state machine and for *sink places* in the Petri net. The colored lines link events and places to locations in the protocol array. The "snake-like" trajectory through the array represents the temporal order in which the "communication" takes place between finite state machine events and the marking of places in the Petri net.

## 3.3 Configuration mechanisms

This section introduces three software patterns that provide the mechanisms needed *to configure* the coordination between activities. The patterns themselves are not explained in detail because that part of the authors' research is beyond the scope of this document. However, they are in use in the experimental demonstration in Section 5. Each of these patterns works at a different *time scale* in the coordination interaction:

- *Semantic registration* ("long term"): an activity that needs to be coordinated is registered (by itself or by a "third party") for a particular coordination using a *semantic ID.* This ID is a *symbolic* unique identifier used in a *model* of the coordination and, hence, can be retrieved from *persistent storage* or *inter-process communication.*
- *Symbol table data structure* ("medium term"): it links the *semantic ID* symbol to a (possibly variable) number of "resources" or "components." The table facilitates the discovery, communication, execution, and introspection of the "resource" at runtime, which can also be done by activities that have been developed independently.
- *Acquire–release* ("short term"): this pattern structures access to a shared resource by expecting the resource-using activities *to acquire* access from the resource-owning activity and *to release* their granted access explicitly.

The *registration* puts the semantic ID into a table (or a "map") with (at least) the following columns:

- The *semantic ID.*
- The *name* of the coordinated activity, as used in the *source code* of the implementation.
- The *binary pointer(s)* to the memory where the coordination data structure(s) are stored.

TABLE 1 **Example of a table for *registering* the access of activities to shared resources. This particular example uses a *mutex* to coordinate the access to data structures encoder_t and motor_t, shared by three activities in a robot, control, proprioception, and drive.**

| Semantic ID | | | | |
|---|---|---|---|---|
| Datatype | Model | Pointer | Mutex | Activity |
| encoder_t | Left | 0x0a00 | 0x0a08 | Drive and proprioception |
| encoder_t | Right | 0x0a10 | 0x0a18 | Drive and proprioception |
| motor_t | Left | 0x0a20 | 0x0a28 | Drive and control |
| motor_t | Right | 0x0a30 | 0x0a38 | Drive and control |

Table 1 shows an example of such a symbol table. The semantic ID itself has two fields, datatype and model. There can be multiple semantic IDs with the same model label, but the tuple (datatype, model) must be unique. Multiple activities can access the same variables, and coordination is done via mutexes.

The above-mentioned mechanisms are needed for the following reasons:

- *Unambiguous ownership*: registration *implies* that there is a "shared object" to register to and that the system developers should make one, and only one, *activity* the responsible "owner" of that object. (The "owning" object can be a fully passive library and need not be an activity in itself.)
- *Runtime reconfiguration*: because registrations are objects with a lifetime, they can have a life cycle state machine on their own. This is important to coordinate the reconfiguration of

the "object" at runtime and between a changing number of registered activities.

This paper focuses on this short-term time scale, hence, on a low-latency implementation of the *acquire–release* protocol. The "objects" in this paper are coordination objects, specifically Petri nets, and the scope of the presented research calls for Petri nets to be created at runtime. For example, in manufacturing or logistics cases, dozens of shared resources occur, to which, at *any time*, two, three, or more robots want access, and those robots can be different ones every time.

## 4 Implementation

The focus of the paper is on the software mechanisms that are used to realize coordination between a (possibly large) set of concurrently executing activities. The Petri net model plays a central role within the coordination mechanisms presented in the last section. Therefore, an implementation with the purpose of multi activity coordination is presented.

This paper's *design drivers* of the *implementation* of the *design* discussed in Section 3 are typical for *embedded* systems: low-latency and asynchronicity within a shared memory deployment. The presented design is not claimed to be efficient for other use cases, such as the *offline analysis* of Petri nets in search of deadlocks, livelocks, starvation, etc.

One implementation decision is easy to make: while *finite state machines* and *Petri nets* are two complementary coordination mechanisms at the *conceptual* level, their *implementations* are extremely similar; both need "states" and "transitions," with incoming "events" as triggers of the evaluation of the mechanism, as well as the evaluation's possible outcomes. Figure 2 explains the direct mapping of a finite state machine into the equivalent Petri net, so this section restricts itself to the implementation of Petri nets only.

This summary from previous sections is behind the other implementation decisions:

- Petri net models are expected to be *generated at runtime* from symbolic *models.* This allows the use of data structures that can exploit the knowledge of the *number* of *places*, *transitions*, and *events.*
- Petri nets are expected to be *executed in an event loop* of *real-time* activities (Samek and Ward, 2006). This allows a "5Cs" design that *pre-empts* the execution when a *maximum number* of transitions, places, and/or events have been processed, with a known impact on the latency this introduces.
- The *coordinated activities* typically run *asynchronously* with the *coordinating activity* (that is, the one that executes the Petri net). Hence, measures have to be taken to guarantee *data consistency.* This implementation provides two of these measures: *memory barriers* with acquire and release semantics (Standardization committee C and C++, 2017) and *circular buffers* for *wait-free* exchange of events (Desnoyers and Dagenais, 2012; Varghese and Lauck, 1987).
- The target applications are expected to be *always on*, so all of the above-mentioned features must be *(re)configurable.*

### 4.1 Data structures

Figure 4 shows the data structures to represent and execute Petri nets. The data structures above will always be accessed *synchronously* within only the *Petri net executor* activity. The efficiency is designed for the following execution use case:

- Computation of the status changes: the Petri net's status is updated as soon as the activity reacts to incoming events. The events are received asynchronously by the Petri net executor activity (in the *communication* part of the activity's event loop, Supplementary Appendix SA), and our design uses *circular buffers* for this purpose. Circular buffers are also used inside the synchronous part to encode the "to-do lists" of places and transitions that need processing based on the incoming events. The buffers make use of *memory barriers* (of the *acquire-release* type, as provided by the *concurrency support* part of the C/C++ standard libraries) in the trade-off between efficiency of execution and the consistency of data. The latter is a concern to be dealt with by the *application* developers and is introduced by *out-of-order execution* optimizations in modern compilers and CPUs.
- *Data locality*: the data structures needed in nearby moments in the computations are stored in nearby bytes in physical memory. So, *cache coherence* is optimized in two complementary ways:
    - Minimally sized data structures to keep their status. For example, when there are $N$ places, one needs only $M$ 8-bit bytes, where $8 \times M$ is the smallest number larger than $N$. For example, when there are less than 255 places in a Petri net, one char is enough. Such low numbers are not exceptional in the use cases of this paper because access coordination is almost always very local and between a low number of coordinated activities.
    - All data structures are *arrays* of the same type. This reduces the need for *padding* between non-homogeneous parts in the data structures and, hence, indirectly their size as well.
    - The individual data structures are all *cache line aligned* to avoid *cache trashing.*
    - *Arrays instead of linked lists*: the *semantic IDs* of the representation of places, transitions, and events are mapped to unsigned integers ranging from 0 to an *a priori* known integer value $N$. These integers can then also serve as *indices in arrays* so that the inefficient search through lists is replaced by efficient direct access into the arrays.

This section uses teletype font, like this, to represent data structures and operations that are used in the software implementation of this paper's concepts. The following data structures represent the *structure* of a Petri net (Figure 4, left):

- place_to_transitions: this is a *map* (or *symbol table* or *associative array*, Section 3.3) to quickly find the *output transitions* of a place with a given ID. It contains i) pointers bi in an array place_to_transitions_pointer to the binary representation of the transition with a given ID and ii) an array place_to_transitions_number containing the number of transitions for the referenced place.

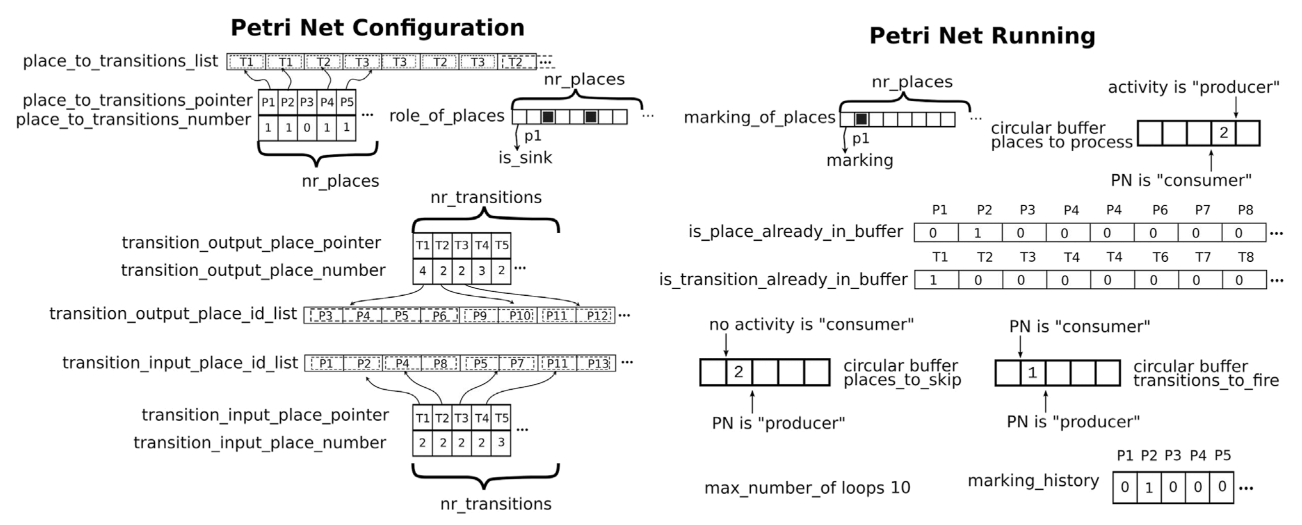


FIGURE 4
Overview of the data structures used in this paper's implementation of Petri nets. Left: *to represent* a Petri net. Right: *to execute* a Petri net. Both sets can be *(re)configured* at compilation time or runtime.

- transition_input_place and transition_output_place: similar to place_to_transition, these maps allow quick access to the output and input places of a given transition.
- sink_places: an array of bits in which each bit represents whether the place is a sink. There is no need to encode whether a place is a *source* or an *internal* place as their behavior does not impact the synchronous execution of the Petri net.

The following data structures represent the *synchronous execution status* of a Petri net (Figure 4, right):

- marking: similar to sink_places, this bit array encodes which places are marked and are, hence, candidates to be processed next.
- places_to_process: this circular buffer (fully inside the *synchronous* context of the coordinating activity) represents the *to-do* list of the IDs of places that must still be inspected to detect *enabled* transitions. In addition, the size of this array can be kept minimal, given the knowledge of the number of places. It also does not make sense to put one particular place more than once on this *to-do* list.
- places_to_skip: this circular buffer represents the list of the IDs of places that still have to be processed, but whose processing has been postponed until the next run of the event loop. Because of the *event loop* context and the *deterministic low-latency* driver, the system developers can decide to limit the number of places on the to-do list that will be processed in each run of the event loop and the number of times such processing is done. This approach provides a *configurable trade-off* between reactivity and deterministic execution time via the configuration variables below.
- is_place_already_in_buffer: these bit arrays remember whether a place is already being checked to avoid the repetition of processing within the same execution loop.
- marking_history: this array of $L$ unsigned integers contains counters indicating how many times each place in the Petri net *has been processed* during this event loop execution.
- max_number_of_loops: this defines the maximum number of times a place *can be processed* per event loop execution before loop's execution is preempted.
- transitions_to_fire and is_transition_already_in_buffer: these serve similar functions to places_to_process and is_place_already_in_buffer but are used for processing of transitions instead of places.

In order to reduce the *cache missing* latency when accessing all these data structures, they should be aligned on cache lines, including padding the last needed cache line with empty bytes.

## 4.2 Discussion

The presented design aims to improve execution latency at the cost of some extra memory in the data structures:

- The data structures place_to_transition and transition_input_place *both* encode the connection of outgoing arcs from places to transitions of the Petri net. This redundancy in memory allows faster lookups in the Petri net execution loop.
- The IDs to process in the circular buffers transitions_to_fire and places_to_fire correspond one-on-one to the flags marked in the status buffers is_place_already_in_buffer and is_transition_already_in_buffer. Every time a new entry is added to the circular buffers, it is also added to the status buffers. This is, strictly speaking, redundant information, but this redundancy yields fast verification of what is already in the *to-do lists*, hence avoiding repeated processing of the same data.
- A similar motivation is behind the design of the data structures sink_to_events and sink_index, which also contain redundant information about the mapping from place ID to sink ID.

Installation instructions, examples, and the code for the implementation of Petri nets explained in this paper are available in[1].

## 4.3 Results for generation and execution performance

To evaluate the execution time of the previous Petri net implementation, five Petri net models presented by Piedrafita and Villarroel (2011) have been built in the library. The range for scaling the size of the Petri nets is taken from the same reference. The Petri net models built were as follows:

- SEQ: Petri nets of *p* sequential processes.
- PR1: Petri nets of *p* sequential processes with two states and one shared resource.
- P1R: Petri nets of one sequential process with *p* resources.
- PH: Petri net of the philosophers' problem with *p* philosophers.
- SQUARE: Petri nets of *p* sequential processes with *p-1* resources.

Within this context, two tests were performed: 1) performance test with immediate firing of one transition; in this case, the execution time of 2,000 triggered transitions is measured. 2) Test with immediate firing of all transitions in the net; this type of test is expected as it marks the maximum reaction time for the complete evaluation of the Petri net. In the latest test, all the transitions of the Petri net will be enabled and triggered in each loop as the Petri net is saturated. The execution time is measured for 2,000 loops for each Petri net. The tests have been run on an HP ZBook Firefly 14 G7 Mobile Workstation.

The X-axis in Figures 5, 6 marks the computation time, while the Y-axis is the scaling parameter, which denotes the number of sub nets in the Petri net [as described by Piedrafita and Villarroel (2011)]. Figure 5 (left) presents the generation time for the Petri nets. The generation time comprises both memory allocation for the data structures in Figure 4 and its initialization. For the Petri net models SEQ, PR1, PH, and P1R, the allocation time is dominant over the initialization time, making the generation time stable within the order of nets tested. In the case of SQUARE, as the size of the net scales quadratically, the initialization time dominates.

Figure 5 (right) shows the performance of the execution of firing one transition per Petri net evaluation. In the case of SEQ, PR1, PH, and P1R, the execution time does not escalate with size, as the number of filled outgoing places from transitions is constant. In the case of SQUARE Petri nets, as the scale factor increases, the number of places to be filled after triggering a transition increases proportionally. Figure 6 shows the execution of saturated Petri nets. The execution time grows linearly for the Petri net models SEQ, PR1, PH, and P1R. This is expected because the number of evaluations is proportional to the number of places in the net. With the same logic, the time for the SQUARE Petri nets grows quadratically with respect to the scale parameter.

As the Petri nets are saturated in the second set of tests (Figure 6), the time in the graphs is taken as an upper bound for the processing time of the Petri net. For instance, a sequential Petri net with 20 processes can take up to 722 ns (1.44 ms/2,000) in the case of all processes coordinated from a mediator.

1 https://gitlab.kuleuven.be/u0141779/coordination_library.git

## 5 Experimental validation

The design and best practices proposed in this article were applied in an experimental setup with two autonomous mobile robots (AMRs) operating in an area with a pre-defined traffic layout. The demonstration case is an artificial scenario of an emergency AMR entering an area with an AMR operating at a lower speed. According to the situation, the slower robot has to reconfigure its execution at discrete and continuous levels in order to let the emergency AMR overtake. Moreover, for the coordination in the shared area, a mediator is introduced to ensure the execution of the synchronization of the AMRs.

### 5.1 Robot setup

Figure 7 shows one of the identical mobile platforms and the 5C activity components running on the onboard computer. Each platform is equipped with an active KELO drive 100, a Hokuyo URG-04LX LiDAR Sensor, and an ODROID XU4 Embedded Computer. In each robot, the following activities are running:

- *Mobile platform drive*: this receives sensor data and transmits wheel setpoints via EtherCAT to the KELO wheel drive.
- *Proprioception*: this estimates the relative motion of the vehicle using wheel encoders (dead-reckoning).
- *LiDAR*: this captures range data via a serial interface from the Hokuyo URG-04LX.
- *Navigation*: this detects and tracks features in the environment (perception) and computes the steering and forward speed commands to perform a desired maneuver (control).
- *Adaptive free-space motion tube*: this evaluates and adapts the control commands provided by the navigation activity to ensure that the vehicle moves within the free-space.
- *Control*: this transform control commands (steering and speed) to KELO wheel setpoints.
- *Communication*: exchanges data with other processes.

These seven activities are registered in five threads (represented in different colors in Figure 7) running at different frequencies. The five threads run in a single multi-threaded process, which allows for efficient in-memory data exchange among the different components. Figure 7 shows some examples of data shared between the activities in colored circles. For that, the variables (objects) need to be first registered in the symbol table with a semantic ID (name and datatype) by the activity owning the resource, e.g., "LiDAR measurements," range_scan_t is registered by the LiDAR activity. For access to a shared variable, first, an activity requests the data pointers corresponding to a particular semantic ID (configuration) from the symbol table.

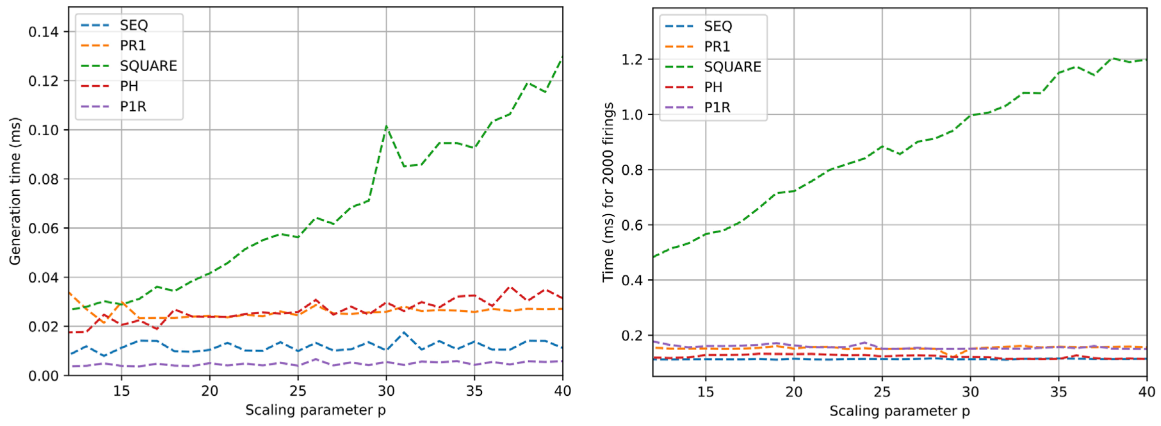


FIGURE 5
Generation (left) and execution (right) time results for different Petri net models. Execution time refers to transition triggering in the net 2,000 times.

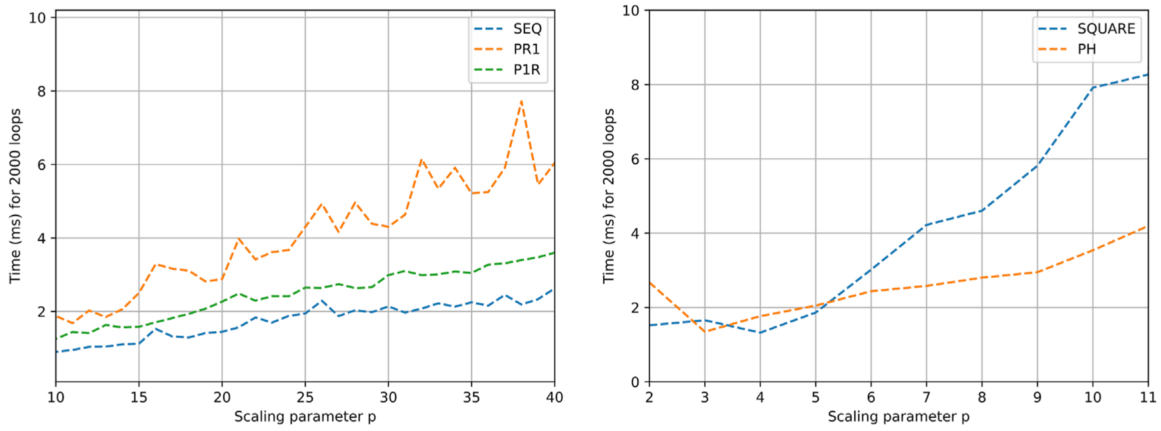


FIGURE 6
Time results for different Petri net design executions. Execution of all transitions enabled in the net 2,000 times.

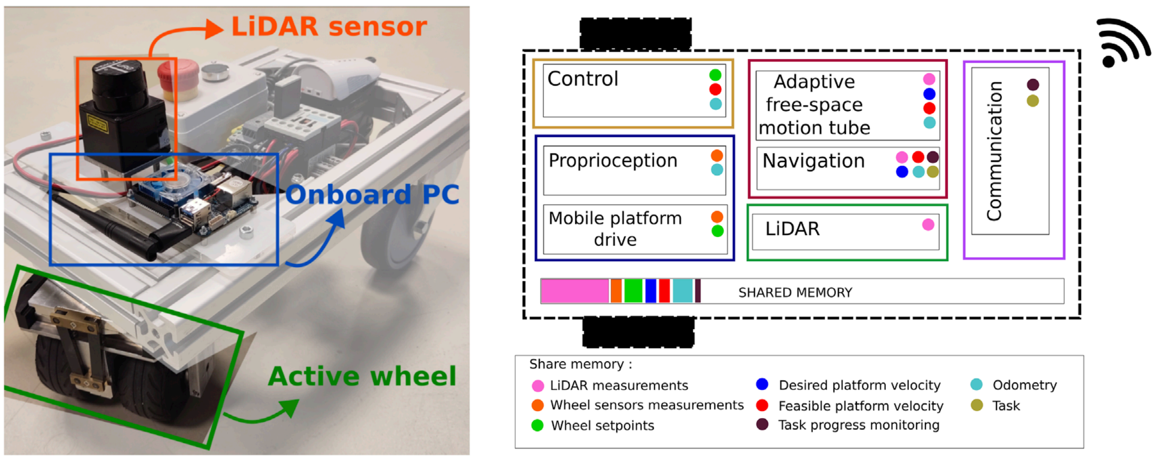


FIGURE 7
Mobile platform, hardware view (left), and thread and activities running on the onboard computer (right). Activities running in the same process exchange data via shared memory. Some of the data chunks accessed via shared memory are illustrated in small colored circles.

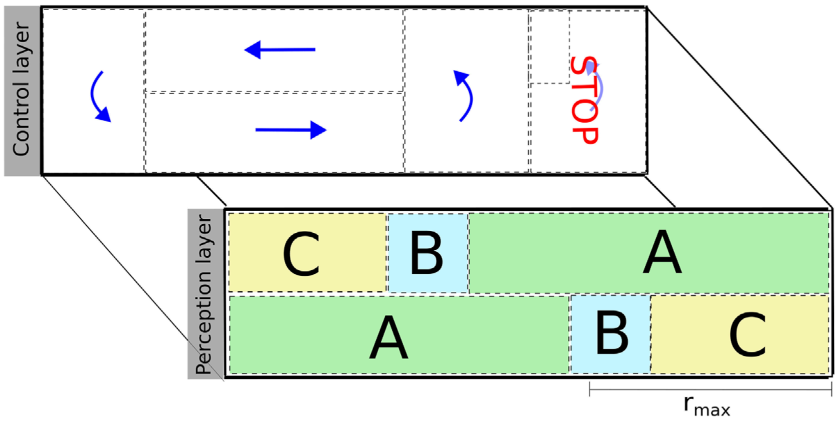


FIGURE 8
Illustration of the control and perception layers of the semantic map designed for the experimental validation. These layers encode the expected control and perception behaviors of the robot within a particular area. Both layers have several monitors associated with them for triggering the coordination mechanism and reconfiguring the schedule of the navigation activity of the platforms.

After that, it can read the values (using acquire/release) directly from the memory without going through the symbol table (communication).

The traffic layout consists of semantic areas that are anchored in environmental features perceived by the robot (corridor and dead-ends). Figure 8 shows the control and perception layers of the semantic map. A solid black box around the map indicates solid walls detected by the LiDAR, while dashed black lines limit semantic areas in each of the layers. The control layer indicates the maneuver that a robot is expected to perform: move forward, make a U-turn, or stop. It also encodes constraints such as limits for driving velocity and deviation from the lane. The perception layer shows the feature that the robot has to track in different colored rectangles labeled as “A,” “B,” and “C.” For example, in area “A” (green), the robot resorts to a corridor detection and tracking algorithm for estimating its relative orientation and lateral position with respect to the corridor. In area “C” (yellow), the robot also tracks its relative longitudinal position with respect to the end of the solid wall at the end of the lane. The reason for the different perception behaviors is due to the finite range of the sensor, which is limited by $r_{\text{max}}$. The robot does not continuously search for the solid wall at the end of the lane but only when it reaches area “B” (blue).

The schedule of the navigation activity links together perception, control, and monitoring algorithms in the form of a skill. The schedule of the activity changes at runtime according to the situation due to coordination and (re)configuration. For example, the robot starts in a known location of area “A” and moves around the circuit. A monitor that uses the information provided by dead-reckoning detects that the robot has reached area “B.” The schedule of the navigation activity changes: the algorithm for detecting the end of the lane is added to the schedule, along with a monitor that checks whether the quality of the estimation is stable. When the estimation is stable, the schedule of the navigation activity changes once again by adding (end-of-lane controller) and removing (corridor controller) algorithms accordingly.

## 5.2 Coordinator setup

For coordination purposes in the semantic area, an area manager is introduced for registering robots in an area and sending events to the robots when necessary. These events will trigger the (re)configuration of the schedule of the robots. The area manager is a multi-threaded process running on a different computer. It has two activities composed according to the 5C paradigm:

- *Area management*: this keeps track of the coordination state of the area and coordinates the robots if required.
- *Communication*: this binds and starts the communication with the robots. It is connected through shared memory to the area management activity. It shares a queue with the updates (task progress monitoring) from the robot and sends commands (tasks) from the area management activity in its event loop.

In this experiment, the execution of commands from both robots is not coupled, meaning that the autonomous execution of each robot does not implicitly change according to other robots in the area. Instead, the area manager works as a mediator between the two robots, coordinating them.

- The area manager makes the decisions on the interaction behavior: the interaction of the bases with their shared resource (space) is set by the mediator, giving access to the areas it manages through events.
- The area manager decouples execution: the area manager is the only “agent” aware of the complete state of the coordinated execution at the discrete level by keeping the execution state in a Petri net.
- The area manager allows execution when the robots have incomplete information of the environment: the robots do not detect each other in the experimental setup proposed, which means that the mediator is required to allow the execution in shared spaces without disruption.

There are two acquire–release protocols between the area manager and AMRs. One of which is from the robot to the area

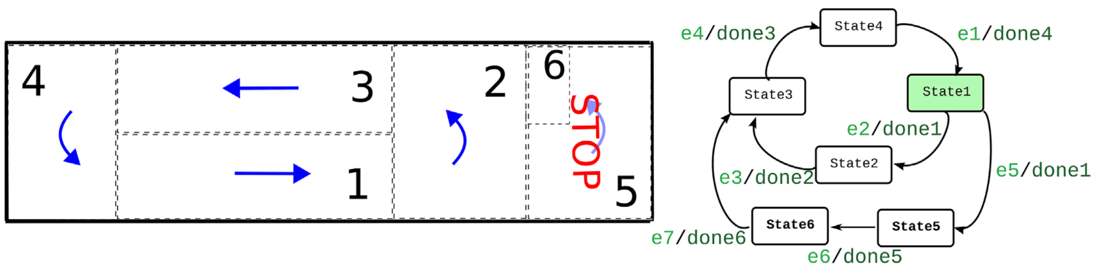


FIGURE 9
Finite state machine of AMRs and navigation map. The states in the finite state machine denote the traversal of the numbered areas according to directional arrows in the map. For example, state 1 would be the navigation in area 1. The input events (light green, at the left-hand side of the slash) among states come from either the navigation or communication activity (from the area manager). The output events (dark green, at the right-hand side of the slash) are triggered by the navigation activity.

manager to access the area. When the robot is navigating toward a local area, it has to request access to the area manager. When access is granted, the semantic ID of the robot and its role (normal or emergency robot) are registered in the list of robots coordinated by the area manager. This list contains the robots that "own" the area (as a passive resource) at a given time.

The area management activity coordinates the interaction of the robots in the area via the following components:

- Area state: the record of the robots that have requested entering an area and releasing an area. In case two robots declare they need to enter the same area, the area management activity sends control requests to the robots.
- Petri net state: when coordination among the robots in the area is required, a Petri net model with the coordination in the area gets initialized. On top of the coordinated states in the Petri net, configuration parameters can be added.

When coordination is needed among the robots in a given area, a second acquire–release interaction is established. The area management activity "acquires" the discrete control of the AMRs and releases it when the coordination is over. This means that, while the robot normally coordinates itself by executing the skills in its FSM depicted in Figure 9, when coordination happens, this is not the case anymore. When coordinated, the management activity takes control of the AMR at the discrete level via the coordination Petri net (Figure 10), with its connected protocols that trigger the sending of events to the AMRs. While the FSM in the robot is still tracking the execution of the robot, it does not trigger the maneuvers. When the coordinated execution is finished, the area management activity releases the AMR execution with a last shared event and deletes the coordination.

The Petri net used in the demonstration is depicted in Figure 10. The color legend of the image explains the ownership of the places, meaning which activity has control over the events to set the marking in the place. The white places are internal places of the Petri net, which denote the state of the AMRs in the execution. The dark places are source places, which are filled in by the coordinating activity once the corresponding event arrives from the AMRs. The light places are sink places to be filled in by the Petri net execution, triggering the sending of events to the AMRs.

For example, Figure 10 shows a case of the sink place "Rae1," to which only the area manager has writing access. Its marking is filled in by the triggering of the Petri net. When the place "Rae1" gets a token, the connected flag "e1" in the protocol with the communication activity is raised. When the flag "e1" is raised, an event is sent to the emergency AMR, which indicates it may enter "Area 1."

A case of the source place is "Radone1." The communication with the emergency AMR has writing access right to the flag "done1" in the protocol, and the area manager activity reads this flag. When the event arrives from the communication activity connected to the emergency AMR, the flag "done1" is raised. Once the flag "done1" is read by the area manager activity, the place "Radone1" gets a token, and the outgoing transitions can be evaluated to continue with the coordinated execution.

The processing of incoming and outgoing events according to the sources and sinks in the coordination structure of the Petri net in Figure 10 allows the execution of the coordinated motions of the robots without disruption. Moreover, apart from the sink and source places, the internal places are added to denote the concurrent state of different activities in the coordination.

## 5.3 Execution of experimental demonstration

The management activity remains idle after initialization until two robots are passed to it (along with a communication channel to them). The robots are passed according to their roles in the coordination: normal robot or emergency robot. The emergency robot has priority over the normal robot. The initial state of each robot is informed to the Petri net, which translates to the initial marking.

Once the coordination is properly configured, the event loop of the management activity starts. In the event loop, the management activity processes the messages coming from the AMRs, updating them on the execution progress with respect to the skill they are performing. The activity updates the marking of the coordination Petri net when the events of finished skills are received. After the marking of the Petri net is updated from all robots, the Petri net is triggered. In the case that all the places of a transition have a token,

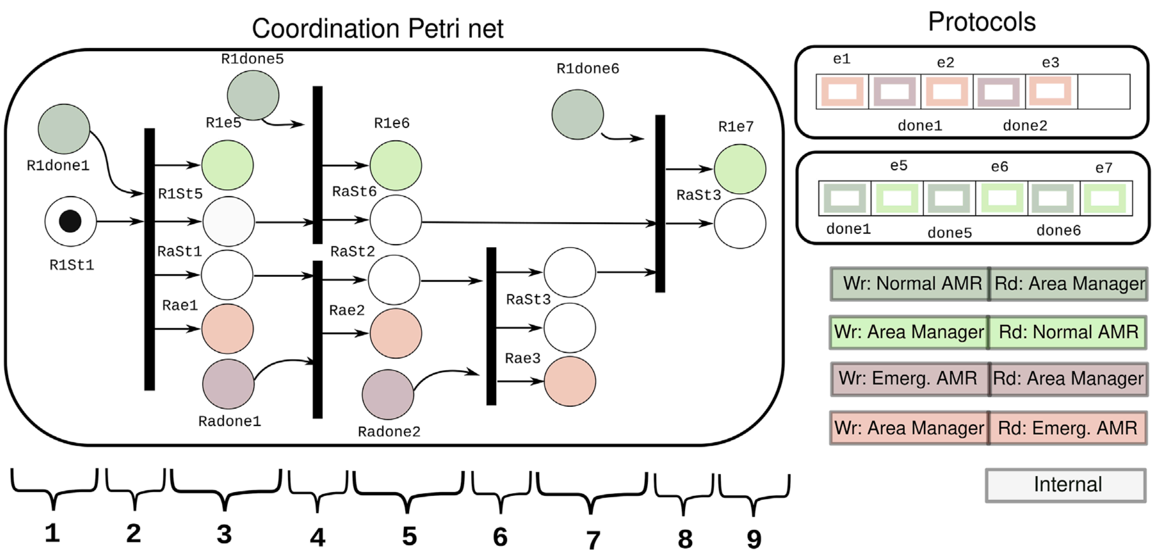


FIGURE 10
Petri net and protocols used for coordinating the normal AMR and the emergency AMR in the demonstration. The color legend shows the reading and writing "rights" for the flags in the protocols. The execution of the coordination at the discrete level according to the coordination in the Petri net: 1) normal AMR crosses area 1. 2) Wait until normal AMR crosses. 3) Emergency AMR is allowed to start its execution in area 1, while normal AMR gets the signal to go to the padding area with higher speed. 4) Wait for normal AMR to be out in area 5 and emergency AMR finishes in area 1. 5) Normal AMR is in padding area while emergency AMR crosses area 2. 6) Wait for emergency to be done in area 2. 7) Emergency AMR can start crossing area 3, and it is released from the coordination. 8) Wait for the normal AMR to go out of the padding area. 9) Normal AMR can start crossing area 3, and it is released from the coordination.

the marking of the Petri net is updated. The updated marking is then passed to the communication modules to send the events coming from the Petri net to the robots.

The area manager is the only activity aware of the coordinated execution but is not responsible for configuring the schedule being executed in the coordinated robots. The change in configuration (e.g., of the normal robot when the emergency AMR is behind) is achieved via events that are sent from the area manager to the robots via the communication channel. These events lead to a change in the configuration of the robots, e.g., emergency AMR needs to slow down because the normal robot is still ahead or the normal robot has to drive to area 5 and wait there.

Once the coordination has finished (the emergency has overtaken the normal robot), the area manager gives back control to the AMRs because there is no need for mediation. Both robots continue their autonomous execution with their initial configuration.

### 5.4 Secondary demonstration: area manager for heterogeneous AMRs

The same area manager as in the previous experiment was deployed in a setup with three heterogeneous AMRs. The demonstration case is the access area to docking stations in a warehouse. In this application, coordination is needed to mediate the access to an area that can be used as two lanes by two small AMRs or one lane for a big AMR. The execution of the coordinated robots and the Petri net used can be seen in one of the videos in the multimedia part of this paper.

## 6 Discussion

The paper's focus is on the *efficient implementation* of runtime coordination needs in multi-robot applications. Most of the efficiency comes from *knowing in advance* (the sizes and types) of all data structures because that knowledge allows making the most cache-efficient and data locality-driven implementations: (almost) linear-time indexing of data pointers, optimal cache alignment, known maximum usage in both time and space, etc. These efficiencies are typically only possible and useful in *embedded* and/or *real-time* software systems. *Single producer*, *single consumer* event queues, and to-do lists are common practices in this context because it is normal "to know everything" about such systems.

This section discusses implementation decisions that system developers have to be aware of to make the best use of the presented design:

- *Semantic registration* of all primitives involved (activities, Petri nets, etc.): while all data structures and operators presented in this paper can be implemented manually from scratch, they are also designed to allow an even partial and gradual development path toward more *code generation*, from models of the coordination mechanisms to executable code. It is important to consider the most advanced version in the applications, i.e., the version in which these models are "downloaded" or "adapted" at runtime and updated code is generated by a running system itself.
- *Symbol tables* for data sharing: they facilitate data sharing among activities running concurrently. This not only helps the above-mentioned code generation but is also useful

in allowing "browsing" or "introspection" of a running application: the symbolic names can then be used to navigate from "component" to "component" and inspect and/or adapt the local values in these components.

- *Acquire–release* pattern: this pattern is used for acquiring access to write and read the variables kept in the symbol table. At the deepest level of detail, the presented implementation already uses this pattern via the *acquire* and *release* semantics of memory barriers. However, a similar approach can be used at all higher levels of detail, such as to connect two robots to a third one at runtime, where the latter is responsible for the coordination of the unique access to one of its resources by the former two robots. For example, the third robot could let the first robot use its pan-tilt camera.
- The Petri net and finite state machine data structures need only to be known at the time of *runtime software configuration* (that is, not necessarily at *compile time* or even *deployment time*) because memory allocation can be postponed until just before the data structures are used. A *memory pool* approach is also a good fit.
- "*Best*" *design of the monitors.* Coordination is, by nature, a *reactive* behavior triggered by *events* that represent that "something has happened." Hence, the efficiency and correctness of the coordination tasks in an application are not only realized by the efficiency and correctness of the coordination mechanisms presented in this paper but also by the "appropriate" design of the *monitors* that are needed to generate the events by monitoring Boolean combinations of status variables that can be spread over several components of the application. Similarly, the events generated by the coordination mechanisms must still be reacted to in an "appropriate" way via decision-making functions in the relevant system components.
- *Simultaneous events.* The application context of this paper is that of *concurrent activities*, each of which can generate multiple events and is expected to react to multiple events. No software design is known *to guarantee* that the *order* in which events end up in each activity's event queue is the same temporal order in which these events were generated. Hence, the *system architects* have the responsibility to introduce coordination logic (in FSMs, Petri nets, and protocols) that is "*appropriately*" *robust* against such order "inversions." To the best of the authors' knowledge, the scientific foundations to generate such robust coordination logic are still to be discovered.
- *Errors in coordination logic.* Even a perfect software implementation of the mechanisms in this paper cannot guarantee that there are no errors in the coordination logic of the application, possibly leading to deadlocks or livelocks in the overall system.

For example, a robot might attempt to enter an area to which it has not yet been granted access to, or it might try to enter another area. System designs can be made more robust by introducing extra monitor activities to detect deadlocks or livelocks when the coordinated activities have not foreseen this monitoring themselves.

- *Hierarchy in coordination.* The coordination examples presented in this paper are "*flat*": there is one Petri net layer added to the robots' individual task controllers. This hides the implicit assumption that the coordinated robots collaborate *only* with the coordinating activity and that the actions that Petri net decides about have no "competition" of decisions made elsewhere. The authors believe that solutions to these problems are *not* to be found in extra software primitives but in using the presented ones in non-flat, application-specific "coordination hierarchies."

One reason for introducing such "non-flat" coordination is to address the *coordination logic errors* of the previous item. In addition to the deadlock/livelock monitors mentioned above, system designs can be made more robust by introducing *pre-emption*: the Petri net and/or finite state machines are extended with places/states that represent a phase in the coordination where that coordination can be pre-empted. In any case, such pre-emption needs coordination itself because all coordinated activities must somehow be brought back into a known and consistent interaction state.

This "hierarchical coordination" topic is beyond the scope of this paper.

## Data availability statement

The original contributions presented in the study are included in the article/Supplementary Material; further inquiries can be directed to the corresponding author.

## Author contributions

MA: conceptualization, investigation, methodology, software, validation, writing–original draft, and writing–review and editing. RR: conceptualization, methodology, software, validation, writing–original draft, and writing–review and editing. LV: software, writing–original draft, and writing–review and editing. HB: conceptualization, methodology, writing–original draft, and writing–review and editing.

## Funding

The author(s) declare that financial support was received for the research, authorship, and/or publication of this article. This work was supported by the Flanders Make projects *AssemblyRecon* ("Decision Framework for Assembly System Reconfiguration"), *HySLAM* ("A Hybrid SLAM approach for autonomous mobile systems"), and *CTO action on Cooperative motions* and by the European Horizon 2020 project *RobMoSys* ("Composable Models and Software for Robotic Systems") under grant agreement No. 732410.

## Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors, and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

## Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/frobt.2024.1363041/full#supplementary-material

## References

Abdellatif, T., Combaz, J., and Sifakis, J. (2013). Rigorous implementation of real-time systems—from theory to application. *Math. Struct. Comput. Sci.* 23, 882–914. doi:10.1017/s096012951200028x

Barylska, K., Best, E., Schlachter, U., and Spreckels, V. (2017). Properties of plain, pure, and safe Petri nets. *Trans. Petri Nets Other Models concurrecncy. Vol 10470 Lect. notes Comput. Sci.*, 1–18. doi:10.1007/978-3-662-55862-1_1

Berthomieu, B., Ribet, P.-O., and Vernadat, F. (2004). The tool tina–construction of abstract state spaces for Petri nets and time Petri nets. *Int. J. Prod. Res.* 42, 2741–2756. doi:10.1080/00207540412331312688

Brugali, D., and Scandurra, P. (2009). Component-based robotic engineering (Part I) [Tutorial]. *IEEE Robotics Automation Mag.* 16, 84–96. doi:10.1109/MRA.2009.934837

Brugali, D., and Shakhimardanov, A. (2010). Component-based robotic engineering (Part II). *IEEE Robotics Automation Mag.* 17, 100–112. doi:10.1109/MRA.2010.935798

Bruyninckx, H. (2023). *Building blocks for complicated and situational aware robotic and cyber-physical systems.* KU Leuven: Department of Mechanical Engineering.

Costelha, H., and Lima, P. (2007). "Modelling, analysis and execution of robotic tasks using petri nets," in 2007 IEEE/RSJ international conference on intelligent robots and systems, San Diego, CA, October 29–Novamber 02, 2007 (IEEE), 1449–1454.

Davidrajuh, R. (2010). *Gpensim: a new Petri Net simulator (InTech).*

Delanote, D., Van Baelen, S., Joosen, W., and Berbers, Y. (2008). "Using AADL to model a protocol stack," in *IEEE international conference on engineering of complex computer systems*, 277–281.

Desnoyers, M., and Dagenais, M. R. (2012). Lockless multi-core high-throughput buffering scheme for kernel tracing. *ACM SIGOPS Oper. Syst. Rev.* 46, 65–81. doi:10.1145/2421648.2421659

Dijkstra, E. W. (1982). "On the role of scientific thought," in *Selected writings on computing: a personal perspective* (Springer-Verlag), 60–66.

Dingle, N. J., Knottenbelt, W. J., and Suto, T. (2009). Pipe2: a tool for the performance evaluation of generalised stochastic Petri nets. *SIGMETRICS Perform. Eval. Rev.* 36, 34–39. doi:10.1145/1530873.1530881

Figat, M., and Zieliński, C. (2022). Parameterised robotic system meta-model expressed by hierarchical Petri nets. *Robotics Aut. Syst.* 150, 103987. doi:10.1016/j.robot.2021.103987

Figat, M., Zieliński, C., and Hexel, R. (2017). "FSM based specification of robot control system activities," in *2017 11th international workshop on robot motion and control RoMoCo (IEEE)*, 193–198.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). Design patterns: elements of reusable object-oriented software

Gomes, L., Rebelo, R., Barros, J. P., Costa, A., and Pais, R. (2010). "From Petri net models to C implementation of digital controllers," in *2010 IEEE international Symposium on industrial electronics (IEEE)*, 3057–3062.

Hrúz, B., and Zhou, M. (2007). *Modeling and control of discrete-event dynamic systems: with Petri Nets and other tools*. Springer.

Klotzbücher, M., Biggs, G., and Bruyninckx, H. (2012). "Pure coordination using the Coordinator–Configurator pattern," in *Proceedings of the 3rd international workshop on domain-specific languages and models for robotic systems*, 1–4.

Kučera, E., Haffner, O., Drahoš, P., Leskovský, R., and Cigánek, J. (2020). PetriNet editor+ PetriNet engine: new software tool for modelling and control of discrete event systems using Petri nets and code generation. *Appl. Sci.* 10, 7662. doi:10.3390/app10217662

Lacerda, B., and Lima, P. U. (2019). Petri net based multi-robot task coordination from temporal logic specifications. *Robotics Aut. Syst.* 122, 1–13. doi:10.1016/j.robot.2019.103289

Mealy, G. H. (1955). A method for synthesizing sequential circuits. *Bell Syst. Tech. J.* 34, 1045–1079. doi:10.1002/j.1538-7305.1955.tb03788.x

Murata, T. (1989). Petri nets: properties, analysis and applications. *Proc. IEEE* 77, 541–580. doi:10.1109/5.24143

Pereira, F., Moutinho, F., Costa, A., Barros, J.-P., Campos-Rebelo, R., and Gomes, L. (2022). "Iopt-tools–from executable models to automatic code generation for embedded controllers development," in *International conference on applications and theory of Petri nets and concurrency* (Springer), 127–138.

Piedrafita, R., and Villarroel, J. L. (2011). Performance evaluation of Petri nets centralized implementation. the execution time controller. *Discrete Event Dyn. Syst.* 21, 139–169. doi:10.1007/s10626-010-0090-7

Radestock, M., and Eisenbach, S. (1996). "Coordination in evolving systems," in *Trends in distributed systems. CORBA and beyond* (Springer-Verlag), 162–176.

Samek, M., and Ward, R. (2006). Build a super simple tasker. *Embed. Syst. Des.* 19, 18–37.

Standardization committee C and C++ (2017). Memory barriers in the C standard. *CPP Reference.com.*

Van Baelen, S., Peeters, G., Bruyninckx, H., Pilozzi, P., and Slaets, P. (2022). Dynamic semantic world models and increased situational awareness for highly automated inland waterway transport. *Front. Robotics AI* 8, 739062–739071. doi:10.3389/frobt.2021.739062

Vanthienen, D., Klotzbücher, M., and Bruyninckx, H. (2014). The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming. *J. Softw. Eng. Robotics* 5, 17–35. doi:10.6092/JOSER_2014_05_01_p17

Varghese, G., and Lauck, A. (1987). "Hashed and hierarchical timing wheels: data structures for the efficient implementation of a timer facility," in *Proceedings of the eleventh ACM symposium on operating systems principles*, 25–38.

Zhou, H., Min, H., Lin, Y., and Zhang, S. (2017). "A robot architecture of hierarchical finite state machine for autonomous mobile manipulator," in *Intelligent robotics and applications: 10th international conference, ICIRA 2017, wuhan, China, august 16–18, 2017, proceedings, Part III 10* (Springer), 425–436.

Ziparo, V. A., Iocchi, L., Lima, P. U., Nardi, D., and Palamara, P. F. (2011). Petri net plans: a framework for collaboration and coordination in multi-robot systems. *Aut. Agents Multi-Agent Syst.* 23, 344–383. doi:10.1007/s10458-010-9146-1

frontiers | Frontiers in Robotics and AI

TYPE Original Research
PUBLISHED 03 October 2024
DOI 10.3389/frobt.2024.1346580

OPEN ACCESS

EDITED BY
Ivano Malavolta,
VU Amsterdam, Netherlands

REVIEWED BY
Jonathan M. Aitken,
The University of Sheffield, United Kingdom
Luciana Rebelo,
Gran Sasso Science Institute, Italy
Miguel Campusano,
University of Southern Denmark, Denmark

*CORRESPONDENCE
Marcela G. dos Santos,
marcela.santos1@uqac.ca

RECEIVED 29 November 2023
ACCEPTED 28 August 2024
PUBLISHED 03 October 2024

CITATION
dos Santos MG, Hallé S, Petrillo F and Guéhéneuc Y-G (2024) AAT4IRS: automated acceptance testing for industrial robotic systems.
*Front. Robot. AI* 11:1346580.
doi: 10.3389/frobt.2024.1346580

# AAT4IRS: automated acceptance testing for industrial robotic systems

Marcela G. dos Santos[1]*, Sylvain Hallé[1], Fabio Petrillo[2] and Yann-Gaël Guéhéneuc[3]

[1]Départment d'Informatique et Mathématique, Université du Québec à Chicoutimi, Chicoutimi, QC, Canada, [2]Département de génie logiciel et TI, École de Technologie Supérieure (ÉTS), Montréal, QC, Canada, [3]Department of Computer Science and Software Engineering, Concordia University, Montréal, QC, Canada

Industrial robotic systems (IRS) consist of industrial robots that automate industrial processes. They accurately perform repetitive tasks, replacing or assisting with dangerous jobs like assembly in the automotive and chemical industries. Failures in these systems can be catastrophic, so it is important to ensure their quality and safety before using them. One way to do this is by applying a software testing process to find faults before they become failures. However, software testing in industrial robotic systems has some challenges. These include differences in perspectives on software testing from people with diverse backgrounds, coordinating and collaborating with diverse teams, and performing software testing within the complex integration inherent in industrial environments. In traditional systems, a well-known development process uses simple, structured sentences in English to facilitate communication between project team members and business stakeholders. This process is called behavior-driven development (BDD), and one of its pillars is the use of templates to write user stories, scenarios, and automated acceptance tests. We propose a software testing (ST) approach called automated acceptance testing for industrial robotic systems (AAT4IRS) that uses natural language to write the features and scenarios to be tested. We evaluated our ST approach through a proof-of-concept, performing a pick-and-place process and applying mutation testing to measure its effectiveness. The results show that the test suites implemented using AAT4IRS were highly effective, with 79% of the generated mutants detected, thus instilling confidence in the robustness of our approach.

KEYWORDS

**robotics, industrial robots, software testing, automated testing, acceptance testing**

## 1 Introduction

According to the International Federation of Robotics, the operational stock of industrial robotic systems (IRS) is increasing. In 2022, the number of robot installations hit a record high, reaching 553,052 units. This marked the second consecutive year with over 500,000 units, reflecting a 5% increase from the previous year and a compound annual growth rate (CAGR) of 7% from 2017 to 2022. The operational stock of industrial robots also experienced significant growth, reaching 3,903,633 units—an increase of 12%—with an average annual growth of 13% over the past 5 years. Despite a slowing global economy, the forecast for 2023 predicts over 590,000 global robot installations (IFR, 2024). The increase in the number of industrial robotic systems operating in the most diverse environments also increases the necessity for these systems to handle failures and meet quality aspects.

Conventional software systems can be defined as "failing" when customers' expectations have not been met and/or when the software does not help the customer (Chillarege, 1996). To prevent this, developers and stakeholders must use automated software testing to improve fault detection in the system, which will be reflected in improved software quality (Naik and Tripathy, 2018).

Our aim is to improve fault detection in IRS by applying a specific software testing approach. Some research has applied software testing in robotic systems (Bretl and Lall, 2008; Estivill-Castro et al., 2018; Mossige et al., 2015; Chung and Hwang, 2007; Erich et al., 2019; Nguyen et al., 2023). In addition, there are some systematic literature reviews and surveys on related topics that we used to clarify the issues and challenge to apply software testing to robotic systems (Afzal et al., 2020; Afzal et al., 2021).

Afzal et al. (2020) conducted a qualitative study on the challenges of testing robotics systems. They identified five testing challenges to writing and designing for robotic systems: unpredictable corner cases, engineering complexity, culture of testing and coordination, collaboration, and documentation. The coordination, collaboration, and documentation challenge according to the authors is "…the lack of proper channels for coordination and collaboration among multiple teams and a lack of documentation."

According to Afzal et al. (2020), one of the significant challenges in **coordination, collaboration, and documentation** stems from the need for adequate channels for coordination and collaboration among multiple teams and for more documentation. Coordination within many robotics companies often requires bridging gaps between teams with diverse backgrounds, such as software and hardware teams. Additionally, it is common to encounter the need to integrate and write tests for third-party components without any accompanying documentation. More standards and guidelines for writing and designing tests for robotic systems must also be developed.

Another challenge identified based on participants' responses concerns the **culture of testing**. One defining feature of the robotics community is its ability to bring together individuals from various disciplines, such as electrical and mechanical engineering. This diversity not only drives significant advances in robotics but also poses certain challenges. For example, a representative quote from one of the respondents is, "The world of robotics unites folks from different backgrounds. Folks from a software background might observe testing differently from those who are not."

Behavior-driven development (BDD) is a software development approach that promotes collaboration between technical and non-technical stakeholders during the development process. It introduces a common language made up of structured sentences expressed in natural language. This language aims to improve communication and understanding between project team members and business stakeholders, resulting in more effective software development and clearer alignment with business objectives (Irshad et al., 2021).

Solis Pineda and Wang (2011) identified six key characteristics of BDD: (i) the use of ubiquitous language based on business terminology; (ii) an iterative decomposition process for high-level specifications; (iii) templates to write user stories and scenarios; (iv) automated acceptance tests; (v) readable specification code; (vi) elaboration of behaviors based on the needs of the development phase. The software testing applied in BDD is automated acceptance testing that emphasizes the validation of a system's performance within the context for which it is intended.

Our objective is to streamline the software testing process for IRS through the introduction of a specialized approach we have developed known as AAT4IRS (Automated Acceptance Testing for Industrial Robotic Systems). This ST approach entails customizing and implementing automated acceptance templates derived from the principles of BDD. By leveraging this approach, we aim to enhance IRS fault detection.

To evaluate our methodology in an industrial setting, we developed a test suite to apply our approach. The scenario involved an industrial robot picking items and placing them into a designated box based on color. Pick-and-place behavior is utilized in almost all industrial environments that use industrial robotics to automate processes. This scenario is similar to that (and the requirements) used in robotics competitions and benchmarks. Our decision to use as our inspiration the requirement used in a robotics competition was because of the absence of public repositories of more real requirements (used in industry) available (Nguyen et al., 2023).

The effectiveness of our approach was evaluated through mutation testing. The mutation score serves as a reliable metric for assessing the efficacy of a test set in identifying faults. Research indicates that achieving a higher mutation score markedly improves the detection of faults (Jia and Harman, 2011; Papadakis et al., 2018). As robotic systems interact with the real world, we created "mutants" to simulate these interactions. Thus, we implemented mutation testing to evaluate our test suite in the context of robotic systems.

The initial results show that the test suite implemented using AAT4IRS was able to kill 79% of the mutants. These results show the effectiveness of a test suite implemented by following AAT4IRS. Despite the benefits observed in the proposed approach, it is still open to improvement. We detail such future possibilities in Section 8.

The main contributions of this study are (i) a software testing approach to apply automated acceptance testing in IRS (Section 3), (ii) an empirical study to evaluate our approach (Section 4), and (iii) an initial guideline to create mutants for industrial robotic systems (Section 5).

The remainder of this paper is organized as follows: Section 2 provides background on the core concepts for our study (industrial robotics systems, acceptance and mutation testing) and related work. Section 6 discusses our results and highlights the observed benefits. Section 7 discusses the threats to validity. Section 8 synthesizes final remarks and future work.

This paper is an extended version of a demo paper (Santos et al., 2022), with the following changes: (i) expanded explanations for the proposal approach; (ii) mutation testing to evaluate it; (iii) initial guideline to create mutants for IRS.

## 2 Background and related work

### 2.1 Industrial robotic systems

An "industrial robotic system" (IRS) is a system composed of industrial robots, end-effectors (grippers, magnets, vacuum heads), sensors (visual, torque, collision detection, 3D vision), and equipment (belt conveyors). As with any robot in general, an industrial robot is a complex system comprising both hardware and software; as such, it can be subject to failure in any of these components. The scope of our study is on the software component, which is composed of two parts. First, the *control layer* is responsible for translating commands so that the IRS can understand and execute them; thus, it is essentially a collection of drivers interacting with the hardware. Second, the *application layer* is composed of a software script which defines the robot's desired behavior according to business requirements (Ahmad and Babar, 2016; ISO/IEC 8373, 2012).

According to Heimann and Guhl (2020), the methods of programming industrial robots can be classified based on the interaction between the operator (who is responsible for programming and operating the industrial robot) and the robot. These methods can be online, offline, or hybrid. In the first *online* method, the operator programs the robot on the shop floor, using either the "lead through" method (the operator takes the robot manually and guides it through a trajectory) or the "teach-pendant" method (the operator guides the manipulator to specific points, records these points to compose a trajectory, and the manipulator executes the trajectory). The shop floor must stop its production from having the IRS programmed in both.

In contrast, in the *off-line* mode the operator uses an environment composed of industrial robot programming languages (IRPLs) and/or simulation software. IRPLs are purpose-built, domain-specific programming languages that include special instructions to move the robot's arm(s), as well as standard control-flow instructions and APIs to access low-level resources (Pogliani et al., 2020).

Finally, in the *hybrid* method, the operator works offline to create the flow and calibrate and validate the physical system (online). The hybrid method thus utilizes the benefits of online and offline methods.

Validation in IRS depends on the method used. When the online method is applied, the operator must stop the shop floor, program the robot, and make it execute the program. The robot is then evaluated to determine whether it had the expected behavior by visual checking and/or with a reading sensor (if some are available in the environment).

In the offline method, the validation process takes place in a simulator; only if the robot behavior fits the business requirement (BR) can the program (i.e., the application layer) be sent to the robot on the shop floor. The validation process uses the operator's knowledge and experience with the expected robot behavior.

In the hybrid method, the validation also used the simulators, after the operator fine-tunes the program using the online method to make real-time adjustments based on the robot's behavior interacting with the physical environment.

In summary, although these are different methods of programming IRS, they have the same aspect concerning validation, which uses the operator's knowledge and expertise about the expected robot behavior. Furthermore, the validation process is manual for each program written. Suppose that there are changes to the BR (modification or addition of a feature). In that case, the operator must change and validate the program again, even if they validated part of the program before the modification. This manual aspect of code validation in an industrial environment increases project time and cost; automated software testing is thus needed for industrial robots.

### 2.2 Automated acceptance testing (AAT)

There are various levels of software testing, one of which is the *acceptance test*. This type of testing aims to check if the system meets a set of acceptance criteria (AC) that guarantee that its quality is suitable for the particular business requirements.

As with other forms of testing, acceptance testing can be applied manually or using automation tools. The benefits of test automation are increased test productivity, better coverage of regression tests, reduced duration of testing phases, reduced cost of software maintenance, and increased effectiveness of test cases. With test automation, an organization can create a rich library of reusable test cases, facilitating the execution of a consistent set of test cases (Naik and Tripathy, 2018).

However, introducing automation creates a gap between the business requirements and technical aspects of software testing. On the one hand, business teams write the BRs and use them in the definition of AC; on the other hand, these requirements must result in the generation and execution of test cases.

To align the business and technical needs of software, behavior-driven development (BDD) can be a good choice (Irshad et al., 2021). In BDD, one typically uses a high-level human-readable language, such as Gherkin (Wynne and Hellesoy, 2012), to bridge the gap between BRs and technical aspects. On the one hand, the document describing the requirements is easily readable by developers, QA teams and business teams. On the other hand, the development team uses the same document to automate the execution of acceptance tests (Nicieja, 2017).

In BDD, the AC for each business requirement is described in two parts: the *feature* and the *scenario*. The feature is a deliverable piece of functionality to allow the business to achieve its goals. It is described using the user story format: "**As a** [description of

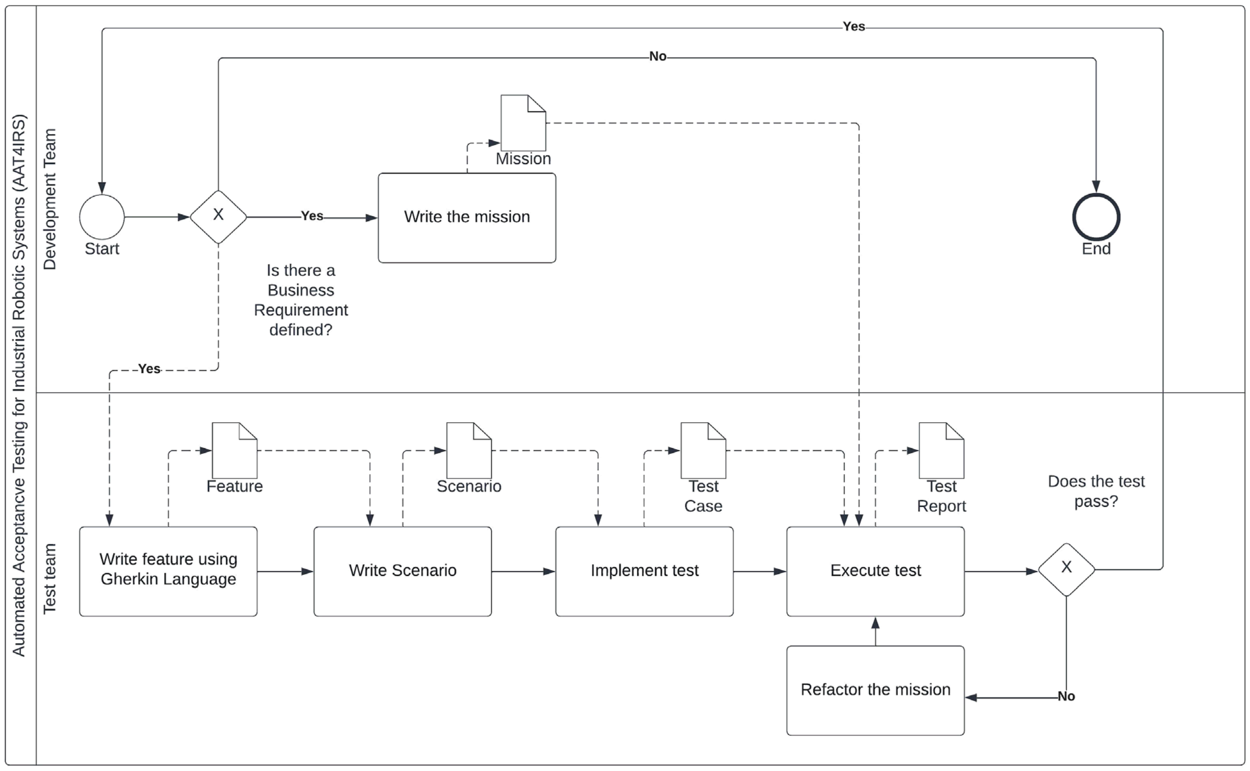


FIGURE 1
Proposed approach to applying automated acceptance testing for IRS.

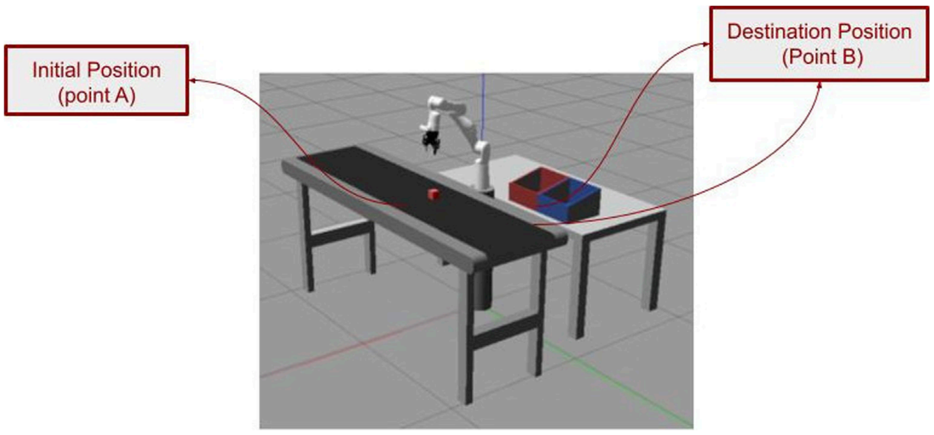


FIGURE 2
Pick-and-place task performed using a Gazebo simulator.

the user], **I want** [functionality] **so that** [benefit].” The template to describe scenarios in BDD is: “**Given** [pre condition for the scenario and test environment], **when** [action under test], **then** [expected outcomes]” (Smart, 2014). A scenario is itself composed of *step definitions* responsible for interacting directly with the system.

## 2.3 Mutation testing

Mutation testing is a software testing technique in which the original code suffers some changes called “mutated versions” or “mutants”. These mutants represent different potential defects that are modifications in the source code. These modifications

```
send(``pick'')
send(``lift/5'')
if read(``color'') = ``red'' then
    send(``turn/90'')
else
    send(``turn/270'')
end if
send(``drop/5'')
send(``release'')
```

FIGURE 3
Simple robot program.

TABLE 1 Original X mutant command for robotic systems.

| Original | Mutant |
|---|---|
| rotateleftbyX | rotaterightbyX |
| rotaterightbyX | rotateleftbyX |
| translateforwardbyX | translatebackwardsbyX |
| translatebackwardsbyX | translateforwardbyX |
| translateforwardbyX | translateforwardtoX |
| translatebackwardsbyX | translatebackwardstoX |
| docommandX | donothing |
| docommandX | docommandXtwice |
| distancevalueisX | distancevalueis − X(reversedirection) |
| distancevalueisX | distancevalueisX + noise |

include changing a mathematical operator, swapping the order of two statements, or replacing a conditional with the opposite. The mutation score is the percentage of killed mutants (detected by the test suite) with the total number of mutants.

In order to apply mutation testing, we used the test suite against each mutant. If the tests detect the changes made to create the mutants, the mutants die. If the test suite does not detect the error, the mutants survive. The number of mutants killed and that survive is used as a metric to evaluate the effectiveness of the test suite (Jia and Harman, 2011).

According to Jia and Harman (2011), mutation testing can measure the effectiveness of a test suite in terms of its ability to detect faults. ISO/IEC 25010 defines "effectiveness" as the "…accuracy and completeness with which users achieve specified goals" (ISO/IEC 25010 2011).

Our decision to use a mutation-based approach for evaluating the testing campaign is based on its ability to replicate a wide range of scenarios, surpassing the limitations of variations in environmental descriptions found within scenario files from BDD alone. Our methodology intentionally integrates non-deterministic mutants to emulate unpredictable behaviors commonly encountered in simulation and robotic systems, such as sensor noise and fluctuations in simulation speed.

Additionally, we drew inspiration from previous studies where mutation testing has been successfully applied to assess cyber-physical systems. For instance, Leotta et al. (2018) conducted a thorough investigation into automated acceptance testing for IoT systems, including sensors/actuators, smartphones, and remote cloud-based infrastructure. They evaluated their approach using mutation testing and demonstrated its practical application. Similarly, Afzal (2021) presented an innovative automated testing framework using software-in-the-loop simulation for cyber-physical systems. Their use of mutation testing showcased the effectiveness of their approach in evaluating system performance and robustness.

## 2.4 Related research

We consider related research for our study, a primary study that performed software testing activity (design and generate test cases) at different levels (unit, integration, acceptance, system) for a type of robotic system (mobile, industrial).

In this study, Nguyen et al. (2023) analyzed robotic application requirements and acceptance criteria, explicitly focusing on robotic competitions and benchmarks. They aimed to address the challenges of representing and managing requirements in the context of robotic applications' increasing complexity and diversity. We consider this research to be complementary to ours. In both studies, the authors applied BDD to rewrite the requirements for IRS. However, Nguyen et al. (2023) highlighted the application of BDD to express and manage requirements for robotic applications, emphasizing the potential for introducing automation into verifying and validating these requirements in robotics. Our study used BDD to rewrite the requirements for industrial robotic systems and implement executable tests using a simulator.

Ashraf et al. (2020) proposed coverage criteria for white-box testing to test industrial robot tasks and a framework to automatically generate the test cases to achieve the coverage criteria defined by them. However, our research does not aim to automatically generate test cases.

Breitenhuber (2020) proposed an application-level testing framework for robot software applications that uses known robotic software to describe the expected behavior of an application or its components. They focus on evaluating the component behavior in robotics systems that use the ROS framework in application-level testing. They apply the testing tools available in the ROS environment to test the components. In our study, we aimed to apply automated acceptance for industrial robotic systems in general, not just in ROS-based systems. Furthermore, in our approach, acceptance testing is automated, and the BR is translated to the BDD template for AC.

Erich et al. (2019) presented a framework for automatically testing applications for collaborative robots and demonstrated the

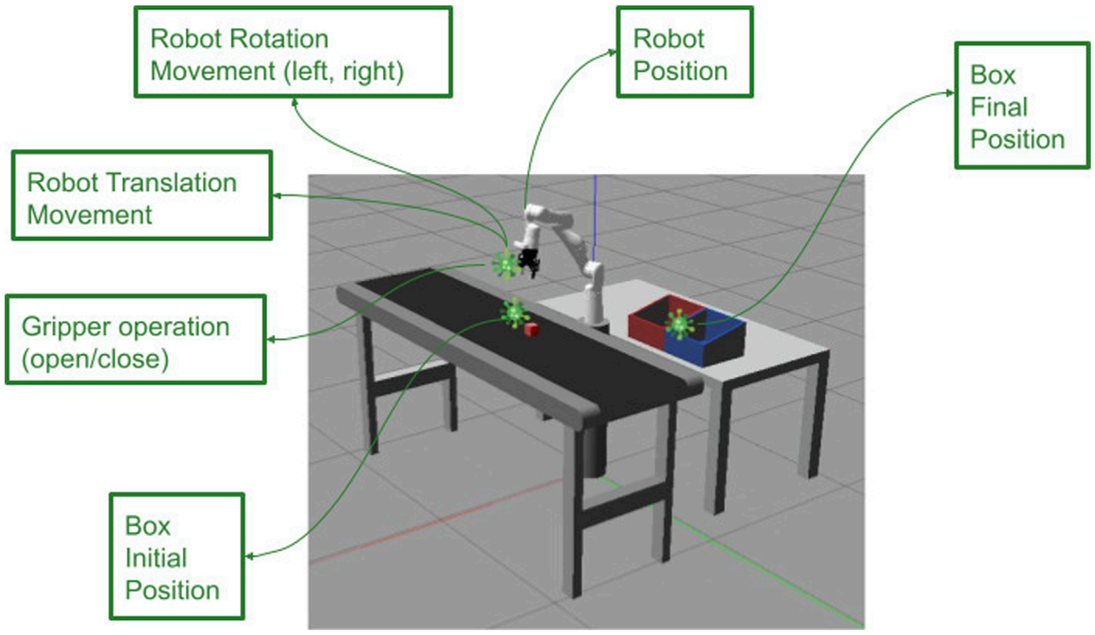

Robot Rotation Movement (left, right)
Robot Position
Box Final Position
Robot Translation Movement
Gripper operation (open/close)
Box Initial Position

FIGURE 4
Localization and operation that we applied to the mutation.

proposal in a case study for automatically testing a pick-and-place application. Their proposal was a framework applied in a physical environment and at the level of integration testing. Ours focuses on acceptance testing, leading us to consider their research and ours to be complementary.

Estivill-Castro et al. (2018) proposed a robot simulator following the model-view-controller software pattern. They use simulators with the stripped GUI under a continuous integration paradigm for robots to scale up the testing integration with robot behavior. The simulator was an environment to be applied in our approach and was not developed in our study.

Mossige et al. (2015) presented cost-effective automated testing techniques to validate complex industrial robot control systems in an industrial context and employed their methodology in continuous integration and constraint-based testing techniques. Although our research was also focused on industrial robotic systems, we focused on automated acceptance testing, so our studies are complementary.

Alexander et al. (2015) proposed the concept of situation coverage. They empirically evaluated situation coverage by testing a simple simulated autonomous road vehicle and comparing its effectiveness with random test generation. We highlight that the challenges in testing autonomous robots, as summarized by them, are similar to those for testing industrial robots, making our study and theirs complementary. However, they proposed the concept of situation coverage, which measures the proportion of all possible situations tested by a given test set as a potential solution. For them, situations are starting states and rules for projecting future states; they do not commit to a linear sequence of events. Our approach uses features and scenarios written in natural language to guide the test generation. The scenario coverage approach aims to cover a representative set of scenarios described by linear sequences. Therefore, the goal of both approaches is the same (improving software in robotics through software testing), but they use different methods.

Chung and Hwang (2007) presented a testing process and evaluation elements to test the software of intelligent robots. They proposed a test case design methodology based on user requirements and ISO standards for software testing. However, they did not perform acceptance testing using the BRs as input. Our study aims to apply AAT using acceptance criteria defined by BRs.

To our knowledge, our work is the first to apply automated acceptance testing for industrial robotic systems using the BDD template to reduce the effort in discovering faults and to ensure that the application meets specific BRs.

## 3 Proposed approach

Our approach is to apply automated acceptance testing (AAT) to evaluate whether the system meets the business requirement (BR). The system under test (SUT) is the IRS performing the expected behavior defined by the BR. We concentrate on the off-line method, which makes use of simulation.

When testing software for IRS, it is important to consider their unique features and needs. To ensure that the software meets the specific requirements and expectations of the IRS, it is crucial to involve domain experts, stakeholders, and end-users in the testing process.

According to Afzal et al. (2020), robotic systems differ from conventional software in several critical dimensions. Robots are complex systems composed of software and hardware. The

TABLE 2 List of mutants.

| Mutant number | Number | Description |
|---|---|---|
| #1 | Translation | Change the y-value in translation |
| #2 | Rotation | Change the angle orientation in rotation |
| #3 | Translation | Change the z-value in translation |
| #4 | Gripper operation | Do not change the gripper status |
| #5 | Gripper operation | Change the gripper status twice |
| #6 | Gripper operation | Do not change the gripper status with the opposite expected operation |
| #7 | Rotation | Change the angle orientation in rotation |
| #8 | Translation | Change the x-value in translation |
| #9 | Robot initial position | Sensor reading with the opposite expected value for the x-component |
| #10 | Robot initial position | Sensor reading with the opposite expected value for the y-component |
| #11 | Robot initial position | Sensor reading with the opposite expected value for the z-component |
| #12 | Robot initial position | Sensor reading with noise in the x-component |
| #13 | Robot initial position | Sensor reading with noise in the y-component |
| #14 | Robot initial position | Sensor reading with noise in the z-component |
| #15 | Box initial position | Sensor reading with the opposite expected value for the x-component |
| #16 | Box initial position | Sensor reading with the opposite expected value for the y-component |
| #17 | Box initial position | Sensor reading with the opposite expected value for the z-component |
| #18 | Box initial position | Sensor reading with noise in the x-component |
| #19 | Box initial position | Sensor reading with noise in the y-component |
| #20 | Box initial position | Sensor reading with noise in the z-component |
| #21 | Box initial position | Sensor reading with the opposite expected value for the x-component |
| #22 | Box initial position | Sensor reading with the opposite expected value for y-component |
| #23 | Box initial position | Sensor reading with the opposite expected value for the x-component |
| #24 | Box initial position | Sensor reading with noise in the x-component |
| #25 | Box initial position | Sensor reading with noise in the y-component |
| #26 | Box initial position | Sensor reading with noise in the z-component |

latter interacts with the physical world through sensors and actuators, which can lead to errors that are challenging to predict. Furthermore, the notion of correctness is hard to specify.

Thus, we propose an approach that considers the differences between conventional and robotic systems by validating whether the software meets the needs defined in the BRs. Figure 1 shows our approach, which outlines the necessary activities and corresponding input/output. Starting the AAT4IRS process requires a decisive definition of BRs and acceptance criteria (AC) to ensure that the features developed provide actual business value. Collaboration with stakeholders, including robot operators, engineers, and other relevant parties, is crucial to clearly define the AC for the robot software.

As for conventional IRS systems, the BRs need to describe the business need or problem that requires a solution. These requirements should be measurable and actionable and include AC to ensure that all stakeholders agree on what the system should accomplish.

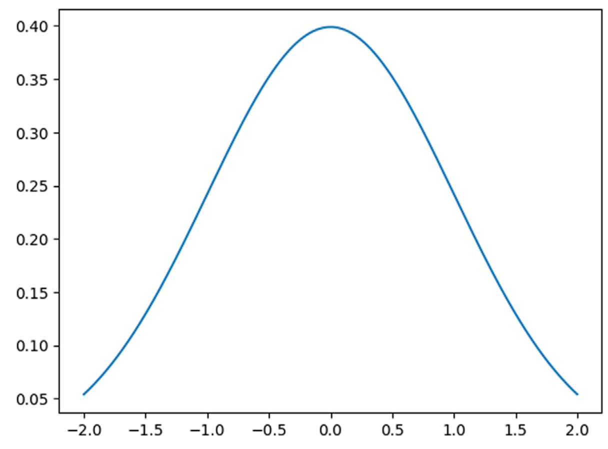


FIGURE 5
Noise added to the sensor reading.

TABLE 3 Mutant score for each round.

| *Round* | *MutantScore* |
|---|---|
| #1 | 81% |
| #2 | 77% |
| #3 | 81% |
| #4 | 81% |
| #5 | 77% |

In our approach, we can take two paths with the BR defined. One starts with the development team *writing the mission* (the application layer defined in Section 2) —essentially the system under test. The other path starts with the test using the Gherkin language to *write the feature*. The feature in our approach will adapt the template defined in BDD and will follow the following format: "**As an** operator, **I want** [process to be automated] **so that** I can automate the [process] using an IRS."

The next step in AAT4IRS is *writing scenarios*, and we follow the BDD template (Given–When–Then). The *Given* step involves outlining the initial conditions for industrial processing using robots, including robot setup, environment preparation, and sensor calibration. We connect these procedures with the *And* connector in the BDD template. The *When* step outlines the system under test. Lastly, in the *Then* step, we evaluate whether the system meets the expected behavior using the AC defined in the first activity. For example, we can use a sensor to evaluate the final position of a box.

Following the AAT4IRS approach, we implemented the test, where we must implement a function for each sentence defined in each scenario. For example, we will have a function linked with the sentence *Given* defined in the scenario. When we implement the tests, we create the link between the tests and the system under test. For example, if we have the following sentence "Given that the robot is in the initial position…", we will need to implement a function to put the robot in the initial position and also to assert whether the robot achieves the expected position.

Finally, we need to *execute the tests*. Running them will trigger the functions that access the application layer. If the test passes, the process starts again with another BR. However, if the test fails, the process refactors the mission (application layer) start until the test passes.

The output of the activity *execute tests* is a test report. We aim to improve some important aspects of the software development process for IRS. By following our approach, the software in industrial robots can undergo thorough acceptance testing to ensure that it meets the requirements and expectations of end users. It is important to involve domain experts and stakeholders throughout the acceptance testing process to ensure that the software aligns with the specific needs of the industrial robot application.

Our proposed process draws similarities with applying automated acceptance testing (AAT) to conventional systems. However, crafting features, scenarios, and test cases requires nuanced adaptation that aligns with the demands and objectives of industrial processes. To achieve this, we incorporate industry-specific language when formulating scenarios using the Given–When–Then structure. For example, the Given statement sets the initial conditions, such as the starting position of a robot. Furthermore, we utilize instrumentation tailored to industrial settings to establish AC. By employing positional sensors across three axes, we validate positions and define AC based on sensor characteristics, ensuring alignment with business objectives. Ultimately, our approach entails integrating domain-specific language to customize automated acceptance testing within the BDD standards framework.

## 4 Applying AAT4IRS to pick-and-place task

The use of industrial robots in pick-and-place scenarios is common in competitions and benchmarking exercises. Our decision to draw inspiration from a robotic competition is rooted in the limited accessibility of real-world industrial requirements, as noted by Nguyen et al. (2023). As such, we looked to the Robotic Grasping and Manipulation Competition's Task Pool for direction. In particular, we turned to a specific task outlined in their competition framework: "Pick Up and Place Using Tongs" (Sun et al., 2018).

The robot model used was the Gen3 from Kinova Robots (2022), and the end-effector is a gripper, the Robotiq-2f-85 from Robotiq (2022). This model has the follow sensors: torque, position, current, voltage, temperature, accelerometer, and gyroscope. We use the position sensor that gives us the arm position for the three axes (x, y, z).

The experiment took place in the esteemed Robot Operating System (ROS) environment, which is highly regarded in the robotics community for its adaptability and reliability. ROS is widely accepted as an ideal platform for developing robotic software due to its extensive range of libraries, tools, and conventions. Its seamless communication between various components simplifies the development of complex robot software systems. As mentioned by Quigley et al. (2015), a ROS-based system involves numerous concurrent programs that exchange messages, enabling effective collaboration and coordination.

For our experiment, we have an industrial environment with an IRS performing a pick-and-place process (an industrial process in which an industrial robot picks up an object from one location and places it in another).

We implement the experiment within simulated environments, which can lower testing expenses and expand opportunities for test automation. Timperley et al. (2018) argue that many real-world robotics bugs could be replicated and addressed in simulation environments. Moreover, simulations mitigate the risk of damaging equipment (Bossecker et al., 2023), eliminate the necessity for physical prototypes (Roth et al., 2003) and offering a cost-effective means to implement changes (Robert et al., 2020).

Therefore, we integrated a suitable simulator within the ROS environment. Among the available options, Gazebo emerged as the leading choice for robotic simulation due to its ability to replicate diverse robotic platforms equipped with standard sensors like cameras, GPS units, and IMUs. Despite operating independently, Gazebo seamlessly integrates with ROS via the "gazebo_ros" package, enabling bidirectional communication (Quigley et al., 2015; Farley et al., 2022).

The robot's mission performed in our experiment was to transfer a box from its initial position (point A) on the conveyor to its destination at the delivery table (point B) (Figure 2). The robotic arm needed to precisely navigate to the target location and use its gripper to securely grasp the object while ensuring proper alignment. Upon successful pickup, the robotic arm was to carefully place the box on the delivery table before returning to its original position. The instrumentation necessary to run our experiments, besides the IR, is sensors to read the IR, box color, and the box positions.

Following the approach defined at Section 3, we write the *mission* for the IR to meet the follow BR.

**BOX 1 Business requirement.**

We must move the box from the conveyor to the delivery point, respecting the box color. The robot needs to be positioned correctly before the pick-and-place process can begin. The final position of the box should not exceed 0.02 cm in any of the three axes.

We would like to highlight that the AC defined by the BR was related to the box's final position. However, to perform the pick-and-place process, we also needed the robot's and box's initial position. Therefore, we rewrote the BR to add these AC with the acceptable threshold.

**BOX 2 Business requirement.**

We must move the box from the conveyor to the delivery point, respecting the box color. The robot needs to be positioned correctly before the pick-and-place process can begin within a threshold of 0.02 cm in all three axes. For the box, the threshold is 0.01 cm, and also for the three axes. The final position of the box should not exceed 0.02 cm in any of the three axes.

After writing the **mission**, we created the test suite following AAT4IRS. We used the *pytest-bdd* library (Pidsadnyi and Bubenkov, 2022), the most popular framework in Python that implements a subset of the Gherkin language to enable the automating of project requirements testing.

To achieve this, we use Gerkin language to write a **feature**. The first step was to establish the initial environment for our experiment, which involved initializing both the robot and conveyor, moving the robot to the home position, and placing the box in a specific location. Given that each position has three axes, we composed a *Given* statement for each axis, as shown in the Listing 1. The *When* statement involved the pick-and-place automation system we are testing, while the *Then* statement determined whether the box was in the expected position with the maximum error as defined by the AC in the BR.

In order to proceed, we needed to perform a test that created a function for each statement in the scenario. The criteria for accepting are related to the robot and box within specified position limits. The next step was to use the feature to construct the scenario. As the AC were based on data from sensors that monitor the positions of the robot and box, we incorporated sensor readings into the test script to create assertions (Listing 2 and 3). Once the test suite was developed, we ran it against the original code and successfully passed all tests.

## 5 Evaluation

We evaluated AAT4IRS through mutation testing. The mutants defined took into account the fact that the code for a robotic system is not just any piece of procedural code but a specific type of program that manipulates robotic components that interact with the physical world.

Figure 3 shows a pseudo-code for a possible program in which the IR performs the pick-and-place task. The robot picks the box, lifts it from the ground by some amount, and does the reverse operations after turning to two possible angles, depending on the box's color determined by a visual sensor. The code is mostly a linear sequence of *send* and *read* commands, which, in the context of ROS, give commands to the robot and probe the state of the environment, respectively. The program has few control structures, variables, or arithmetical operations, which are the typical points where mutation operators are applied. There are very few locations where mutations can be injected.

In this context, one could understand that the numbers 90 and −270 mean "left" and "right", and that confusing one for the other is probably more likely for a developer than providing an angular value that is incorrect by a single degree, such as 90 and −90. We also introduced mutants that replicated errors in the sensor reading. Our goal was to create a comprehensive set of mutants that accurately represent the solution space under examination, and to provide an informative value related to the effectiveness of our approach. Thus, we identified the high-level write/read operations and defined appropriate ways of mutating them. Table 1

```
#sendBox.feature
Feature: Send box according to the box color to the target with the same color
  As a operator,
  I want to send the box with a specific color to the the delivery point using the robot
  So that I can automate the process using a industrial robotic system

  Scenario: Move the box with a specific color from conveyor to the target with the same color
    Given the industrial robotic system is initialized
    And the conveyor is initialized
    And the robot is at home with the position error threshold being less than or equal "0.02" cm in the
     x-axis
    And the robot is at home with the position error threshold being less than or equal "0.02" cm in the
     y-axis
    And the robot is at home with the position error threshold being less than or equal "0.02" cm in the
     z-axis
    And the box is in the conveyor with the position error threshold being less than or equal "0.01" cm
    in the x-axis
    And the box is in the conveyor with the position error threshold being less than or equal "0.01" cm
    in the y-axis
    And the box is in the conveyor with the position error threshold being less than or equal "0.01" cm
    in the z-axis
    When the pick-and-place finished
    Then the box should be in the specific target with the position error threshold being less than or
    equal "0.02" cm in the x-axis
    Then the box should be in the specific target with the position error threshold being less than or
    equal "0.02" cm in the y-axis
    Then the box should be in the specific target with the position error threshold being less than or
    equal "0.02" cm in the z-axis
```

Listing 1. Feature and scenario.

shows the possible transitions between the original code and the mutant.

Our approach for assessing our methodology involved following the guidelines specified in Table 1. It is worth mentioning that we customized these to align with the language and instrumentation utilized in robotics and simulation. Our evaluation entailed generating mutants that pertained to rotation (orientation modifications) and translation (position adjustments) operations for the robot, as well as variations in the initial and final positions for both the robot and the box. Furthermore, we created mutants for the gripper operations, comprising opening and closing actions (Figure 4).

Table 2 shows the 26 mutants created using our guideline and the adaptation needs for the specific robot used in our experiment.

For the noise added to sensor readings, we added the noise shown in Figure 5. This noise is a simulation of a Gaussian noise normally distributed, often used to model measurement errors or communication noise. As we can see, the lowest value for this sign is around *0.053* and the biggest is around *0.39*.

In order to guarantee the randomness of the noise added and diversity of the boxes, we conducted the test suite for each of the 26 mutants five times. The mutant scores for each round are presented in Table 3.

## 6 Discussion

During our experiment, the industrial robot (IR) was tasked with picking boxes from a conveyor and delivering them to designated points. To thoroughly evaluate the effectiveness of our methodology, we conducted a test suite that included the original code and 26 carefully selected mutants, representing a wide range of potential solutions. This comprehensive evaluation provided valuable insights into the efficacy of our approach.

We obtained an average score of **79%** effectiveness. Analyzing the surviving mutants in all 130 executions, those that survived were **#5**, **#10**, **#14**, **#20**, and **#24** in different rounds.

The fifth mutation affects the gripper's operation, causing it to close twice instead of once. This change persisted as it did not interfere with the robot's mission. However, if time is a crucial factor in meeting our acceptance criteria, the additional operation may affect the overall mission execution time.

A reversal in the error value for the y-component is the transition for **Mutant #10**. Although the expected value stands at 0.002 cm, the acceptance criteria (AC) specifies 0.02 cm. Despite this, the sensor reading remained within the acceptable threshold due to the introduced error. This highlights the **importance of aligning the definition of acceptance criteria in the business requirement (BR) with the magnitude order of variables utilized in the process**. Such alignment directly influences the resulting outcomes.

```
//testPickAndPlace.py
...
scenarios('../features/sendBox.feature')
@given("the industrial robotic system is initialized")
def setup_robot():
@given("the conveyor is initialized")
def setup_conveyor():
@given(parsers.cfparse('the robot is at home with the position error threshold being less than or equal
    "{errorThreshold:Number}" cm in the x-axis'
    , extra_types=dict(Number=float)))
def robot_position_initialX(errorThreshold):
@given(parsers.cfparse('the robot is at home with the position error threshold being less than or equal
    "{errorThreshold:Number}" cm in the y-axis', extra_types=dict(Number=float)))
def robot_position_initialY(errorThreshold):
@given(parsers.cfparse('the robot is at home with the position error threshold being less than or equal
    "{errorThreshold:Number}" cm in the z-axis', extra_types=dict(Number=float)))
def robot_position_initialZ(errorThreshold):
@given(parsers.cfparse('the box is in the conveyor with the position error threshold being less than or
    equal "{errorThreshold:Number}" cm in the x-axis',extra_types=dict(Number=float)))
def box_positionAX(errorThreshold):
@given(parsers.cfparse('the box is in the conveyor with the position error threshold being less than or
    equal "{errorThreshold:Number}" cm in the y-axis',extra_types=dict(Number=float)))
def box_positionA(errorThreshold):
@given(parsers.cfparse('the box is in the conveyor with the position error threshold being less than or
    equal "{errorThreshold:Number}" cm in the z-axis',extra_types=dict(Number=float)))
def box_positionA(errorThreshold):
@when("the pick-and-place finished")
def pickplace_mission():
@then(parsers.cfparse('the box should be in the specific target with the position error threshold being
    less than or equal "{errorThreshold:Number}" cm in the x-axis',extra_types=dict(Number=float)))
def box_positionBX(errorThreshold):
@then(parsers.cfparse('the box should be in the specific target with the position error threshold being
    less than or equal "{errorThreshold:Number}" cm in the y-axis',extra_types=dict(Number=float)))
def box_positionBY(errorThreshold):
@then(parsers.cfparse('the box should be in the specific target with the position error threshold being
    less than or equal "{errorThreshold:Number}" cm in the z-axis',extra_types=dict(Number=float)))
def box_positionBZ(errorThreshold):
```

Listing 2. All the statements created at feature are contemplated by a test in the test file.

```
    @given(parsers.cfparse('the robot is at home with the position error threshold being less than or
    equal "{errorThreshold:Number}" cm in the x-axis'
    , extra_types=dict(Number=float)))
def robot_position_initialX(errorThreshold):
    positionRobot = readSensor(1)
    assert round(abs(0.5761017203330994 - positionRobot[0]),2) <= errorThreshold
```

Listing 3. An example for the test using the AC.

We utilized a randomized sorting method to introduce variability into the noise added to sensor readings. However, we discovered that for surviving mutants **#14**, **#20**, and **#24**, the noise added was within the established error threshold of the acceptance criteria. Consequently, we determined that aligning the acceptance criteria in the BR with the physical attributes of the instrumentation is crucial. This alignment directly impacts the outcomes. Thus, when defining the acceptance criteria in our experiment, it is imperative to consider the accuracy of the sensors.

Our analysis also revealed that adopting a natural language approach to define AC in our methodology can provide significant benefits for business analysts, developers, and testers throughout the system development lifecycle since they are also observed when application acceptance testing from BDD is applied in conventional systems. This approach generates a report through the when application acceptance testing process that serves as living documentation, accessible to the entire team. Importantly, this living documentation is not just a snapshot but is consistently updated to reflect the latest version of the application, making it an invaluable resource for the team. As highlighted by Afzal et al. (2020), the complexity of designing and crafting tests for software systems requires effective channels for coordination, collaboration, and documentation within robotic systems development teams. Adopting AAT is an excellent strategy for addressing these challenges. However, it needs qualitative studies with different stakeholders.

## 7 Threats to validity

Validity threats usually occur in a mapping study, and it was no different in our study. We highlight some of these threats and the mechanism we applied to address them.

*Mutant generation.* We created mutants with just one transition for each mutant. This was acceptable because of the nature of the acceptance test for each scenario. The tests implementing each sentence used in our approach (Given–Then–When) are executed in sequence. Thus, when one test fails, the process is stopped—if the test that implements the *Given* sentence fails, the process stops, and the test report is generated. Furthermore, no more tests are executed. Therefore, the unique transition for each mutant was an acceptable method for creating mutants to evaluate our approach.

*Input data.* As observed in the experiment, sensor characteristics were the reason for the survived mutant. Our approach is not concerned with the input data, but we strongly suggest that domain experts must also choose the input data. Moreover, our experiment performed a well-known process in the robotic field, and we used near-accurate data. Thus, further experiments with more realistic input data will be necessary to confirm the effectiveness of our approach.

## 8 Conclusion

The present research outlines an approach to automated acceptance testing (AAT) that aims to improve fault detection in industrial robotic systems (IRS). However, one challenge to applying software testing for robotic systems is related to communication and collaboration: the culture of testing.

Our study utilized an approach based on behavior-driven development (BDD); more specifically, AAT that uses natural language. Our implementation used ROS, Gazebo, and pytest-bdd, a Python library dedicated to BDD. To evaluate the effectiveness of our software testing approach, we tested the generated test suites against mutants created from the original code. The test suites produced using AAT4IRS achieved an effectiveness score of 79%.

In our assessment, we utilized mutation testing to generate mutants that accurately reflect the complexities of the robotic landscape. Our thorough methodology entailed creating mutants that focused on non-deterministic elements that are inherently present in robotic systems, such as fluctuations in sensor readings, as well as mutants that accounted for linguistic subtleties. By employing this nuanced approach, we were able to gain valuable insights into the robustness and flexibility of our proposed methodology within the constantly evolving field of robotics.

When evaluating business requirements for industrial robotic systems, it is crucial to consider both the robot's physical attributes and overall business objectives. Achieving alignment across all teams involved in the project, which may include individuals from various backgrounds, is essential when establishing acceptance criteria (AC). Additionally, utilizing live documentation made possible by AAT4IRS implementation can help foster collaboration among teams, allowing for more effective problem-solving when facing the complex challenges of these types of projects.

Our aim for future research is (i) to apply our approach by performing controlled experiments with a group of roboticists, (ii) to apply and evaluate AAT4IRS using physical IRS, (iii) and to perform a qualitative evaluation with different stakeholders.

## Data availability statement

The original contributions presented in the study are publicly available. This data can be found here: https://github.com/mgdossantos/aat4irs_v3.

## Author contributions

MS: investigation, methodology, software, writing–original draft, and writing–review and editing. SH: conceptualization, funding acquisition, investigation, supervision, and writing–review and editing. FP: conceptualization, funding acquisition, supervision, and writing–review and editing. Y-GG: resources and writing–review and editing.

## Funding

The authors declare financial support was received for the research, authorship, and/or publication of this article. This paper is an extended version of our conference paper that was accepted at IEEE Robotic Computing 2022 (Santos et al., 2022). The NSERC Discovery Grant and Canada Research Chair programs partially supported the authors.

## Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## Publisher's note

## References

Afzal, A. (2021). Automated testing of robotic and cyberphysical systems. doi:10.1184/R1/16645639.v1

Afzal, A., Goues, C. L., Hilton, M., and Timperley, C. S. (2020). "A study on challenges of testing robotic systems," in *2020 IEEE 13th international conference on software testing, validation and verification (ICST)*, 96–107. doi:10.1109/ICST46399.2020.00020

Afzal, A., Katz, D. S., Le Goues, C., and Timperley, C. S. (2021). "Simulation for robotics test automation: developer perspectives," in *2021 14th IEEE conference on software testing, verification and validation (ICST)*, 263–274. doi:10.1109/ICST49551.2021.00036

Ahmad, A., and Babar, M. A. (2016). Software architectures for robotic systems-a systematic mapping study. *J. Syst. Softw.* 122, 16–39. doi:10.1016/j.jss.2016.08.0391016/j.jss.2016.08.039

Alexander, R., Hawkins, H., and Rae, A. (2015). Situation coverage – a coverage criterion for testing autonomous robots

Ashraf, A. K., D'Souza, M., and Jetley, R. (2020). "Coverage criteria based testing of industrial robots," in *2020 IEEE 16th international Conference on automation Science and engineering (CASE) (Hong Kong, China: IEEE)*, 16–21. doi:10.1109/CASE48305.2020.9217031

Bossecker, E., Sousa Calepso, A., Kaiser, B., Verl, A., and Sedlmair, M. (2023). "A virtual reality simulator for timber fabrication tasks using industrial robotic arms," in *Proceedings of mensch und computer 2023* (New York, NY, USA: Association for Computing Machinery), 23, 568–570. MuC. doi:10.1145/3603555.3609316

Breitenhuber, G. (2020). "Towards application level testing of ros networks," in *2020 fourth IEEE international conference on robotic computing (IRC)*, 436–442. doi:10.1109/IRC.2020.00081

Bretl, T., and Lall, S. (2008). Testing static equilibrium for legged robots. *IEEE Trans. Robotics* 24, 794–807. doi:10.1109/TRO.2008.2001360

Chillarege, R. (1996). What is software failure? *IEEE Trans. Reliab.* 45, 354. doi:10.1109/TR.1996.536980

Chung, Y. K., and Hwang, S.-M. (2007). "Software testing for intelligent robots," in *2007 international conference on control, automation and systems* (Seoul, Korea (South): IEEE), 2344–2349. doi:10.1109/ICCAS.2007.4406752

Erich, F., Saksena, A., Biggs, G., and Ando, N. (2019). "Design and development of a physical integration testing framework for robotic manipulators," in *2019 IEEE/SICE international Symposium on system integration (SII) (paris, France: IEEE)*, 602–607. doi:10.1109/SII.2019.8700444

Estivill-Castro, V., Hexel, R., and Lusty, C. (2018). Continuous integration for testing full robotic behaviours in a GUI-stripped simulation. *CEUR Workshop Proc.* 2245, 453–464.

Farley, A., Wang, J., and Marshall, J. A. (2022). How to pick a mobile robot simulator: a quantitative comparison of coppeliasim, gazebo, morse and webots with a focus on accuracy of motion. *Simul. Model. Pract. Theory* 120, 102629. doi:10.1016/j.simpat.2022.102629

Heimann, O., and Guhl, J. (2020). Industrial robot programming methods: a scoping review. *2020 25th IEEE Int. Conf. Emerg. Technol. Fact. Automation (ETFA)* 1, 696–703. doi:10.1109/ETFA46521.2020.9211997

IFR (2024). Executive summary world robotics 2023 industrial robots. Available at: https://ifr.org/img/worldrobotics/Executive_Summary_WR_Industrial_Robots_2023.pdf (Accessed April 06, 2024).

Irshad, M., Britto, R., and Petersen, K. (2021). Adapting behavior driven development (bdd) for large-scale software systems. *J. Syst. Softw.* 177, 110944. doi:10.1016/j.jss.2021.110944

ISO/IEC 25010 (2011). ISO/IEC 25010:2011, Systems and software engineering — systems and software Quality Requirements and Evaluation (SQuaRE) — system and software quality models. *Tech. Rep.* International Organization for Standardization.

ISO/IEC 8373 (2012). Robots and robotic devices — vocabulary. *Tech. Rep.* International Organization for Standardization.

Jia, Y., and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* 37, 649–678. doi:10.1109/TSE.2010.62

Leotta, M., Clerissi, D., Olianas, D., Ricca, F., Ancona, D., Delzanno, G., et al. (2018). An acceptance testing approach for internet of things systems. *IET Softw.* 12, 430–436. doi:10.1049/iet-sen.2017.0344

Mossige, M., Gotlieb, A., and Meling, H. (2015). Testing robot controllers using constraint programming and continuous integration. *Inf. Softw. Technol.* 57, 169–185. doi:10.1016/j.infsof.2014.09.009

Naik, K., and Tripathy, P. (2018). *Software testing and quality assurance: theory and practice*. Nova Jersey: EUA Wiley Publishing. 2nd edn.

Nguyen, M., Hochgeschwender, N., and Wrede, S. (2023). "An analysis of behaviour-driven requirement specification for robotic competitions," in *2023 IEEE/ACM 5th international workshop on robotics software engineering (RoSE)*, 17–24. doi:10.1109/RoSE59155.2023.00008

Nicieja, K. (2017). *Writing great specifications: using specification by example and Gherkin*. 1st edn. USA: Manning Publications Co.

Papadakis, M., Shin, D., Yoo, S., and Bae, D.-H. (2018). "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *Proceedings of the 40th international conference on software engineering* (New York, NY, USA: Association for Computing Machinery), 18, 537–548. ICSE. doi:10.1145/3180155.3180183

Pidsadnyi, O., and Bubenkov, A. (2022). Welcome to pytest-bdd's documentation. Available at: https://pytest-bdd.readthedocs.io/en/stable/ (Accessed November 03, 2022).

Pogliani, M., Maggi, F., Balduzzi, M., Quarta, D., and Zanero, S. (2020). "Detecting insecure code patterns in industrial robot programs," in *Proceedings of the 15th ACM asia conference on computer and communications security* (New York, NY, USA: Association for Computing Machinery), 20, 759–771. ASIA CCS. doi:10.1145/3320269.3384735

Quigley, M., Gerkey, B., and Smart, W. D. (2015). *Programming robots with ros: a practical introduction to the robot operating system*. 1st edn. Sebastopol, California: O'Reilly Media, Inc.

Robert, C., Sotiropoulos, T., Waeselynck, H., Guiochet, J., and Vernhes, S. (2020). The virtual lands of oz: testing an agribot in simulation. *Empir. Softw. Engg.* 25, 2025–2054. doi:10.1007/s10664-020-09800-3

Robotiq (2022). Start production faster—robotiq. Available at: https://robotiq.com/products/2f85-140-adaptive-robot-gripper (Accessed August 19, 2022).

Robots, K. (2022). Discover our gen3 robots—kinova. Available at: https://www.kinovarobotics.com/product/gen3-robots (Accessed August 19, 2022).

Roth, H., Ruehl, M., and Dueber, F. (2003). A robot simulation system for small and medium-sized companies. *IFAC Proc. Vol.* 36, 91–96. doi:10.1016/S1474-6670(17)33375-X

Santos, M. G. D., Petrillo, F., Hallé, S., and Guéhéneuc, Y.-G. (2022). "An approach to apply automated acceptance testing for industrial robotic systems," in *2022 sixth IEEE international conference on robotic computing (IRC)*, 336–337. doi:10.1109/IRC55401.2022.00066

Smart, J. (2014). *BDD in Action: behavior-driven development for the whole software lifecycle (Shelter Island)*. New York, United States: Manning Publications.

Solis Pineda, C., and Wang, X. (2011). A study of the characteristics of behaviour driven development, 383–387. doi:10.1109/SEAA.2011.76

Sun, Y., Falco, J., Cheng, N., Choi, H. R., Engeberg, E. D., Pollard, N., et al. (2018). "Robotic grasping and manipulation competition: task pool," in *Robotic grasping and manipulation* (Cham: Springer International Publishing), 1–18.

Timperley, C., Afzal, A., Katz, D. S., Hernandez, J., and Goues, C. L. (2018). "Crashing simulated planes is cheap: can simulation detect robotics bugs early?," in *2018 IEEE 11th international conference on software testing, verification and validation (ICST)* (Los Alamitos, CA, USA: IEEE Computer Society), 331–342. doi:10.1109/ICST.2018.00040

Wynne, M., and Hellesoy, A. (2012). *The cucumber book: behaviour-driven Development for Testers and developers* (pragmatic bookshelf)

Frontiers in Robotics and AI

TYPE Original Research
PUBLISHED 03 January 2025
DOI 10.3389/frobt.2024.1363443

OPEN ACCESS

EDITED BY
Ivano Malavolta,
VU Amsterdam, Netherlands

REVIEWED BY
Peng Wang,
Manchester Metropolitan University,
United Kingdom
Uwe Aßmann,
Technical University Dresden, Germany

*CORRESPONDENCE
Momina Rizwan,
momina.rizwan@cs.lth.se
Christoph Reichenbach,
christoph.reichenbach@cs.lth.se

RECEIVED 30 December 2023
ACCEPTED 20 September 2024
PUBLISHED 03 January 2025

CITATION
Rizwan M, Reichenbach C, Caldas R, Mayr M and Krueger V (2025) EzSkiROS: enhancing robot skill composition with embedded DSL for early error detection.
*Front. Robot. AI* 11:1363443.
doi: 10.3389/frobt.2024.1363443

# EzSkiROS: enhancing robot skill composition with embedded DSL for early error detection

Momina Rizwan[1]*, Christoph Reichenbach[1]*, Ricardo Caldas[2], Matthias Mayr[1] and Volker Krueger[1]

[1]Department of Computer Science, Faculty of Engineering (LTH), Lund University, Lund, Sweden, [2]Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden

When developing general-purpose robot software components, we often lack complete knowledge of the specific contexts in which they will be executed. This limits our ability to make predictions, including our ability to detect program bugs statically. Since running a robot is an expensive task, finding errors at runtime can prolong the debugging loop or even cause safety hazards. This paper proposes an approach to help developers catch these errors as soon as we have some context (typically at pre-launch time) with minimal additional efforts. We use embedded domain-specific language (DSL) techniques to enforce early checks. We describe design patterns suitable for robot programming and show how to use these design patterns for DSL embedding in Python, using two case studies on an open-source robot skill platform SkiROS2, designed for the composition of robot skills. These two case studies help us understand how to use DSL embedding on two abstraction levels: the high-level skill description that focuses on what the robot can do and under what circumstances and the lower-level decision-making and execution flow of tasks. Using our DSL EzSkiROS, we show how our design patterns enable robotics software platforms to detect bugs in the high-level contracts between the robot's capabilities and the robot's understanding of the world. We also apply the same techniques to detect bugs in the lower-level implementation code, such as writing behavior trees (BTs), to control the robot's behavior based on its capabilities. We perform consistency checks during the code deployment phase, significantly earlier than the typical runtime checks. This enhances the overall safety by identifying potential issues with the skill execution before they can impact robot behavior. An initial study with SkiROS2 developers shows that our DSL-based approach is useful for finding bugs early and thus improving the maintainability of the code.

KEYWORDS

**embedded domain-specific languages, robot skills, skill-based control platforms, behavior trees, domain-specific language design patterns**

## 1 Introduction

The design and implementation of robotic systems to perform socio-technical missions have never been more relevant or challenging. To ensure that robot developers can meet market demands with confidence in the correctness of their systems, a range of development tools and techniques is required. Specifically, robot development tools should provide expressive programming languages and

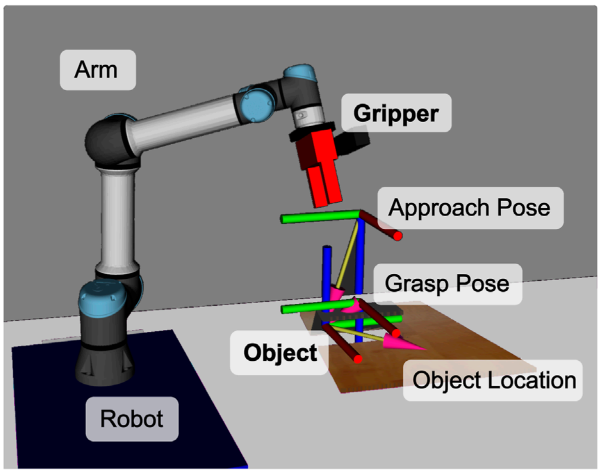


FIGURE 1
Robot using a pick skill with a visualization of the necessary parameters. To run this skill, we only need the Gripper and the Object parameters. SkiROS2 can deduce all other necessary parameters through a set of rules in the skill description shown in listings 4 and 6.

frameworks that allow human developers to describe correct robot behavior (Brugali et al., 2007). One such robot development platform is *SkiROS2*[1], a skill-based robot control platform with knowledge integration. *SkiROS2* (Mayr et al., 2023b) allows developers to define modular skills for autonomous mission execution.

These skills, ranging from "pick" to "drive," are modularly defined with pre- and post-conditions. In SkiROS2, the assessment and validation of these conditions rely on the robot's knowledge, systematically organized into an ontology. These *ontologies* are a rich, interlinked representation of concepts and relationships within a specific domain. They serve as a foundation for verifying that all necessary conditions for skill execution are satisfied. For instance, in an automated assembly line or robotic healthcare surgery, the ontology would encompass all relevant entities and their relationships, providing a comprehensive context for skill execution.

Consider a scenario where the robot has to pick an Object with its Gripper as shown in Figure 1. The pre-conditions of a "pick" skill might include ontology-based relationships such as "*the gripper is part of the robot arm*." This relationship assists in deducing additional parameters such as "*which arm to move*" by employing subtle semantic differences of entities and their relationships in the ontology. For example, if we say that the gripper is part of the arm, then we know which arm to move if we want to pick an object with the gripper. The distinction between relationships like "*is part of*" and "*is holding*" is critical in ensuring the correct application of parameters and actions during skill execution.

The developer must be careful when declaring such relationships as bugs introduced at this stage can lead to silent errors, disrupting the skill's behavior and potentially leading to incorrect or inefficient task execution. The reason is that some of these errors in the skill description are logical errors that would not manifest themselves as explicit runtime errors. Certain errors may only become evident when a particular skill is executed, which could be weeks later when demonstrating the robot under specific circumstances that are not immediately predictable. This delay in detection makes troubleshooting and rectifying these errors more challenging. Therefore, properly defining relationships and conditions within the ontology and skill descriptions is crucial to ensure the technical correctness and operational reliability of robotic skills in real-world applications.

In SkiROS2, each high-level skill description acts as a behavioral contract, setting parameters and conditions that the corresponding implementations must satisfy. These descriptions guide the development of concrete skill implementations. Many implementations use extended behavior trees (BTs) that reuse other existing skills, relying on their pre-conditions and post-conditions for a structured execution. Extended BTs in SkiROS2 merge task-level planning and execution, allowing for modularity and reactivity (Rovida et al., 2017b). The reactivity stems inherently from BTs in the way with which tasks are organized, which defines their priority order, with more important tasks interrupting less important tasks (Iovino et al., 2022). However, constructing consistent and correct BTs is crucial as inconsistencies can lead to unexpected failures and outcomes.

To avoid such errors, we propose using a domain-specific language (DSL) to allow the code to be analyzed for potential errors before deploying it on the actual robot. Our proposed approach ensures that the high-level abstract skill descriptions align with the lower-level BTs, providing a comprehensive framework for skill execution. DSLs offer specific constructs for defining and connecting nodes, conditions, and actions, enforcing correct patterns and practices, thus reducing the likelihood of logical or structural errors. The benefits of using DSLs to aid in debugging, visualization, and static checking are well recognized, making them a valuable tool in robot software development. DSLs have been used for mission specification (Dragule et al., 2021) and modeling of robot knowledge (Ceh et al., 2011). Nordmann et al. (2016) collected and categorized over 100 such DSLs for robotics in their *Robotics DSL Zoo*[2].

We aim to support robot developers, particularly those who write control logic in Python, in catching bugs early by embedding DSLs directly in Python. We support our case through the following ways:

- Four design patterns for *embedding DSLs in general-purpose programing languages* that address common challenges in robotics, with details on how to implement these patterns in Python.
- A case study of a robotics software SkiROS2, in which we introduce our DSL EzSkiROS for early detection of type errors and other bugs, highlighting its effectiveness in identifying errors in both high-level skill descriptions and lower-level implementation details.
- A demonstration of how EzSkiROS detects various types of bugs in robot capabilities, world model contracts, and behavior trees,

1 https://github.com/RVMI/skiros2

2 https://corlab.github.io/dslzoo

showcasing the DSL's comprehensive coverage and versatility in early detection of bugs early.

Lastly, we discuss the advancements and distinctions of our approach compared to the initial insights presented in the paper Rizwan et al. (2023), providing an overview of the evolution and impact of our design patterns.

## 2 Related work

Several studies have explored the use of model-driven approaches for programming robots, focusing on the development of DSLs to enhance the reliability of robotic systems. Buch et al. (2014) described an internal DSL technique written in C++, which incorporates structuring of complex actions, where actions are modeled through sets of parameters, and each action contains a pre-condition specifying the state of relevant parts. This structure implies the use of pre- and post-conditions in sequencing robotic skills. Unlike our DSL, their DSL uses a model-driven approach, which instantiates the textual representation of the assembly sequence, which is interpreted to execute the assembling behavior. However, it is unclear if they use early checking techniques to prevent erroneous sequences. Although it discusses error handling and the probabilistic approach to tackle uncertainties, specific methods like early checking techniques are not clearly outlined.

Kunze et al. (2011) proposed the Semantic Robot Description Language (SRDL), a model-based approach that utilizes the Web Ontology Language (OWL) notation to match robot descriptions and actions through the static analysis of robot capability dependencies. SRDL models the knowledge about robots, capabilities, and actions, contributing to the understanding and specification of robotic behaviors. However, the extent to which SRDL supports early dynamic checking in general-purpose languages remains unclear, highlighting the need for further exploration in this area.

Coste-Maniere and Turro (1997) proposed MAESTRO, an external DSL for specifying the reactive behavior and checking in the robotics domain. MAESTRO focuses on complex and hierarchical missions, accommodating concurrency and portability requirements. It allows the specification of user-defined typed events and conditions, offering type-checking of user-defined types and stop condition checks to ensure the correctness and safety of specified behaviors.

Behavior trees have emerged as an effective method to model and execute autonomous robotic behaviors, particularly in dynamic environments. Unlike the traditional finite-state machines (FSMs), BTs represent action selection decisions in a hierarchical tree structure enhancing the flexibility in planning and replanning robotic behavior. Dortmans and Punter (2022) highlighted that BTs offer a more maintainable approach to decision-making than FSMs, which is crucial in the rapidly evolving field of robotics. Originally developed for the video game industry, BTs have been widely adopted in robotics due to their modularity and scalability. Iovino et al. (2022) presented a detailed survey of BTs in robotics and AI, discussing their application, evolution, and benefits. BTs are composed of various types of nodes, including control nodes (e.g., sequences and selectors), leaf nodes (e.g., tasks and conditions), and decorator nodes (modifying the behavior or output of other nodes), organized in a tree structure from a root node and branching out.

Integration of BTs with robotic systems often involves the use of DSLs and frameworks such as the robot operating system (ROS). Ghzouli et al. (2023) emphasized the growing use of BTs in open-source robotic applications supported by ROS, indicating their practicality in the real-world applications. However, verifying the safety and correctness of BTs remains a challenge.

Henn et al. (2022) used SMTs to check safety properties specified in the linear constraint Horn clause notation over behavior tree specifications. Moreover, Tadiello and Troubitsyna (2022) used Event-B for the formal specification and verification of BT instances, ensuring the maintenance of invariant properties.

From a static semantics perspective, BhTSL is an example where the compiler checks the source text for non-declared variables and variable redeclaration (Oliveira et al., 2020). Despite the advancements in BT DSLs, there is a lack of DSLs performing static checks as rigorously as desired. According to the survey paper (Ghzouli et al., 2020), the most used behavior tree DSLs, such as BehaviorTree.CPP[3], py_trees[4], and the behavior tree from Unreal Engine[5], primarily focus on runtime type safety and flexibility. For instance, in the MOOD2Be's[6] project from Horizon 2020, the BehaviorTree.CPP tool offers a C++ implementation of BTs with type safety (Faconti, 2019), but the type-checking capability is largely left to the developer and is subject to runtime checks. This indicates a gap in the domain of DSLs for BTs in ensuring correct execution behavior and preventing inconsistencies in the implementation between the skills or actions in a BT before runtime.

In conclusion, although there have been significant advancements in DSLs for robotics and BTs, there is a continuous need for the development of languages and tools that allow for both static and early dynamic checks to ensure the safety, reliability, and efficiency of robotic systems. Future research should focus on enhancing the capabilities of DSLs to perform comprehensive checks and verification, both at design time and runtime, to address the increasing complexity and demands of modern robotic applications.

## 3 Embedding robotics DSLs in Python

Domain-specific languages can help developers by simplifying the notation, improving performance, or through early error detection. However, development and maintenance of DSLs requires effort. For *external DSLs* (e.g., MAESTRO and SRDL), much of this effort comes from building a language frontend. *Internal* or *embedded DSLs* [as shown in Buch et al. (2014)] avoid this overhead and instead re-use an existing "host" language, possibly adjusting the language's behavior to accommodate the needs of the problem domain.

3 https://github.com/BehaviorTree/BehaviorTree.CPP

4 https://github.com/splintered-reality/py_trees

5 https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/BehaviorTrees

6 https://robmosys.eu/mood2be/

We consider Python as one of the three main languages supported by the popular robotics platform ROS (Quigley et al., 2009). The other two languages, C++ and LISP, also support internal DSLs, but with different trade-offs.

## 3.1 Python language features for DSLs

Although Python's syntax is fixed, it offers several language constructs that DSL designers can repurpose to reflect their domain, such as freely overloadable infix operators (excluding type-restricted Boolean operators), type annotations (since Python 3.0), and runtime reflection.

Listing 1 illustrates how the Python code can use these three techniques. Here, class `MagicDict` inherits from Python's built-in `dict` class (representing mutable finite maps or associative arrays) and defines two Python functions. An instance of this `MagicDict` class behaves almost entirely like a regular `dict`, meaning that we can, e.g., read from and write to its elements (Listing 2, lines 2–5).

The first Python function we define in `MagicDict` is `__getattribute__` (Listing 1, lines 4–11), which is a special operation that Python uses to resolve the names of attributes (meaning fields and methods) in an object. If `m` is a `MagicDict`, then whenever we read from a field of `m` (e.g., when we evaluate `m.f`), Python calls `m.__getattribute__('f')`, which defaults to an internal mechanism in Python that reads out the value of the field of that name or raises an exception. Our implementation overrides this behavior and extends it: whenever we are reading or calling an attribute that is not defined or inherited in the `MagicDict` class, our code instead interprets the attribute name as a key of the underlying dictionary (lines 10–11). We see the effect of this behavior in Listing 2, lines 5 and 6: our `MagicDict` allows us to use `m.foo` as an alternative to `m['foo']` to look up the key `'foo'` in the `MagicDict m`.

Class `MagicDict` overloads the infix subtraction operator in line 13 and defines an operation that allows "subtracting" a `dict` from a `MagicDict`. Our implementation is quite simplistic: if `m1` is

```
 1 >>> m = MagicDict({'bar': 2})
 2 >>> m['foo'] = 1
 3 >>> print(m['foo'])
 4 1
 5 >>> print(m.foo)            # Calls m.__getattribute__('foo')
 6 1
 7 >>> print(m)
 8 {'foo': 1, 'bar': 2}
 9 >>> print(m – { 'bar': 2 })  # Calls m.__sub__({ 'bar': 2 })
10 {'foo': 1}
```

Listing 2. Interactive use of the **`MagicDict`** class from Listing 1. Lines 1–4 demonstrate standard **`dict`** features.

a `MagicDict` and `m2` is a `dict` or an object that behaves similarly, then `m1 - m2` returns a copy of `m1`, but without any keys that are also present in `m2`, as shown in Listing 2; lines 9–10.

Line 13 also illustrates Python's type annotations, annotating the parameter `other` with type `dict`. By default, such annotations have no runtime effect, but DSL designers can access and repurpose them to collect DSL-specific information without interference from Python. With Python 3.5 (with extensions in 3.9), these annotations also allow type parameters (e.g., `x: list [int]`).

Python also permits the dynamic construction of classes (and metaclasses), which we have found particularly valuable for the robotics domain; since the system configuration and world model used in robotics are often specified outside of Python (e.g., in configuration files or ontologies) but are critical to program logic, we can map them to suitable type hierarchies at robot pre-launch time (just after build time).

## 3.2 Robotics DSL design patterns

In the following section, we list our DSL design patterns. A brief summary that highlights each pattern's purpose and key implementation concepts can be found in Table 1.

```
 1 class MagicDict(dict):
 2 
 3   # When we call a method or read a field from a MagicDict, this method finds the field/method by name
 4   def __getattribute__(self, aname):
 5     # Is the attribute (i.e., field or method) of name 'aname' defined?
 6     if hasattr(super(MagicDict, self), aname):  # Using super(MagicDict, self) forces hasattr() to use Python's built–in
 7                                                 # object.__getattribute__() operation to check if attribute aname exists
 8       # Use Python's default behavior to read the attribute:
 9       return object.__getattribute__(self, aname)
10     else:
11       return self[aname]   # No such attribute exists?  Then read from dict entries instead
12 
13   # Returns a MagicDict with all elements from 'self' that are not also in 'other'
14   def __sub__(self, other : dict):
15     return MagicDict({
16       k : self[k] for k in self.keys()
17             if k not in other  })
```

Listing 1. An example of DSL-friendly Python features: Lines 4–11 show runtime reflection, and line 14 shows a custom infix operator definition and a type annotation.

TABLE 1 Patterns summary.

| Pattern | Purpose | Implementation |
|---|---|---|
| Domain language mapping | Make domain notation visible in host language and reduce notational overhead | See the "Piggyback" DSL implementation pattern documented by Spinellis (2001) |
| Staged verification | Detect type and configuration errors in a critical piece of code early, such as during robot pre-launch time, with no or minimal extra effort for developers | Execute all critical pieces of code early, while redefining the semantics of the predetermined set of operations (e.g., ontology relations from our previous example) to immediately return or to only perform checking |
| Symbolic tracing | Detect bugs in a critical piece of code early, if that code depends on parameters or operation return values, with minimal extra effort for developers | Execute the critical code while passing symbolic values as parameters and/or returning symbolic values from operations of relevance |
| | | Collect any constraints imposed by operations on the symbolic values |
| | | After executing the critical code, verify the constraints against the problem domain |
| Source provenance tracking | Make early dynamic error reports more actionable by reporting relevant source locations | Dynamic stack inspection |

### 3.2.1 Domain language mapping

Domain language mapping identifies language concepts in the host language that correspond to the domain language in some sense and then uses the techniques described in Spinellis (2001) to implement them. This mapping can be manual or the result of reflection.

As an example, the Web Ontology Language (OWL) allows us to express the relationships and attributes of the objects in the world, the robot hardware, and the robot's available capabilities (skills and primitives). Existing libraries like *owlready2* (Lamy, 2017) already expose these specifications as Python objects, so if the ontology contains a class `pkg:Robot`, we can create a new "robot" object by writing

`r = pkg.Robot ("MyRobotName")`

and iterate over all known robots by writing

`for robot in pkg.Robot.instances ():` …

Although Moghadam et al. (2013) expressed concerns about "syntactic noise" for DSL embedding in earlier versions of Python, when compared to external DSLs, we found such noise to be modest in modern Python and instead emphasize the advantages of embedding in a language that is already integrated into the ROS environment and developers are familiar with.

#### 3.2.1.1 Maintenance and integration considerations

When domain knowledge is available in the machine-readable form, much or all of the mapping process may be automatable. For example, the *owlready2* library creates these classes at runtime based on the contents of the ontology specification files. Thus, changes in the ontology are immediately reflected in Python; if we rename `pkg:Robot` in the ontology, our earlier code example will trigger an error when it encounters `pkg.Robot` in the Python source code.

Another strategy for automating the mapping process is to generate the code in the host language. In our example, this code would take the form of Python modules, such as `pkg.py`, which contain classes and methods to reflect the mapping (e.g., a `class Robot`). This strategy mirrors the DSL implementation strategies for host languages that lack advanced reflection facilities, such as C (Levine et al., 1992).

Code generation has two potential disadvantages over reflection. First, code generation persists a *snapshot* of the domain language mapping. The build and development process must thus ensure that this snapshot is kept fresh and prevents developers from accidentally modifying the generated code. Second, code generation requires the domain language mapping to take place *before* build time. When the domain knowledge is only available at pre-launch time, the generated code will necessarily be stale, which may render this implementation strategy useless.

In our discussions with practitioners, we did however observe a key advantage that code generation offers. Since the mapping becomes visible as the Python source code, it is also available to language servers and integrated development environments and may help developers find bugs in their code even earlier.

### 3.2.2 Staged verification

Staged Verification verifies certain kind of properties in a critical piece of code at an early stage before execution. The term "staged" refers to performing these verifications in a controlled manner at a specific early point in the process. This approach prevents runtime failures, simplifies debugging, and enables safe validation in systems that integrate complex elements. In tools like SkiROS2, combining Python code, ontologies, and configuration files at runtime introduces points of failure. To detect such failures early, we propose the following second pattern:

```
1  def expand(self, skill):
2    skill.setProcessor(SerialStar())
3    skill(
4      self.skill("Navigate", ""),
5      self.skill("WmSetRelation", "wm_set_relation",
6        remap={'Dst': 'TargetLocation'},
7          specify={'Src': self.params["Robot"].value,
8          'Relation': 'skiros:at', 'RelationState': True}),)
```

Listing 3. Constructing the behavior tree of a drive skill in SkiROS2: It is a sequential execution of a compound skill (a skill with its own BT of smaller, executable skills) "Navigate" and a primitive skill (an atomic skill that cannot be broken down into smaller parts) to update the world model "WmSetRelation."

The *conditions* for this pattern are as follows:

- We can collect all critical pieces of code at a suitably early point during execution.
- The critical code does not depend on return values of operations that we cannot predict at the pre-launch time.

In Python, configuration and type errors only trigger software faults once we run the code that depends on faulty data. In robotics, we might find such code in operations that (a) run comparatively late (e.g., several minutes after the start of the robot) and (b) are difficult to unit-test (e.g., due to their coupling to specific ROS functionality and/or robotics hardware). For robotics developers, both challenges increase the cost of verification and validation (Reichenbach, 2021). A fault might trigger only after a lengthy robot program and require substantial manual effort to reproduce. For example, a software module for controlling an arm might take a configuration parameter that describes the target arm pose. If the arm controller is triggered late (e.g., because the arm is part of a mobile platform that must first reach its goal position), any typos in the arm pose will also trigger the fault late. If the pose description comes from a configuration file or ontology, traditional static checkers will also be ineffective. We can only check for such bugs after we have loaded all relevant configurations.

Through careful software design, developers can work around this problem, e.g., by checking that code and configuration are well-formed as soon as possible, before they run the control logic. If the critical code itself is free of external side effects, the check can be as simple as running the critical code twice. For example, SkiROS2 composes *BTs* (Colledanchise and Ögren, 2018) within such critical Python code (Listing 3); composing (as opposed to running) these objects has no side effects, so we can safely construct them early to detect simple errors (e.g., typos in parameter names). This is a typical example that eludes static checking but is amenable to early dynamic checking. Line 7 depends on `self.params ["Robot"].value`, which is a configuration parameter that we cannot access until the robot is ready to launch.

Not all of the robotics code is similarly declarative. Consider the following example, in a hypothetical robotics framework in which all operations are subclasses of `RobotOp` and must provide a method `run ()` that takes no extra parameters.

```
class MyRobotOp(RobotOp):
  def __init__(self, config): # Configure
    self.config = config
  def check(self):  # Check configuration
    assert self.config.mode in ["A", "B"]
    assert isinstance(self.config.v, int)
  def run(self):    # Run with configuration
    if self.config.mode == "A":
      self.runA();
    elif self.config.mode == "B":
      self.runB(self.config.v + 10);
    else:
      fail()
```

Here, developers introduced a separate method `check ()` that can perform an early check during robot initialization or pre-launch. However, `check ()` and `run ()` both have to be maintained to make the same assumptions.

The early dynamic checking pattern instead uses internal DSL techniques to enable developers to use the same code in two different ways: (a) for checking and (b) for logic.

In our example, calling `run ()` "normally" captures case (b). For case (a), we can also call `run ()`, but instead of passing an instance of `MyRobotOp`, we pass a *mock* instance of the same class, in which operations like `runA ()` immediately return.

```
class MyRobotOpMock:
  def __init__(self, parent):
    self.parent = parent
  @property
  def config(self):
    # self.config  = self.parent.config
    return self.parent.config
  def runA(self):
    pass # mock operation: do nothing
  def runB(self, arg):
    pass # mock operaiton: do nothing
```

If we execute `MyRobotOpMock.run ()` with the same configuration as `MyRobotOp`, `run ()` will execute almost as for `MyRobotOp` but immediately return from any call to `runA` or `runB`. If the configuration is invalid, for example, if `config.mode == "C"` or `config.v == false`, running `MyRobotOpMock.run ()` will trigger the error early.

Since Python can reflect on a class or an object to identify all fields and methods, we can construct classes like `MyRobotOpMock` at runtime; instead of writing them by hand, we can implement a general-purpose mock class generator that constructs methods like `runA` and accessors like `config` automatically. If the configuration objects trigger side effects, we can apply the same technique to them.

However, the above implementation strategy is only effective if we know that the critical code will only call methods on `self` and other Python objects that we know about ahead of time. We can relax this requirement by controlling how Python resolves nonlocal names:[7]

`FunctionType(MyRobotOp.run.__code__, globals() | {'print': g})(obj)`

This code will execute `obj.run ()` via the equivalent `MyRobotOp.run (obj)` but replace all calls to `print` by calls to some function `g`. The same technique can use a custom map-like object to detect at runtime which operations the body of the method wants to call and handle them suitably.

However, the more general-purpose we want the critical code to be, the more challenging it becomes to apply this pattern. For instance, if the critical code can get stuck in an infinite loop, so may the check; if this is a concern, the check runner may need to use a heuristic timeout mechanism. A more significant limitation is that we may not, in general, know what our mocked operations like `runA ()` should return, if anything. If the critical code depends on a return value (e.g., if it reads ROS messages), the mocked code must be able to provide suitable answers. The same limitation arises when the critical code is in a method that takes parameters. If we know the type of the parameter or return value, e.g., through a type annotation, we can exploit this information to repeatedly check (i.e., *fuzz-test*) the critical code with different values; however, without further cooperation from developers, this method can quickly become computationally prohibitive.

If we know that the code in question has a simple control flow, we may be able to apply the next pattern, symbolic tracing.

### 3.2.3 Symbolic tracing

Here, a *symbolic* value is a special kind of mock value that we use to record information (King, 1976).

The *conditions* for this pattern are as follows:

- We can access and execute the critical code.
- We have access to sufficient information (via type annotations and properties) to simulate parameter values and operation return values *symbolically* (see below).
- The number of control flow paths through the critical code is small (see below).

Consider the following `RobotOp` subclass:

```
class SetArmSpeedOp(RobotOp):
 def run(self, speedup):
  self.setArmSpeed(speedup)
  self.setArmSafety(speedup)
```

This class only calls two operations, but its `run` operation depends on a parameter `speedup` about which we know nothing a priori—thus, we cannot directly apply the early dynamic checking pattern.

In cases where we lack prior knowledge about an operation, it may still be possible to obtain useful insights about it. For example, if we are aware that `setArmSpeed` accepts only numeric parameters and `setArmSafety` only accepts Boolean parameters, we can flag this code as having a type error. To avoid blindly testing various parameters, we can pass a symbolic parameter to the run function and employ a modified version of the mock-execution strategy used in early dynamic checking. The mock objects can be adapted as follows:

```
TYPE_CONSTRAINTS = []

class SetArmSpeedOpMock:
 def setArmSpeed(self, obj):
  TYPE_CONSTRAINTS.append((obj, float))
 def setArmSafety(self, obj):
  TYPE_CONSTRAINTS.append((obj, bool))
```

We can now 1) create a fresh object `obj` and an `SetArmSpeedOpMock` instance that we call `mock`, 2) call `SetArmSpeedOp.run (mock, obj)`, and 3) read out all constraints that we collected during this call from `TYPE_CONSTRAINTS` and check them for consistency, which makes it easy to spot the bug. If the constraints come from accesses to `obj` (e.g., method calls like `obj.__add__(1)` that result from code like `obj + 1`), `obj` itself can collect the resultant constraints.

Depending on the problem domain, constraint solving can be arbitrarily complex, from simple type equality checks to automated satisfiability checking (Balldin and Reichenbach, 2020). It can involve dependencies across different pieces of the critical code (e.g., to check if all components agree on the types of messages sent across ROS channels or to ensure that every message that is sent has at least one reader). However, this approach requires information about specific operations like `setArmSpeed` and `setArmSafety`, which can be provided to Python in a variety of ways, e.g., via type annotations.

As an example, consider an operation that picks up a coffee from the table with a gripper, where we annotate all parameters to `run` with the Web Ontology Language (OWL) ontology types.

```
1 class PickCoffeeTableOp(RobotOp):
2  def run(self, robot : rob.Robot,
3        gripper : rob.Gripper,
4        coffee_table : world.Furniture):
5   // bug:
6   assert coffee_table.robotPartOf(robot);
7   ...
```

This example is derived from the SkiROS2 ontologies, with minor simplifications. In the above SkiROS2 code, the developer intended to write a pre-condition that to be able to pick a coffee cup, the robot should be close to the table. Instead, the developer mistakenly wrote that a robot should be a part of the coffee table.

The ontology requires that `robotPartOf` is a relation between a technical `Device` and a `Robot`. However, `Furniture` is not a subtype of `Device`, so the assertion in line 6 is unsatisfiable.

7 Python's *eval* function offers similar capabilities, but as of Python 3.10, it does not seem to allow passing parameters to code objects.

We can again detect this bug through symbolic tracing. This time, we must construct symbolic variables for `robot`, `gripper`, and `coffee_table` that expose methods for all applicable relations, as described by their types. For instance, `gripper` will contain a method `robotPartOf(gripper, obj)` that records on each call that `gripper` and `obj` should be in a `robotPartOf` relation. Meanwhile, `coffee_table` will not have such an operation. When we execute `run ()`, we can then defer to Python's own type analysis, which will abort execution and notify us that `coffee_table` lacks the requisite method.

Key to this symbolic tracing is our use of mock objects as symbolic variables. Symbolic variables reify Python variables to objects that can trace the operations that they interact with, in execution order, and translate them into constraints.

The main *limitation* of this technique stems from its interaction with Python's Boolean values and control flow, e.g., conditionals and loops. Python does not allow the Boolean operators to return symbolic values but instead forces them (at the language level) to be `bool` values; similarly, conditionals and loops rely on access to Boolean outcomes. Thus, when we execute the code in the form `if x: …`, we must decide right there and then if we should collapse the symbolic variable that `x` is bound to `True` or `False`. Although we can re-run the critical code multiple times with different decisions per branch, the number of runs will in general be exponential over the number of times that a symbolic variable collapses to `bool`.

### 3.2.4 Source provenance tracking

The intent in early error detection in (embedded) DSLs is generally to prevent undesirable behavior. When this undesirable behavior is due to a problematic user specification, it is—in our experience—valuable to point the user to the problematic specification. In practice, "blaming" the right part of the program can be non-trivial since the disagreement may be across multiple user specifications (Ahmed et al. (2011) discussed this challenge in more detail).

Handling multiple conflicting constraints can be particularly challenging for embedded DSLs. Let us say that we are using a technique like *symbolic tracing* in two user-defined functions, namely, declaration () and implementation (), such that implementation () must *ensure* the constraints that are *required* declaration ().

```
def declaration(x : int):
  require(x > 0)
  require(x < 10)

def implementation(x : int):
  ensure(x > 3)
  ensure(x <= 10)

...
# more code follows
...
register_implementation(declaration, implementation)
```

In the above example, we might find a bug: implementation allows x = 10, but this is not allowed according to declaration (). A typical but naïve implementation of such a consistency check might simply inform the user that declaration and implementation disagree about what x is allowed to do and raise an exception.

The programmer must now identify the line of code that is the culprit by hand. In practical scenarios, such as our case studies, there may be multiple declaration and implementation functions in the same file (usually as methods), which further complicates the task.

Reflection can help us here; for example, given a function object in Python, we can use reflection to access `implementation.__code__.co_firstlineno` and `implementation.__code__.co_filename` to obtain the location at which the function was defined in the form of the first line of the code and the source file name. For larger definitions, even this information may be insufficiently precise.

Some languages offer facilities that allow us to obtain even the exact lines of code that were responsible for the error (lines 3 and 7, in our example). Although some languages support this inspection through macro- or pre-processor facilities (e.g., `__LINE__` and `__FILE__` in C), Python 3.1 and later versions offer direct read access to the call stack via `inspect. stack ()`. The symbolic tracing code for `require ()` and `ensure ()` can then "walk" this stack down until it finds the first stack frame that belongs to the code under analysis and extract file name and line number from there. The symbolic tracer can then attach this *provenance* information to the constraint and expose it to the user if the constraint is contributing to some error report.

## 3.3 Alternative techniques for checking

Internal DSLs are not the only way to implement the kind of early checking that we describe. The *mypy* tool[8] is a stand-alone program for the type-checking Python code. *mypy* supports plugins that can describe custom typing rules, which we could use, e.g., to check for ontology types. Similarly, we could use the Python `ast` module to implement our own analysis over the Python source code. However, both approaches require separate passes and would first have to be integrated into the ROS launch process. Moreover, they are effectively static, in that they cannot communicate with the program under analysis; thus, we cannot guarantee that the checker tool will see the same configuration (e.g., ontology and world model).

Another alternative would be to implement static analysis over the bytecode returned by the Python disassembler `dis`, which can operate on the running program. However, this API is not stable across Python revisions[9].

An external DSL such as MAESTRO Coste-Maniere and Turro (1997) would similarly require a separate analysis pass. However, it would be able to offer arbitrary, domain-specific syntax and avoid any trade-offs induced by the embedding in Python (e.g., Boolean coercions). The main downside of this technique is that it requires a completely separate DSL implementation, including maintenance and integration.

8 https://mypy-lang.org/
9 https://docs.python.org/3/library/dis.html

## 4 SkiROS2: an open-source software for skill-based robot execution

As a case study, we implement our patterns on the open-source software for skill-based robot execution SkiROS2 (Mayr et al., 2023b). SkiROS2 is used by several research institutions in the context of industrial robot tasks, as demonstrated in Mayr et al. (2022a), Mayr et al. (2022b), Mayr et al. (2023c), Mayr et al. (2023a), Ahmad et al. (2023), and Wuthier et al. (2021). It is a re-implementation of the predecessor SkiROS1 by Rovida et al. (2017a) and is implemented in Python on top of the robot operating system (Quigley et al., 2009) middleware. SkiROS2 uses behavior tree (Colledanchise and Ögren, 2018) formalism to represent procedures.

SkiROS2 implements a layered, hybrid control architecture to define and execute parametric *skills* for robots (Bøgh et al., 2012), Krueger et al. (2016). The SkiROS2 system architecture is shown in Figure 2, which illustrates how different components interact with each other in various phases. It uses *ontologies* to represent the comprehensive knowledge about the world. SkiROS2 represents knowledge about the skills, the robot, and the environment in a *world model* (WM) with the *ontologies* specified in the OWL format. This explicit representation, built on the World Wide Web Consortium's Resource Description Framework (RDF) (Hitzler et al., 2009) standard, allows the use of existing *ontologies*. This approach to knowledge management is important for complex decision-making and reasoning in autonomous systems (Cangelosi and Asada, 2022). WM is central to SkiROS2's architecture and serves as a dynamic repository of the robot's environment and state. It continuously updates and maintains a semantic representation of the surroundings, objects, and the robot's own status. The integration of the WM with the *ontologies* shown in Figure 2 ensures that the robot has a thorough understanding of its operational context, enhancing its interaction capabilities with the environment.

### 4.1 Skill model

*Skills* in SkiROS2 are parametric procedures that modify the world state from an initial state to a final state according to pre- and post-conditions (Pedersen et al., 2016). Every skill has a *Skill Description* and one or more *Skill Implementation* as shown in Figure 2. The *Skill Description* consists of the following four elements:

1. *Parameters* define the input and output of a skill. The types of these parameters can vary from certain primitive data types to a *world model* element in the *ontologies*.
2. *Pre-conditions* must hold before the skill is executed.
3. *Hold-conditions* must be fulfilled during the execution.
4. *Post-conditions* indicate that a skill has been successfully executed.

These conditions are checked by the *Skill Manager* as shown in Figure 2. These conditions are important for planning and also for dynamic sanity checks, when planning is disabled. When a skill is invoked, the system first checks the pre-conditions to decide if it is safe or viable to start the skill. During execution, hold conditions are continuously monitored to ensure ongoing criteria are met. Finally, once the skill reports its completion, post-conditions are checked to confirm successful execution. These checks are essential to maintain the robustness, safety, and reliability of robotic operations, ensuring that skills are only performed when appropriate and achieve the intended results.

#### 4.1.1 Skill description

Listing 4 shows how developers define a "pick" skill in SkiROS2 by calling the Python method addParam to set the parameters of the skill and similarly to define its pre- and post-conditions. The parameters are typed, using basic datatypes (e.g., `str`) or a WM element defined in *ontology*, and can be *required*, *optional*, or *inferred* from the world model. Pre-conditions allow SkiROS2 to check requirements for skill execution and to automatically infer skill parameters from the world model. For example, in the "pick" skill shown in Listing 4, the parameter `"Object"` in line 10 is Required, i.e., it must be set before the execution of the skill. At execution time, SkiROS2 infers the parameter `"ObjectLocation"` (line 9) by reasoning about the pre-condition rule `"ObjectLocationContainObject"` (line 13). If `"Object"` is semantically not at a location in the WM, the pre-conditions are not satisfiable and the skill cannot be executed.

#### 4.1.2 Skill Implementation

The *Skill Implementation*, on the other hand, acts as a class that implements the interface *Skill Description* and refers to the actual coding and logic that enables a robot to perform a task. Skills can be either primitive or compound skills. Depending on the type of skill, primitive skills implement atomic functions that change the real world, such as moving a robot arm, whereas compound skills build complex behaviors in a BT. An example of a pick *Skill Implementation* is shown in Listing 5.

The *createDescription* method (line 2 in Listing 5) sets the description (interface) to an implementation. The *expand* method (line 5 in Listing 5) within the skill implementation uses behavior trees to structure the execution of skills. Each node in the tree could represent a specific skill (action node) or a decision-making process (commonly known as a control flow node) that determines which skill to execute next, as illustrated in Figure 3. The control flow node sets the processor and specifies how the compound skill is decomposed into a behavior tree (line 6). In SkiROS2, control flow nodes or processors dictate how a compound skill invokes its child skills. Before delving into specific processors, it is essential to understand the common states in which a node might return during execution.

- *Success* indicates that the skill or all skills (in case of compound skills) have been completed successfully.
- *Failure* indicates that the skill has failed to complete successfully or conditions for success are not met.
- *Running* indicates that the skill is still in progress and has not yet reached a conclusion of success or failure.

These states are not only exclusive to compound skills but are also applicable to leaf nodes/primitive skills. Following are the lists of processors and how they operate in these states:

- *Serial* processes the children one by one in order until all succeed. It will continuously loop through the children until

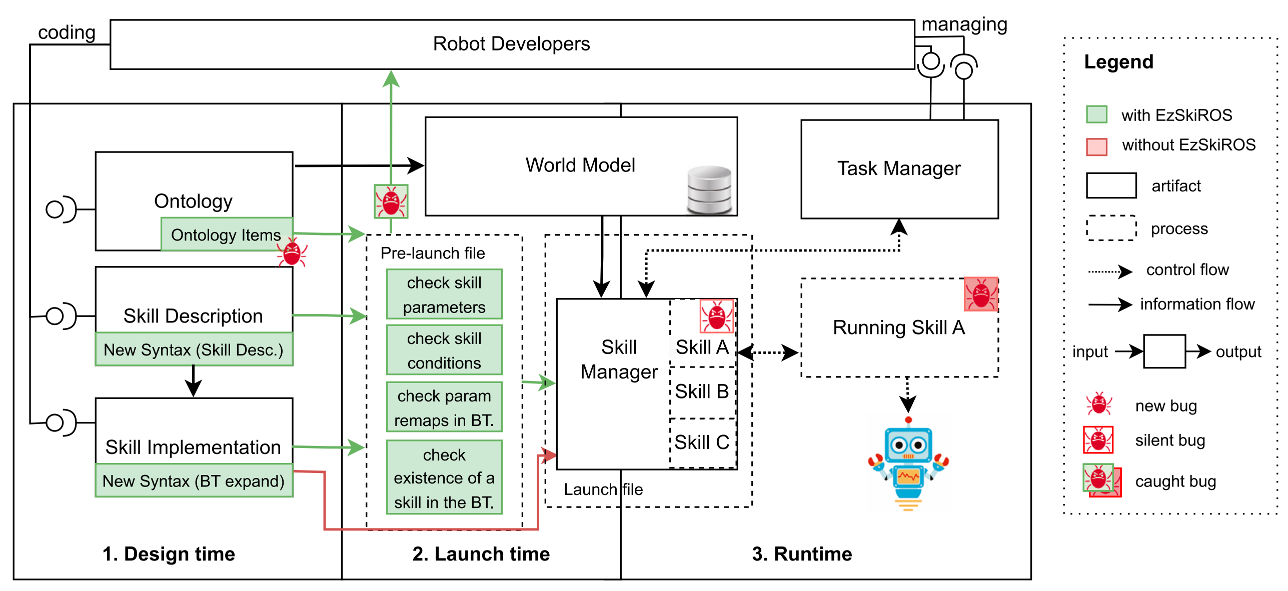


FIGURE 2
Diagram with the different components of SkiROS2, their interaction during different time phases, and the advancements by EzSkiROS (shown as green blocks). In SkiROS2, a bug that has been introduced in a skill description by a developer will often only trigger at runtime. EzSkiROS addresses these costs and risks by adding checks to find a wide range of bugs by running a pre-launch file where the skills are loaded before runtime.

one returns RUNNING or FAILURE or until all children succeed. *SerialStar* is a variation of the serial processor with error handling.

- *Selector* runs its children one after the other until one succeeds (returning SUCCESS) or all fail (returning FAILURE). If a child is in progress (RUNNING), the processor will also return RUNNING. *SelectorStar* is a variation of *Selector* analogous to *SerialStar*.
- *ParallelFf* (parallel first fail) invokes all the children at the same time. It returns SUCCESS only if all children succeed. If any child fails, it immediately returns FAILURE and halts the other children.
- *ParallelFs* (parallel first stop) also runs all the children simultaneously. However, it stops all processes and returns SUCCESS as soon as any child succeeds or FAILURE if any child fails, regardless of the others' states.

```
class Pick(SkillDescription):
 def createDescription(self):
  self.addParam("Robot", Element("cora:Robot"), ParamTypes.Inferred)
  self.addParam("Arm", Element("rparts:ArmDevice"), ParamTypes.Inferred)
  self.addParam("StartPose", Element("skiros:TransformationPose"), ParamTypes.Inferred)
  self.addParam("GraspPose", Element("skiros:GraspingPose"), ParamTypes.Inferred)
  self.addParam("ApproachPose", Element("skiros:ApproachPose"), ParamTypes.Inferred)
  self.addParam("Workstation", Element("scalable:Workstation"), ParamTypes.Inferred)
  self.addParam("ObjectLocation", Element("skiros:Location"), ParamTypes.Inferred)
  self.addParam("Object", Element("skiros:Product"), ParamTypes.Required)
  self.addParam("Gripper", Element("rparts:GripperEffector"), ParamTypes.Required)

  self.addPreCondition(self.getRelationCond("ObjectLocationContainObject", "skiros:contain", "ObjectLocation", "Object", True))
  self.addPreCondition(self.getRelationCond("GripperAtStartPose", "skiros:at", "Gripper", "StartPose", True))
  self.addPreCondition(self.getRelationCond("NotGripperContainObject", "skiros:contain", "Gripper", "Object", False))
  self.addPreCondition(self.getRelationCond("ObjectHasAApproachPose","skiros:hasA", "Object", "ApproachPose", True))
  self.addPreCondition(self.getRelationCond("ObjectHasAGraspPose", "skiros:hasA", "Object", "GraspPose", True))
  self.addPreCondition(self.getRelationCond("RobotAtWorkstation", "skiros:at", "Robot", "Workstation", True))
  self.addPreCondition(self.getRelationCond("WorkstationContainObjectLocation", "skiros:contain", "Workstation", "ObjectLocation", True))
  self.addPostCondition(self.getRelationCond("NotGripperAtStartPose", "skiros:at", "Gripper", "StartPose", False))
  self.addPostCondition(self.getRelationCond("GripperAtApproachPose", "skiros:at", "Gripper", "ApproachPose", True))
  self.addPostCondition(self.getRelationCond("NotObjectContainedObjectLocation", "skiros:contain", "ObjectLocation", "Object", False))
  self.addPostCondition(self.getRelationCond("GripperContainObject", "skiros:contain", "Gripper", "Object", True))
```

**Listing 4. An excerpt of the parameters and pre- and post-conditions of a pick skill in SkiROS2 without EzSkiROS. It depends heavily on the usage of strings to refer to parameters or classes in the ontology.**

When we say that a node "returns" something, we are referring to the result of an operation or computation performed by that node. This result dictates the next action in the behavior tree, such as whether to continue, stop, or try a different approach.

As shown in Listing 5, the `skill()` operator allows us to set the children of the behavior tree of the skill being implemented. To add several children at once, it is possible to use the syntax shown in the Listing 5 (lines 9–22). Each child can either be another processor (to make a nested control structure) or an individual component skill. Individual skills follow the template `self.skill (skilltype, label = " ", specify = {}, remap = {})`, where `skilltype` is a *Skill Description*, i.e., an abstract skill that may have multiple implementations. At runtime, SkiROS2 selects and substitutes one of the implementations of this skill description, unless users manually select a specific implementation using the optional `label` parameter. All skills share a parameter namespace so that parameters with the same names are implicitly unified across all component skills. For example, if we use a compound skill with the parameter Robot set to some specific object, SkiROS2 implicitly sets this parameter in all component skills. Skill developers can override this behavior with the optional `specify` and `remap` parameters to `self.skill.specify` takes a Python dictionary that maps parameter names to concrete values (e.g., the `Duration` of a `Wait` action, in line 19 of Listing 5). Meanwhile, `remap` maps parameter names to the names of other parameters. Considering line 15 shown in Listing 5, this line specifies that the parameter `Target` of the `ApproachMovement` skill should be the parameter `GraspPose`, whereas the same parameter for the same skill in line 23 should be the parameter `ApproachPose`.

The relationship between *Skill Descriptions* and BTs is evident in how the *expand* function uses the behavior tree structure to implement the skill logic. The parameters, pre-conditions, hold-conditions, and post-conditions defined in the *Skill Description* guide the construction and execution of BTs. For instance, the pre-conditions in a skill description determine when a particular branch of the behavior tree is activated, and the post-conditions signal when a skill or sequence of skills has been successfully completed.

These skills are loaded by the *Skill Manager* at robot launch time (shown in Figure 2).

## 5 Case study I: concise and verifiable robot skill interface

We have validated our design patterns in an internal DSL EzSkiROS, which adds early dynamic checking (Section 3.2) to skill descriptions. Following a user-centered design methodology, we developed EzSkiROS by first identifying needs for early bug checking via semi-structured interviews with skilled roboticists who use SkiROS2, reviewed documentation, and manual code inspection. We found that even expert skill developers made errors in writing Skill Descriptions and that Python's dynamic typing only identified bugs when they triggered faults during robot execution.

We designed EzSkiROS to simplify how Skill Descriptions are specified, with the intent to increase their readability,

```
 1 class pick(SkillBase):
 2     def createDescription(self):
 3         self.setDescription(Pick(), self.__class__.__name__)
 4
 5     def expand(self, skill):
 6         skill.setProcessor(SerialStar())
 7         skill(
 8             self.skill("SwitchController", "", specify={'Controller': 'joint_config'}),
 9             self.skill("MoveitCartesianSpaceMotion", "", remap={'GoalPose': 'ApproachPose'}),
10             self.skill("WmSetRelation", "wm_set_relation",
11                        remap={'Src': 'Gripper', 'Dst': 'ApproachPose'},
12                        specify={'Relation': 'skiros:at', 'RelationState': True}),
13             self.skill("HSVDetection", ""),
14             self.skill("SwitchController", "", specify={'Controller':'compliant'}),
15             self.skill("ApproachMovement", "go_to_linear", remap={'Target':'GraspPose'}),
16             self.skill("WmSetRelation", "wm_set_relation",
17                         remap={'Src': 'Gripper', 'Dst': 'GraspPose'},
18                         specify={'Relation': 'skiros:at', 'RelationState': True}),
19             self.skill("Wait", "", specify={"Duration": 2.0}),
20             self.skill("ActuateGripper", "", specify={'Open': False}),
21             self.skill("WmMoveObject", "wm_move_object",
22                         remap={ "TargetLocation": "Gripper"}),
23             self.skill("ApproachMovement", "go_to_linear", remap={'Target':'ApproachPose'}),
24             self.skill("Wait", "", specify={"Duration": 2.0})
25         )
```

Listing 5. The skill implementation of the pick *Skill Description* is shown in Listing 4.

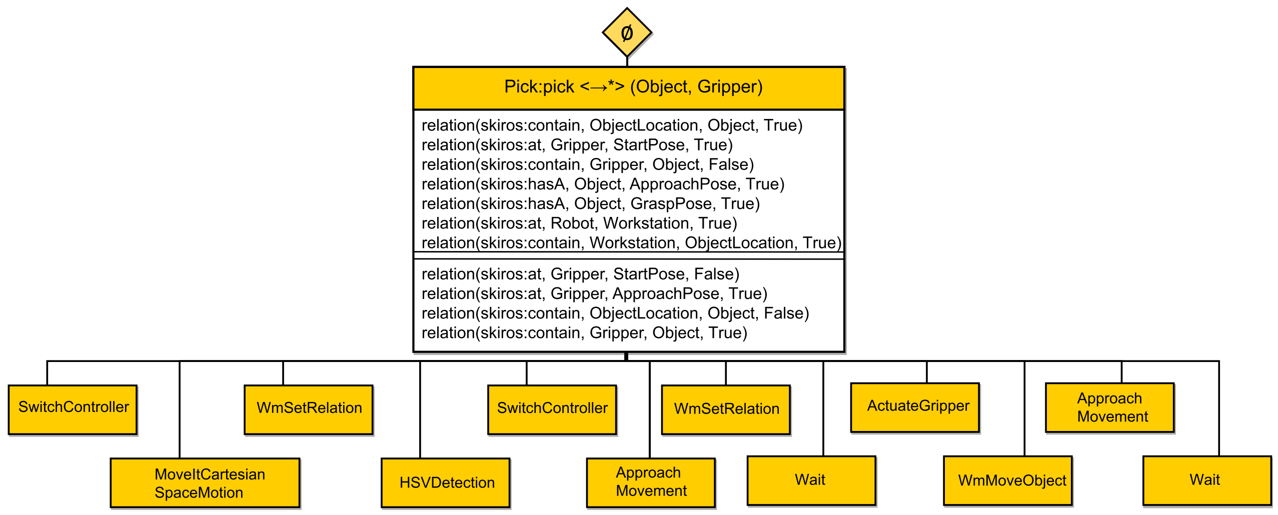

∅
Pick:pick <→*> (Object, Gripper)
relation(skiros:contain, ObjectLocation, Object, True)
relation(skiros:at, Gripper, StartPose, True)
relation(skiros:contain, Gripper, Object, False)
relation(skiros:hasA, Object, ApproachPose, True)
relation(skiros:hasA, Object, GraspPose, True)
relation(skiros:at, Robot, Workstation, True)
relation(skiros:contain, Workstation, ObjectLocation, True)
relation(skiros:at, Gripper, StartPose, False)
relation(skiros:at, Gripper, ApproachPose, True)
relation(skiros:contain, ObjectLocation, Object, False)
relation(skiros:contain, Gripper, Object, True)
SwitchController
MoveItCartesian SpaceMotion
WmSetRelation
HSVDetection
SwitchController
Approach Movement
WmSetRelation
Wait
ActuateGripper
WmMoveObject
Approach Movement
Wait

FIGURE 3
BT of the pick skill in the *eBT* format Rovida et al. (2017b). It has a *SerialStar* operator and will execute all children in sequence. The pre-conditions and post-conditions are shown.

```
1  class Pick(SkillDescription):
2      def description(self,
3          Robot: INFERRED[cora.Robot],
4          Arm: INFERRED[rparts.ArmDevice],
5          StartPose: INFERRED[skiros.TransformationPose],
6          GraspPose: INFERRED[skiros.GraspingPose],
7          ApproachPose: INFERRED[skiros.ApproachPose],
8          Workstation: INFERRED[scalable.Workstation],
9          ObjectLocation: INFERRED[skiros.Location],
10         Object: skiros.Product,
11         Gripper: rparts.GripperEffector):
12
13       self.pre_conditions += ObjectLocation.contain(Object)
14       self.pre_conditions += Gripper.at(StartPose)
15       self.pre_conditions += ~ Gripper.contain(Object)
16       self.pre_conditions += Object.hasA(ApproachPose)
17       self.pre_conditions += Object.hasA(GraspPose)
18       self.pre_conditions += Robot.at(Workstation)
19       self.pre_conditions += Workstation.contain(ObjectLocation)
20       self.post_conditions += ~ Gripper.at(StartPose)
21       self.post_conditions += Gripper.at(GraspPose)
22       self.post_conditions += ~ ObjectLocation.contain(Object)
23       self.post_conditions += Gripper.contain(Object)
```

**Listing 6. The skill description of the pick skill is shown in Listing 4 with EzSkiROS. We represent OWL classes in Python as identifiers in type declarations.**

maintainability, and writability. We map ontology objects and relations into Python's type system. Skill Descriptions can then directly include ontology information in type annotations. This approach streamlines the syntax by avoiding redundant syntactic elements and specifying type information through annotations rather than string encodings, as illustrated with the example of the pick skill in Listing 6. The listing also illustrates the EzSkiROS syntax for the example of the pick skill from Listing 4. The Skill Description shown in Listing 6 is more concise and intuitive, with type annotations providing a clear and direct way to specify the types of parameters and their ontology information.

In EzSkiROS, we employed *owlready2*'s approach to domain language mapping in exposing the world model elements in the ontology as Python types and objects. For instance, as shown in Listing 6, line 3 describes a parameter `Robot` with the type annotation `INFERRED [cora.Robot]`. Here, `cora.Robot` is a Python class that we dynamically generate to mirror an OWL class "`Robot`" in the OWL namespace "`cora`". `INFERRED` is a parametric type that tags *inferred* parameters. We mark *optional* parameters analogously as `OPTIONAL`; all other parameters are *required*. At robot pre-launch time, we use Python's reflection facilities to extract and check this parameter information, both to link with SkiROS2′ skill manager and for part of our early dynamic checking. In addition to our ontology types, we also allowed basic data types (`str`, `float`, `int`, `bool`) in EzSkiROS, enforcing that each must specify a default value. Originally, SkiROS2 also allowed the parameters of data types *list* and *dict*. However, in EzSkiROS, we restricted the use of *lists* and *dicts* as it was not clear if we would need this in practice. One of the developers claimed that *dicts* are considered "hacks" in the system's context. Although *lists* are valid for representing, e.g., joint configurations, it might be better served by a specialized joint-configuration type to encapsulate their complexities and intended use more accurately. We allowed *enums* to handle such parameters, acknowledging that enums cannot encode lists or dicts, but it can provide a more controlled and predictable set of values, enhancing the system's integrity and reliability.

In addition to skill parameters, we also want to make sure that skill conditions satisfy the contracts in our ontology. These pre-, post-, and hold-conditions can be expressed in different ways depending on what aspects of the robot's environment and state we want to assess. According to SkiROS2 documentation, one can define a skill with the help of four kinds of skill conditions:

1. *ConditionHasProperty* is a unary relation to check whether a certain element or entity has a specific property. It is useful when the skill needs to verify certain attributes or characteristics of objects or elements before proceeding. When a condition checks for a property, it is essentially querying the ontology to see if the entity conforms to certain criteria or states defined within it. For instance, if an ontology defines that a "door" entity can have a *state* property with values "open" or "closed," *ConditionHasProperty* might check if the door's state is "open."
2. *ConditionProperty* is a binary relation which relies on the ontology to understand and evaluate properties or attributes of entities. However, it might be used to assess the value or state of a property rather than just its presence. For example, it could check whether the temperature (property) is within a certain range.
3. *ConditionRelation* is used to evaluate the relationships between different elements or entities. It is crucial for tasks that require understanding spatial or hierarchical relationships, such as "is next to," "is on top of," or "is part of." This condition utilizes the relational information in the ontology to assess how entities are related. *Ontologies* define not just entities but also the possible relationships between them. For example, it might check if "object A is on top of object B" by referring to the ontology's definitions of "object A," "object B," and "on top of" relations.
4. *AbstractConditionRelation* is a more generalized or template form of *ConditionRelation*, which can be specified or extended for various specific relational conditions.

Since all types of skill conditions rely heavily on the ontology for their evaluation, it is important to add **Early Dynamic checking** to detect mistyped conditions. We utilize **Symbolic Tracing** as described in Section 3.2. This step collects all pre-, post-, and hold conditions via the overloaded Python operator "+=" (lines 13–23). We then check for wrong ontology relations and ontology type errors among these conditions. Since we use **Domain Language Mapping** to expose the world model entities as classes and relations as Python methods, Python's own name analysis will catch such mistyped ontology relation or entity names, and the symbolic values that we pass into the `description` method capture all types of information that we need for type-checking.

We test our DSL implementation by integrating it with SkiROS2 to see how it behaves with a real skill running on a robot[10]. To demonstrate the effectiveness of our type check in EzSkiROS, we use a "pick" skill written in EzSkiROS (Listing 6) and load it while launching a simulation of a robot shown in Figure 1.

Listing 7 shows that the *ObjectProperty* "`hasA`" is a relation allowed only between a "`product`" and a "`TransformationPose`". If we introduce a nonsensical relation like `Object.hasA (Gripper)`, then the early dynamic check in EzSkiROS over ontology types returns a type error:

```
TypeError: Gripper: <class 'ezskiros.param_
type_system.rparts.GripperEffector'> is not
a skiros.TransformationPose
```

10 Available online in https://github.com/lu-cs-sde/EzSkiROS

In addition to the error message, we also provide the source of the error highlighting the line that contains the error.

## 5.1 Evaluation

To evaluate the effectiveness and usability of EzSkiROS in detecting bugs at pre-launch time, we conducted a user study with robotics experts. Seven robotic skill developers participated in our user study, including one member of the SkiROS2 development team. The user study consisted of three phases: an initial demonstration, a follow-up discussion, and a feedback survey[11]. Due to time limitations, we defer a detailed study, with exercises for users to write new skills in EzSkiROS, to the future.

To showcase the embedded DSL and the early bug checking capabilities of EzSkiROS, we presented a video showing 1) a contrast between the old and new skill descriptions written in EzSkiROS and 2) demonstrating how errors in the skill description are detected early at pre-launch time by intentionally introducing an error in the skill conditions.

During the follow-up discussion, we encouraged participants to ask any questions or clarify any confusion they had about the EzSkiROS demonstration video.

After the discussion, we invited the participants to complete a survey to evaluate the readability and effectiveness of the early ontology type checks implemented in EzSkiROS. The survey included Likert-scale questions about *readability*, *modifiability*, and *writability*. Six participants answered "strongly agree" that EzSkiROS improved readability, and one answered "somewhat disagree." For modifiability, four of them "strongly agree," but three participants answered "somewhat agree" and "neutral." All the participants answered "strongly agree" or "somewhat agree" that EzSkiROS improved writability.

To gain more in-depth insights, the survey also included open-ended questions, e.g., a) "Would EzSkiROS have been beneficial to you, and why or why not?"; b) "What potential benefits or concerns do you see in adopting EzSkiROS in your work?;" and c) "What potential benefits or concerns do you see in beginners, such as new employees or M. Sc. students doing project work, adopting EzSkiROS?"

For question a), all participants agreed that EzSkiROS would have helped them. Participants liked the syntax of EzSkiROS, and they thought that it takes less time to read and understand the ontology relations than before. One of them claimed that "pre- and post-conditions are easy to make sense." They also found that mapping the ontology to Python types would have helped reduce the number of lookups required in the ontology. One of the participants said, "in my experience, SkiROS2 error messages are terrible, and half the time they are not even the correct error messages (i.e. they do not point me to the correct cause), so I think the improved error reporting would have been extremely useful."

For question b), the majority of participants reported that EzSkiROS's concise syntax is a potential benefit, which they believe would save coding time and effort. One participant found

11 A replication of the survey https://github.com/lu-cs-sde/EzSkiROS

EzSkiROS's specific error messages useful, responding that "the extra checks allow to know some errors before the robot is started," while one participant answered that EzSkiROS does not benefit their current work, but it might be useful for writing a new skill from scratch. None of the participants expressed any concerns about adopting EzSkiROS in their work.

For question c), one developer acknowledges the benefits of EzSkiROS by saying "In addition to the error reporting, it seems much easier for a beginner to learn this syntax, particularly because it looks more like "standard" object oriented programming (OOP.)" One person claimed that EzSkiROS would help beginners, describing SkiROS2 as "it is quite a learning curve and needs some courage to start learning SkiROS2 from the beginning autonomously."

In summary, the results of the user evaluation survey indicate a positive perception of EzSkiROS in terms of readability and writability. Most respondents found EzSkiROS to be easy to read and understand, with only one exception. In addition, respondents found EzSkiROS's early error checking to be particularly useful in detecting and resolving errors in a timely manner. This suggests that the users perceived EzSkiROS as an effective tool.

## 6 Case study II: verifiable construction of a behavior tree in skill implementation

In the second case study, we examined the utility of our design patterns by extending EzSkiROS to add **Early Dynamic Checking** to the implementation of compound skills. Compound *Skill Implementation* uses behavior trees to efficiently handle decision-making processes, task execution, and error recovery. Our design methodology involved identifying the requirements for constructing BTs by examining their specifications. To understand common challenges, we analyzed GitHub issues encountered by developers when writing BTs in SkiROS2. This analysis included a systematic search for specific keywords such as "Behavior Tree," "Remaps," and "Skill Implementation," informed by insights from senior Ph.D. students. Subsequently, we engaged in a verification process with the developers to ensure the validity of the identified issues.

We found that past mistakes in BT construction involved mistyped skill names and parameter names (cf. Listing 5), especially in parameter remapping. We additionally identified the concern that the pre-conditions and post-conditions of skills might be mismatched, which we explore in Section 6.1.

As shown in our previous case study, we used **Domain Language Mapping** to identify mistyped names in skill implementations early. Since the parameters to each skill implementation are defined in the skill description that is being implemented, this mapping required us to link each implementation to its corresponding description. Existing SkiROS2 code relied on calls to a `setDescription()` method to dynamically establish this relationship, as shown in line 2 of Listing 5. In practice, each skill implementation has exactly one skill description that it implements, meaning that there is no need to dynamically set this property. Instead, this relationship is closely related to the concepts of subtyping and interface implementation. We thus applied **Domain Language Mapping** to use Python's syntax for inheritance as a device for specifying the link from skill implementation to skill description (as shown in Listing 8; line 1). This approach both shortened the specification and allowed us to reliably identify the parameters and conditions (pre-conditions, post-conditions, etc.,) for each skill implementation.

Recall from the discussion shown in Section 4.1.2 how behavior trees are constructed in the *Skill Implementation* phase. Behavior trees were specified in the *expand* method where a list of skills is passed to a `skill()` wrapper after initializing a processor (lines 7–24). Each skill is defined with `self.skill(skilltype, label = " ", remap = .., specify = ..)`, allowing for parameter remapping. While composing skills in a behavior tree, the skills, their implementations, and the parameter remappings were passed as string parameters. For example, the BT specification for a "pick" skill in Listing 5 consists of a skill `ApproachMovement.go_to_linear` as `self.skill ("ApproachMovement", "go_to_linear", remap = 'Target':'Grasp-Pose')`. There are two problems with this notation that could lead to a runtime error: 1) if we pass a string that does not match any available skill descriptions or its implementations, and 2) incorrect remapping, such as referencing non-existent parameters, can lead to errors. Remapping is critical as it redirects parameters from one skill to another, ensuring proper data flow.

To prevent unexpected behavior at runtime, it is vital to detect and report such errors early. To address these issues, we use **Domain Language Mapping** to expose skill descriptions, implementations, and their parameters as Python objects and passed as identifiers (as shown in line 10 of Listing 8) that allow us to use Python's name analysis to locate skills with correct parameters (to remap to) and to find typos in those identifiers. Listing 8 shows how "pick" skill parameters are passed to the `expandBT` method and accessed directly as `params.ApproachPose`. This approach simplifies parameter remapping, ensuring accuracy and cohesiveness in skill execution.

```
<owl:ObjectProperty rdf:about="http://rvmi.aau.dk/ontologies/skiros.#hasA">
   <rdfs:subPropertyOf rdf:resource="http://rvmi.aau.dk/ontologies/skiros.#spatiallyRelated"/>
   <rdfs:range rdf:resource="#TransformationPose"/>
   <rdfs:domain rdf:resource="#Product"/>
</owl:ObjectProperty>
```

Listing 7. The definition of the object property "has A" in the SkiROS2 ontology.

## 6.1 Need for static pre-/post-condition matching in SkiROS2

As mentioned in Sections 4, 5, pre- and post-conditions in a BT implementation of a compound skill ensure the correct execution of skills to complete a robot's task. These conditions are checked in SkiROS2 by the *Skill Manager* before starting and parameterizing the skill. Although these conditions might seem less critical in controlled or smaller settings, their importance escalates as the complexity and scale of tasks grow. Poor quality or incorrectly defined conditions can significantly limit the ability of SkiROS2 to scale and handle complex, dynamic tasks efficiently. If we do not use a planner, manually creating compound skills or adjusting existing compounds without thorough checks can lead to mismatches between expected and actual skill behaviors. Static checking of pre-/post-conditions becomes essential to identify and correct these errors early in the development cycle, preventing potential failures during execution. To verify this requirement, we randomly selected five SkiROS2 skills written by developers to understand the prevalence of errors. Among those five skills, four of them failed the following basic checks:

- For skills in a *serial* or *serialstar* processor `s = serial (A, B, C)`, the pre-condition of "s" must entail the pre-condition of "A," and the aggregate post-conditions of "A" must entail the pre-condition of "B" and so on.
- For skills in a *selector* or *selectorstar* processor `s = selector (A, B, C)`, the pre-condition of "s" must entail the conjunction of the pre-conditions of "A," "B," and "C." Post-conditions of "s" can be conservatively checked as any of the children can lead to success without a predetermined order.
- For parallel skills, all children must succeed, with specific differences in handling the completion and order. This requires that no post-condition of one skill may invalidate the pre-condition of another due to the simultaneous nature of execution.

This evidence points to a common oversight in defining these conditions carefully and makes it important to have robust tooling to ensure that pre- and post-conditions are correctly matched and implemented. To address these challenges, we plan to create a comprehensive mapping and verification system in the future. This system would track all pre- and post-conditions, manage dependencies and changes, handle remapping accurately, and ensure that all conditions are consistent and verifiable at each step of the skill execution. It would be beneficial to use the design pattern **Source Provenance Tracking** to blame the exact skill whose post-condition did not match the expected state, which will make the debugging of behavior trees easier than before. It would likely involve a combination of static analysis tools, careful structuring of skill descriptions, and possibly enhancements to the SkiROS2 framework to support more robust condition checking and error reporting.

## 7 Overall evaluation of the extended EzSkiROS

Our evaluation of the extension of EzSkiROS (as mentioned in case study II) is primarily based on an in-depth review provided by an experienced SkiROS2 developer and maintainer who has used the tool for transforming the old SkiROS2 code into EzSkiROS. We requested developer feedback on various aspects of EzSkiROS, including its strengths and weaknesses, the impact on code readability and writability, the ease of code translation, the comprehensibility of errors encountered, and any general observations or suggestions they may have. The user's experience offers valuable insights into the strengths, weaknesses, and overall impact of EzSkiROS on skill description development in robotics.

Strengths and Weaknesses: The developer highlighted several key strengths of EzSkiROS.

- Early detection of misuse: EzSkiROS enables the detection of misuse in the world model before the skills are utilized, enhancing the correctness of the code.
- Validation of naming in conditions: The tool validates naming in pre-conditions and post-conditions, ensuring consistency and correctness in element types and names.
- Improved error messaging: Compared to traditional SkiROS2, EzSkiROS provides clearer and more concise error messages.
- Readability: There is a significant improvement in the readability of skill descriptions and skill implementations of both compound and primitive skills.

```
class pick(Pick.SkillBase):

    def expandBT(self, params):
        self.serialstar(
            SwitchController(Controller='joint_config'),
            MoveitCartesianSpaceMotion(GoalPose=params.ApproachPose),
            WmSetRelation.wm_set_relation(Src=params.Gripper, Dst=params.ApproachPose, Relation='skiros.at', RelationState=True),
                           HSVDetection(),
            SwitchController(Controller='compliant'),
            ApproachMovement.go_to_linear(Target=params.GraspPose),
            WmSetRelation.wm_set_relation(Src=params.Gripper, Dst=params.GraspPose, Relation='skiros.at', RelationState=True),
                           Wait(Duration=2.0),
            ActuateGripper(Open=False),
            WmMoveObject(TargetLocation=params.Gripper),
            ApproachMovement.go_to_linear(Target=params.ApproachPose),
            Wait(Duration=2.0)
        )
```

Listing 8. The EzSkiROS representation of the skill implementation is shown in Listing 5. Here, the inheritance from `Pick.SkillBase`links the pick skill description shown in Listing 6to its implementation.

However, the developer also noted a primary weakness.

- Developer productivity: Despite the aforementioned strengths, the developer expects that EzSkiROS will not provide substantial productivity benefits. The developer attributes this to the dynamic nature of most checks and the fact that world model errors abort Python execution, leading to one error being reported at a time.

Impact on Code Quality: The developer review suggests that EzSkiROS positively impacts the code quality in several ways:

- Correctness: By enforcing element types on parameters and consistent naming, the correctness of the code is improved.
- Readability and intuitiveness: The conciseness and clarity in pre- and post-conditions make the code easier to read and understand.
- Clarity in skill dependencies: The dependencies between Skill Description and SkillBase (Skill Implementation) of a skill are more apparent in the code.
- Conciseness in writing behavior trees: Writing behavior trees for compound skills have become more concise and less cluttered.

Translation Process: The developer reported that the translation of existing skill descriptions to EzSkiROS to be straightforward. The time required for translation depends on the number of skill descriptions to be converted, but it can be automated.

Error Reporting and Understanding: The user affirmed that the errors identified by EzSkiROS were sensible and contributed to a better understanding of the issues in the skill descriptions.

General Feedback: The developer acknowledged EzSkiROS as a significant step forward, particularly in moving from string-based descriptions to more natural and correct Python code. The reduction in common errors due to the validation of parameter names and world element relations was especially noted. For future work, the developer suggested the following:

- Static analysis integration: Implementing static analysis to run checks on modules and skills independently, possibly integrated with a linter, to further reduce bugs at an early stage.
- Code generation for enhanced development experience: Utilizing code generation to enable features like autocompletion and static checks during coding, particularly for the world model, to improve the development experience.

The user review provides an insightful evaluation of EzSkiROS, highlighting its strengths in improving code readability, correctness, and error messaging. The contribution of EzSkiROS to reducing common errors and improving the overall quality of skill descriptions is evident. According to the reviewer, it falls short in significantly enhancing developer productivity due to the fact that we do dynamic checks at pre-launch and the user suggests static analysis. It is important to note here that static check requires certain information (ontology and robot configuration) to be available at development time, which is not guaranteed. Modulo this caveat, we see no fundamental barrier toward using the techniques that we describe here for both pre-launch and static checks in practice, using language server or development environment plugins.

## 8 Threats to validity

### 8.1 Internal validity

EzSkiROS was evaluated on the skills implemented by Ph.D. students using SkiROS2 for research purposes. Consequently, there may be undetected errors or issues in other skills that utilize different or more extensive features of SkiROS2. Furthermore, the user study included only a small number of participants, which may not provide a comprehensive representation of all potential SkiROS2 users. This limitation could affect the reliability and generalizability of user feedback and reviews. For the initial in-depth review of EzSkiROS, only one experienced SkiROS2 developer was interviewed, and we have not yet evaluated it with more users of SkiROS2.

### 8.2 External validity

Although we expect that our design patterns can aid other Python-based robotic software, we have not validated this. Moreover, we have only validated these patterns for Python; it is an open question whether they would be effective in other languages such as Ruby or LISP.

## 9 Conclusion

In this paper, we present two analyses of different abstraction levels of robotic software and how we can use DSL design patterns to detect bugs at a pre-launch stage before runtime. Case study I demonstrated the value of our design patterns by showing how they help detect bugs in the high-level contracts between a variety of robot capabilities and the robot's world model. Case study II expands EzSkiROS by adapting the same techniques to detecting bugs in lower-level implementation code; in our case that implementation uses a behavior tree to integrate different robot capabilities.

In exploring the relationship between the two analyses, it is important to ask the following: do they work separately, depend on each other, or are they independent yet work better together, creating a stronger combined effect than each would alone? The study shows that analysis of behavior trees (case study II) requires information about the skill parameters from the higher-level descriptions to check correct information being passed on between skills. Behavior trees also need to access the pre-, post-, and hold-conditions from the skill descriptions of the skill being implemented. On the other hand, the higher-level analysis (case study I) is stand-alone but can benefit from the BT sequencing information to suggest pre- and post-conditions to the developer. Our work demonstrates how embedded DSLs can help robotics developers detect bugs early, even when the analysis depends on data which are not available until run-time. Our evaluation with EzSkiROS further suggests that embedded DSLs can achieve this goal while simultaneously increasing code maintainability.

In our future work, we plan to collect some objective results to further substantiate our efforts. We plan to make EzSkiROS publicly available to SkiROS2 users so that people can write skills and transform their old skills into EzSkiROS, and we can get some error reports and if people find the error reports helpful. We aim to conduct an in-depth user study to explore how EzSkiROS assist users in writing skill descriptions and detecting bugs in behavior trees through pre- and post-condition matching. This study will mainly focus on understanding the user experience with EzSkiROS, particularly in terms of its usability and effectiveness in early bug detection. A significant aspect of this study will be to extend the possibility of the integration of the two analyses at different abstraction levels and see how their combination influences the bug detection process. We are particularly interested in whether this integration simplifies the process of writing error-free skill descriptions and how it impacts the overall development workflow. By analyzing the data collected from this study, we expect to gain valuable insights into the practical applications and limitations of EzSkiROS. This will not only help us in refining the tool but also contribute to the broader understanding of skill programming in robotics.

## Data availability statement

The original contributions presented in the study are included in the article/Supplementary Material; further inquiries can be directed to the corresponding authors.

## Author contributions

MR: conceptualization, data curation, formal analysis, investigation, methodology, project administration, resources, software, validation, visualization, writing–original draft, and writing–review and editing. CR: conceptualization, formal analysis, funding acquisition, methodology, resources, software, supervision, writing–original draft, and writing–review and editing. RC: conceptualization, data curation, investigation, visualization, writing–original draft, and writing–review and editing. MM: resources, visualization, writing–original draft, and writing–review and editing. VK: funding acquisition, supervision, writing–original draft, and writing–review and editing.

## Funding

The author(s) declare that financial support was received for the research, authorship, and/or publication of this article. This work was partially funded by the Wallenberg AI, Autonomous Systems, and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation.

## Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## Publisher's note

## References

Ahmad, F., Mayr, M., and Krueger, V. (2023). "Learning to adapt the parameters of behavior trees and motion generators (btmgs) to task variations," in *2023 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, 10133–10140. doi:10.1109/IROS55552.2023.10341636

Ahmed, A., Findler, R. B., Siek, J. G., and Wadler, P. (2011). "Blame for all," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 201–214.

Balldin, H., and Reichenbach, C. (2020). "A domain-specific language for filtering in application-level gateways," in *GPCE 2020*, 111–123.

Bøgh, S., Nielsen, O. S., Pedersen, M. R., Krüger, V., and Madsen, O. (2012). "Does your robot have skills?," in *Proceedings of the 43rd international symposium on robotics* (Denmark: VDE Verlag GMBH).

Brugali, D., Agah, A., MacDonald, B., Nesnas, I. A., and Smart, W. D. (2007). "Trends in robot software domain engineering," in *Software engineering for experimental robotics* (Springer), 3–8.

Buch, J. P., Laursen, J. S., Sørensen, L. C., Ellekilde, L.-P., Kraft, D., Schultz, U. P., et al. (2014). "Applying simulation and a domain-specific language for an adaptive action library," in *Simulation, modeling, and programming for autonomous robots*, 86–97. doi:10.1007/978-3-319-11900-7_8

Cangelosi, A., and Asada, M. (2022). *Cognitive robotics*. MIT Press.

Ceh, I., Crepinšek, M., Kosar, T., and Mernik, M. (2011). Ontology driven development of domain-specific languages. *Comput. Sci. Inf. Syst.* 8, 317–342. doi:10.2298/csis101231019c

Colledanchise, M., and Ögren, P. (2018). *Behavior trees in robotics and AI: an introduction*. CRC Press. 1st edn. doi:10.1201/9780429489105

Coste-Maniere, E., and Turro, N. (1997). "The maestro language and its environment: specification, validation and control of robotic missions," in RSJ International Conf. on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS (IEEE), 836–841. doi:10.1109/iros.1997.655107

Dortmans, E., and Punter, T. (2022). Behavior trees for smart robots practical guidelines for robot software development. *J. Robotics* 2022, 1–9. doi:10.1155/2022/3314084

Dragule, S., Gonzalo, S. G., Berger, T., and Pelliccione, P. (2021). "Languages for specifying missions of robotic applications," in *Software engineering for robotics* (Springer), 377–411.

Faconti, D. (2019). *Mood2be: models and tools to design robotic behaviors*. Barcelona, Spain: Eurecat Centre Tecnologic. Tech. Rep 4.

Ghzouli, R., Berger, T., Johnsen, E. B., Dragule, S., and Wasowski, A. (2020). "Behavior trees in action: a study of robotics applications," in *Proceedings of the 13th ACM SIGPLAN international conference on software language engineering*, 196–209.

Ghzouli, R., Berger, T., Johnsen, E. B., Wasowski, A., and Dragule, S. (2023). Behavior trees and state machines in robotics applications. *IEEE Trans. Softw. Eng.* 49, 4243–4267. doi:10.1109/tse.2023.3269081

Henn, T., Völker, M., Kowalewski, S., Trinh, M., Petrovic, O., and Brecher, C. (2022). "Verification of behavior trees using linear constrained horn clauses," in *International conference on formal methods for industrial critical systems* (Springer), 211–225.

Hitzler, P., Krötzsch, M., and Rudolph, S. (2009). *Foundations of semantic Web technologies*. Boca Raton, FL: Taylor & Francis.

Iovino, M., Scukins, E., Styrud, J., Ögren, P., and Smith, C. (2022). A survey of behavior trees in robotics and ai. *Robotics Aut. Syst.* 154, 104096. doi:10.1016/j.robot.2022.104096

King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM* 19, 385–394. doi:10.1145/360248.360252

Krueger, V., Chazoule, A., Crosby, M., Lasnier, A., Pedersen, M. R., Rovida, F., et al. (2016). A vertical and cyberA vertical and cyber–physical integration of cognitive robots in manufacturingphysical integration of cognitive robots in manufacturing. *Proc. IEEE* 104, 1114–1127. doi:10.1109/jproc.2016.2521731

Kunze, L., Roehm, T., and Beetz, M. (2011). "Towards semantic robot description languages," in 2011 IEEE International Conference on Robotics and Automation (IEEE). doi:10.1109/icra.2011.5980170

Lamy, J.-B. (2017). Owlready: ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. *Artif. Intell. Med.* 80, 11–28. doi:10.1016/j.artmed.2017.07.002

Levine, J. R., Mason, T., and Brown, D. (1992). *Lex and yacc*. Sebastopol, CA: O'Reilly Media, Inc.

Mayr, M., Ahmad, F., Chatzilygeroudis, K., Nardi, L., and Krueger, V. (2022a). Combining planning, reasoning and reinforcement learning to solve industrial robot tasks. *arXiv Prepr. arXiv:2212.03570*.

Mayr, M., Ahmad, F., Chatzilygeroudis, K., Nardi, L., and Krueger, V. (2022b). "Skill-based multi-objective reinforcement learning of industrial robot tasks with planning and knowledge integration," in *2022 IEEE international conference on robotics and biomimetics (ROBIO)*.

Mayr, M., Ahmad, F., Duerr, A., and Krueger, V. (2023a). "Using knowledge representation and task planning for robot-agnostic skills on the example of contact-rich wiping tasks," in *2023 IEEE 19th international Conference on automation Science and engineering (CASE)* (IEEE), 1–7.

Mayr, M., Hvarfner, C., Chatzilygeroudis, K., Nardi, L., and Krueger, V. (2022c). "Learning skill-based industrial robot tasks with user priors," in *2022 IEEE 18th international conference on automation science and engineering (CASE)*, 1485–1492. doi:10.1109/CASE49997.2022.9926713

Mayr, M., Rovida, F., and Krueger, V. (2023b). "Skiros2: a skill-based robot control platform for ros," in 2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE), 6273–6280.

Moghadam, M., Christensen, D. J., Brandt, D., and Schultz, U. P. (2013). Towards python-based domain-specific languages for self-reconfigurable modular robotics research. *arXiv Prepr. arXiv:1302.5521*.

Nordmann, A., Hochgeschwender, N., Wigand, D. L., and Wrede, S. (2016). A survey on domain-specific modeling and languages in robotics. *J. Softw. Eng. Robotics (JOSER)* 7, 75–99. doi:10.1007/978-3-319-11900-7_17

Oliveira, M., Silva, P. M., Moura, P., Almeida, J. J., and Henriques, P. R. (2020). *Bhtsl, behavior trees specification and processing*.

Pedersen, M. R., Nalpantidis, L., Andersen, R. S., Schou, C., Bã¸gh, S., KrÃ¼ger, V., et al. (2016). Robot skills for manufacturing: from concept to industrial deployment. *Robotics Computer-Integrated Manuf.* 37, 282–291. doi:10.1016/j.rcim.2015.04.002

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., et al. (2009). "Ros: an open-source robot operating system," in *ICRA workshop on open source software* (Kobe, Japan), 5.

Reichenbach, C. (2021). "Software ticks need no specifications," in *ICSE-NIER 2021* (IEEE), 61–65.

Rizwan, M., Caldas, R., Reichenbach, C., and Mayr, M. (2023). "Ezskiros: a case study on embedded robotics dsls to catch bugs early," in *2023 IEEE/ACM 5th international workshop on robotics software engineering (RoSE)* (IEEE), 61–68.

Rovida, F., Crosby, M., Holz, D., Polydoros, A. S., Großmann, B., Petrick, R. P., et al. (2017a). "SkiROS— a skill-based robot control platform on top of ROS," in *Robot operating system (ROS)* (Springer), 121–160.

Rovida, F., Grossmann, B., and Krueger, V. (2017b). "Extended behavior trees for quick definition of flexible robotic tasks," in *RSJ international conf. On intelligent robots and systems (IROS)* (IEEE), 6793–6800.

Spinellis, D. (2001). Notable design patterns for domain-specific languages. *J. Syst. Softw.* 56, 91–99. doi:10.1016/s0164-1212(00)00089-3

Tadiello, M., and Troubitsyna, E. (2022). *Verifying safety of behaviour trees in event-b. arXiv preprint arXiv:2209.14045*.

Wuthier, D., Rovida, F., Fumagalli, M., and Krueger, V. (2021). "Productive multitasking for industrial robots," in *2021 IEEE international conference on robotics and automation (ICRA)*, 12654–12661. doi:10.1109/ICRA48506.2021.9561266

TYPE Original Research
PUBLISHED 29 January 2025
DOI 10.3389/frobt.2024.1363150

Check for updates

OPEN ACCESS

EDITED BY
Ivano Malavolta,
VU Amsterdam, Netherlands

REVIEWED BY
Ricardo Sanz,
Polytechnic University of Madrid, Spain
Christian Eymüller,
University of Augsburg, Germany

*CORRESPONDENCE
Herman Bruyninckx,
herman.bruyninckx@kuleuven.be

RECEIVED 29 December 2023
ACCEPTED 27 November 2024
PUBLISHED 29 January 2025

CITATION
Schneider S, Hochgeschwender N and Bruyninckx H (2025) Semantic composition of robotic solver algorithms on graph structures. *Front. Robot. AI* 11:1363150. doi: 10.3389/frobt.2024.1363150

# Semantic composition of robotic solver algorithms on graph structures

Sven Schneider[1,2], Nico Hochgeschwender[3] and Herman Bruyninckx[2,4,5]*

[1]Department of Computer Science, Institute for AI and Autonomous Systems, Hochschule Bonn-Rhein-Sieg, Sankt Augustin, Germany, [2]Department of Mechanical Engineering, KU Leuven, Leuven, Belgium, [3]Department of Mathematics and Computer Science, University of Bremen, Bremen, Germany, [4]Department of Mechanical Engineering, TU/e Eindhoven, Eindhoven, Netherlands, [5]Flanders Make, Lommel, Belgium

This article introduces a model-based design, implementation, deployment, and execution methodology, with tools supporting the systematic composition of algorithms from generic and domain-specific computational building blocks that prevent code duplication and enable robots to adapt their software themselves. The envisaged algorithms are numerical solvers based on graph structures. In this article, we focus on kinematics and dynamics algorithms, but examples such as message passing on probabilistic networks and factor graphs or cascade control diagrams fall under the same pattern. The tools rely on mature standards from the Semantic Web. They first synthesize algorithms symbolically, from which they then generate efficient code. The use case is an overactuated mobile robot with two redundant arms.

KEYWORDS

**solvers based on graph traversal, model-based engineering, algorithm synthesis, code generation, composability and compositionality, kinematics and dynamics**

## 1 Introduction

Figure 1 shows a complicated, overactuated mobile robot with two redundant, torque-controlled arms performing a dual-arm manipulation task. A typical implementation of such an application relies on a wide range of algorithms, including (i) kinematics and dynamics solvers for forward kinematics or inverse dynamics problems as available in libraries like Pinocchio (Carpentier et al., 2019), the Rigid Body Dynamics Library (RBDL) (Felis, 2016), or the Kinematics and Dynamics Library (KDL)[1]; (ii) probabilistic filters and estimators, implemented by libraries such as the Georgia Tech Smoothing and Mapping library (GTSAM) (Dellaert, 2012), or the Bayesian Filtering Library (BFL)[2], to determine the state of the robot and its environment, for example, by simultaneous

1 http://www.orocos.org/kdl
2 http://www.orocos.org/bfl

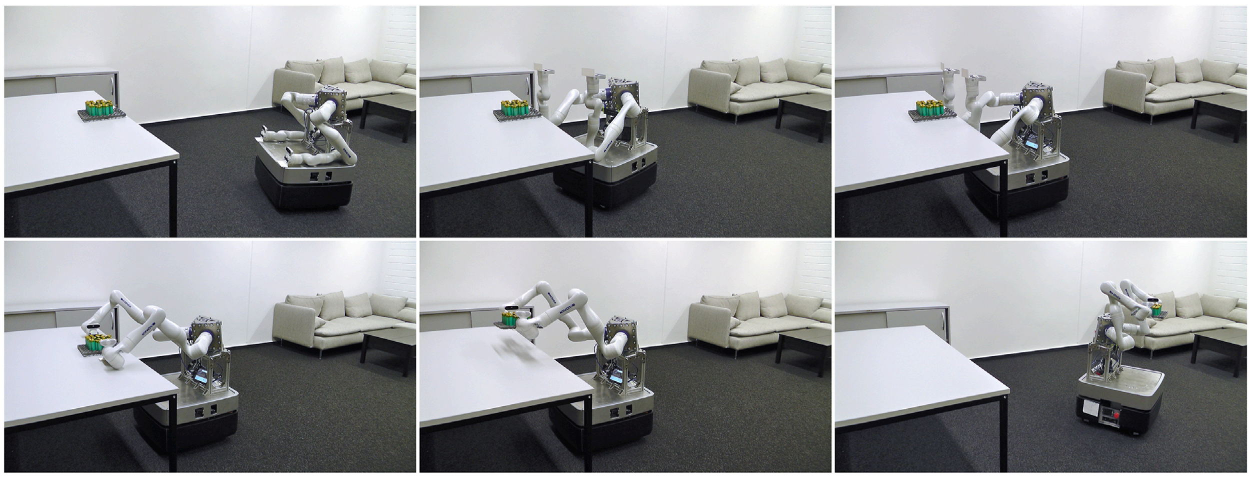


FIGURE 1
Complicated robotic system solving a dual-arm manipulation task. In the first row, the robot approaches the table with its mobile base to then perform a touch-based alignment. In the second row, the two manipulators grasp and lift the object. Finally, the robot leaves the table while carrying the object.

localization and mapping (SLAM); (iii) data-flow computations in cascade control diagrams such as the MATLAB Control System Toolbox[3], in the Stack-of-Task's (Mansard et al., 2009) dynamic-graph[4], or in video-processing pipelines like GStreamer[5]; and (iv) task specifications, expressing the *desired* behavior of the robot's dynamics and its controllers as well as the *desired* sensor processing outputs, realized via expression graphs (Aertbeliën and De Schutter, 2014). The overall integration of such functionalities and their realization that the robot requires to solve its tasks is also known as a robot control architecture.

Even if these algorithms and libraries originate from different yet highly relevant robotics domains, they share two important commonalities. First, they rely on an underlying structural model of a *graph* that represents a kinematic chain, a probabilistic network or factor graph, a data-flow network between operators, and an expression graph, respectively. Second, they answer *queries* on these graphs by (i) *propagating* data between the graph's nodes and (ii) *dispatching*[6] computations on that data for each visited node or edge while (iii) performing one or more graph *traversals*. Here, a traversal represents a particular choice of *serializing* or *scheduling* the computations to establish a computational control flow. The good news is that for many of these queries, the knowledge already exists about how to create efficient execution schedules. This includes kinematics and dynamics problems (Popov et al., 1978), inference in Bayesian networks (Pearl, 1982), or cutting cascade control loops into a series of computations for each time scale. The main differences between these solvers comprise the data encoded in the graph and the specific policies or choices imposed on the algorithms, that is, which data to propagate, which computations to perform, and how to traverse the graph. The solution to such a recurring problem in architectures is known as a *design pattern* and has been popularized in software engineering by the "Gang of Four" (Gamma et al., 1994) in object-oriented software development. Hence, given these commonalities and differences, we classify such *graph-based solvers* as a fundamental pattern that has not yet been described in the existing literature.

On the one hand, software libraries that implement graph-based solvers allow their users to customize the structural graphs at compile time. On the other hand, they keep the solver algorithms that act on these graphs inaccessible, which leads to the following three problems. First, such designs prevent many customizations and optimizations of the computational control flow as well as the introspection and instrumentation of the executing algorithms. The easy way to introduce a new algorithm or adapt an existing one is for developers to implement it completely from scratch or by copying and refactoring a previous implementation. For instance, in KDL, we have counted twelve realizations of the computations for the forward position kinematics (FPK) and seven realizations for the forward velocity kinematics (FVK) across 22 solvers in total. This is a clear violation of the "Don't Repeat Yourself" (DRY) (Hunt and Thomas, 2019) principle for good software engineering and leads to technical debt. A second problem comes from how the libraries support *configuration*. One approach is to create an application programming interface (API) where the configuration options are part of the input parameters. This leads to very long function signatures, so the pragmatic choice is to limit the configuration capabilities of the library. Another (better) approach to configuration is to provide a *setters and getters* API via which any desired set of parameters can be given new values at runtime. However, this introduces the risk of data inconsistencies because, in most cases, several parameters should be updated *together* in an *atomic* way. A third problem is that, at runtime, applications may require multiple solvers with partially overlapping "computational states," such as the position and motion of sensors

3 https://www.mathworks.com/products/control.html
4 https://github.com/stack-of-tasks/dynamic-graph
5 https://gstreamer.freedesktop.org/
6 By "dispatching," we mean to execute or trigger a computation. It resembles dispatchers in operating systems [cf. (Tanenbaum and Bos, 2014)].

or tools on the robot's kinematic chain. This leads to redundant computations and challenges in keeping a consistent state between these multiple solvers.

To address these problems, this article introduces the *software engineering* aspects of a model-based design, implementation, deployment, and execution methodology, with tools supporting the systematic composition of algorithms from generic and domain-specific computational building blocks. A first contribution is that the granularity of these building blocks is designed for **composability**: on the one hand, they are so small that each of them is a pure function and, on the other hand, they need not be smaller than what is contained in one node of the graph that represents the computational control flow. That is, where the `if-then-elses` and the `for` or `while` loops are introduced to differentiate between different behaviors of the executed algorithms. The algorithm's building blocks are models of the *data structures*, the pure *functions* that act on this data, and the order in which these functions are called, that is, the *schedule* or *control flow*. We present the *mechanism* to model, compose, and execute complicated algorithms. Simultaneously, we ensure that each mechanism is *configurable* so that a large variety of data-flow *policies* can be composed on top. Examples include, among others, incremental computations by processing sub-graphs on demand, employing optimized computations for sub-graphs, or injecting instrumentation and logging into the algorithm. In summary, our main contributions are:

- We analyze kinematics and dynamics solvers as the main representatives of algorithms that perform computations on graph structures. Their commonalities and differences allow us to identify and describe the graph-based solver pattern.
- We derive free and open-source licensed, vendor-neutral models and metamodels[7] to represent and compose graph-based solvers for kinematics and dynamics solvers. The models include the data structures, the operators or functions that manipulate these data structures, and the ordering constraint on the functions. We reify each of these elements so that they can be symbolically referenced.
- We develop a toolchain that processes the above graph-structured models using symbolic queries to synthesize kinematics and dynamics solvers and generate code from the resulting models. We complement the toolchain by an implementation of a software library that implements the pure functions to implement the solvers. Both are released under a free and open-source license.
- We showcase the models, toolchain, and backend software library in a case study for kinematics and dynamics solvers.

The remainder of this article is structured as follows. In Section 2, we revisit the application to provide a detailed review of kinematics and dynamics solvers to then derive requirements for graph-based solvers in Section 3. Afterward, Section 4 provides the background on composable models. We present the tooling for solver synthesis and code generation in Section 5, followed by a case study in Section 6. Section 7 discusses our approach and tools, while Section 8 concludes the article.

## 2 Kinematics and dynamics solvers

This section describes the structural and computational policies used in numerical solvers for kinematic chains via (i) the topology of the underlying graphs; (ii) the types of traversals (that is, the serialization of the computations) over these graphs; (iii) the representation of data structures; (iv) the types of computations on these data structures; (v) the handling of cycles in the graphs; (vi) the handling of domain-specific, composite and hierarchical nodes; and (vii) the support of incremental computations to only evaluate output that depends on changed input and caching of intermediate results. In the supplementary material, we provide an additional analysis of graph-based solvers for probabilistic networks, data-flow programming, and expression graphs.

The robot in Figure 1 exemplifies the most relevant types of kinematic chains: each of the two manipulators by itself is a *serial chain*, but when connecting both arms to the robot, a *tree-structured chain* with the torso as its root emerges. Finally, the mobile base is an example of a *parallel chain* where the ground couples (or "constrains") the motion of all wheels. The objective of solvers that act on such kinematic chains is to answer queries that compute the instantaneous forces and motions of all links when a particular subset of them is given as inputs (or "drivers") for the motion. Figure 2A depicts an example where multiple motion drivers are attached to a kinematic chain to move or accelerate a body in a certain direction while resisting external forces. Additionally, the application specifies the expected solver outputs, such as the pose (position and orientation) and velocity of an end-effector, or the joint-level control torques to achieve the desired motions.

The following paragraphs provide some concrete examples of queries and their solvers. Algorithm 1 shows an FPK solver that, given a model of a kinematic chain with $N$ bodies and the joint positions $\boldsymbol{q}$ as inputs, computes the pose ${}^{i}\boldsymbol{X}_0$ of each body $i$ with respect to the root body 0. To this end, in Line 2, it composes the static pose over the body (or "link") $\boldsymbol{X}_{L,i}$ with the pose over the joint $\boldsymbol{X}_{J,i}(q_i)$ that depends on the current joint position. The result is the relative pose of the current body $i$ with respect to its parent $p(i)$. Then, in Line 4, the solver accumulates the parent's pose with that relative pose. Here, a single outward traversal (Line 1) of the kinematic chain from a selected root to the leaves suffices to compute the answer. In the context of kinematic chains, such a graph traversal that serializes kinematic or dynamic computations is also called a *sweep*.

The FVK solver in Algorithm 2 computes the Cartesian velocity $\dot{\boldsymbol{X}}_i$ for each body in the kinematic chain given the joint velocities $\dot{\boldsymbol{q}}$. A comparison of the FVK solver with the FPK solver reveals that the former is an extension of the latter: only two lines have been added, while the others remain the same. These two lines are the initialization of the root body's velocity (Line 1) and the accumulation of velocities along the kinematic chain (Line 6). The accumulation step consists of, from right to left, (i) mapping the joint velocity $\dot{q}_i$ to Cartesian space with the joint Jacobian $\boldsymbol{S}_i$; (ii)

7 A metamodel is a model that represents the constraints that a concrete model must satisfy, structurally and semantically, in order to be a "well-formed model" in the context of the application domain the model is designed for.

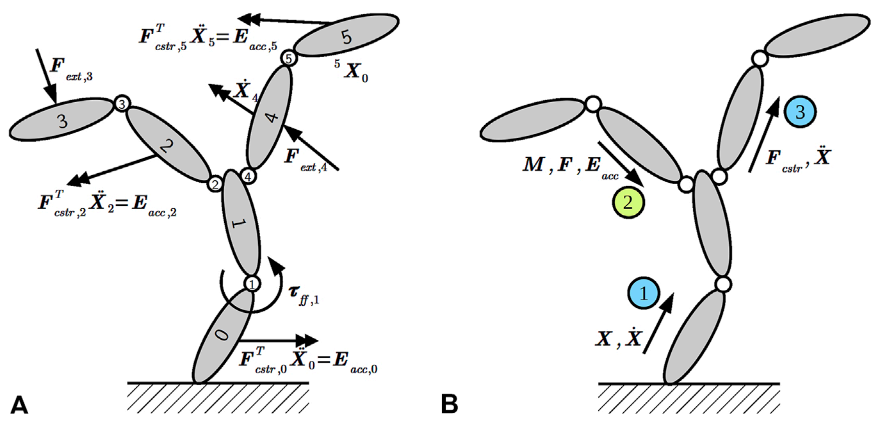

$F^T_{cstr,5}\ddot{X}_5 = E_{acc,5}$
$^5X_0$
$F_{ext,3}$
$\dot{X}_4$
$F_{ext,4}$
$F^T_{cstr,2}\ddot{X}_2 = E_{acc,2}$
$\tau_{ff,1}$
$F^T_{cstr,0}\ddot{X}_0 = E_{acc,0}$
A
$M, F, E_{acc}$
$F_{cstr}, \ddot{X}$
$X, \dot{X}$
B

FIGURE 2
**(A)** Kinematic chain with three types of motion drivers attached to links and joints as solver input: desired Cartesian accelerations via constraint forces $F_{cstr}$ and acceleration energy $E_{acc}$; Cartesian external forces $F_{ext}$; and joint torques $\tau_{ff}$. The solver's output is the answer to the query that asks for the fifth link's pose $^5X_0$ and the fourth link's velocity $\dot{X}_4$. **(B)** Kinematic chain consisting of links and joint with positions $X$ and velocities $\dot{X}$ propagated outward (blue circle) during the first sweep; inertia $M$, force $F$, and acceleration energy $E_{acc}$ propagated inward (green circle) during the second sweep; as well as constraint forces $F_{cstr}$ and acceleration $\ddot{X}$ propagated outward during the third sweep.

```
Input  : Chain model, $q$
Output : $^1X_0 \dots {}^NX_0$
1 sweep $i \leftarrow 1..N$:
2     $^iX_{p(i)} \leftarrow X_{L,i}X_{J,i}(q_i)$
3     if $i \neq 1$ then
4         $^iX_0 \leftarrow {}^iX_{p(i)}{}^{p(i)}X_0$
```

**Algorithm 1. Forward position kinematics.**

```
Input  : Chain model, $q$, $\dot{q}$
Output : $^1X_0 \dots {}^NX_0$, $\dot{X}_0 \dots \dot{X}_N$
1 $\dot{X}_0 = 0$
2 sweep $i \leftarrow 1..N$:
3     $^iX_{p(i)} \leftarrow X_{L,i}X_{J,i}(q_i)$
4     if $i \neq 1$ then
5         $^iX_0 \leftarrow {}^iX_{p(i)}{}^{p(i)}X_0$
6     $\dot{X}_i \leftarrow \dot{X}_{p(i)} + {}^iX_0^{-1}S_i\dot{q}_i$
```

**Algorithm 2. Forward velocity kinematics.**

transforming that Cartesian velocity to the root coordinate frame using the inverse transformation matrix $^iX_0$; and (iii) adding the Cartesian velocity $\dot{X}_{p(i)}$ that has already been accumulated in the previous step, up to and including the parent body.

Another policy for the velocity accumulation step is to express the velocities in the moving coordinate frame instead of the stationary root frame: $\dot{X}_i \leftarrow {}^iX_{p(i)}\dot{X}_{p(i)} + S_i\dot{q}_i$. *Many* more such policies exist, especially when considering complicated algorithms, including forward and inverse dynamics solvers (cf. Featherstone, 2008; Vereshchagin, 1989) that map forces to accelerations and *vice versa*, respectively. The variety in solver policies is due to the large set of choices that are possible, for example, (i) the choice of physical units that must be kept consistent across all operations; (ii) the propagation of the motion drivers that could either be accumulated as soon as possible (for the most efficient computations) or only during the third, solver sweep (for most flexibility); or (iii) the choice of matrix inversion and the handling of singularities during such an inversion. The two solver examples already demonstrate how a naïve implementation of such algorithms leads to code duplication when each algorithm resides in its own function or class, as is commonly the case in software libraries. For example, a hypothetical solver library may provide the functions `fpk(chain, q)` for Algorithm 1, `fvk_stationary(chain, q, qd)` for Algorithm 2, and `fvk_moving(chain, q, qd)` for the choice of the moving coordinate frame, each containing computations of the FPK solver. The number of policies increases even further when the chain's dynamics also enter into the solver.

The two algorithms above demonstrate a computation *on* a graph. The graph represents the kinematic and dynamic properties of the kinematic chain (including the *topology* of the connections between links) but does not contain all data structures found in the algorithms. Instead, all variables apart from the already specified pose over the link, $X_{L,i}$, must be added to the graph. The computations are the various types of operators with *physical* meaning that are represented *mathematically* by either matrix multiplication or vector addition (composition of poses, maps from joint space to Cartesian space, transformation of a velocity, or addition of two velocities). The top-to-bottom order of the lines is a physically imposed ordering constraint: here, the transformation of velocities depends on (the presence of) poses. Finally, more complicated solvers rely on up to three sweeps, as depicted in Figure 2B: positions and velocities travel outward from the root to the leaves in the first sweep; inertia, force, and acceleration energy travel inward in the opposite direction during the second sweep; the third sweep is outward again, accumulating those computational results that are needed for the actual query.

Some kinematic chains have *cycles*. Solvers deal with such cycles in two complementary ways. The first option is to cut an edge in each cycle, which results in a spanning tree of the kinematic chain. For each cut, the solver adds Cartesian acceleration constraints on either side of the cut, representing the physical reality that both sides of the cut must move with the same acceleration. That reality is a *constraint* that the solver algorithm must take into account. The solver deals with the loop constraint by computing the *constraint forces* that would make both sides of the cut accelerate in exactly the same way. The second option is to cluster the cycles into composite nodes that must then be solved for numerically using an explicit matrix inversion (Jain, 2012; Chignoli et al., 2023).

We have already introduced a hierarchy in the description of the types of kinematic chains above. Here, the apex of that hierarchy could be the whole robot, the base with two arms, to be treated as one kinematic chain on which a single solver operates. However, more commonly, roboticists decouple the arms from the base and associate dedicated solvers with each; often, kinematics solvers that run at lower control frequencies suffice for the base, whereas for the arms dynamics solvers at higher control frequencies are required to handle contact situations. Finally, the individual joints of the arms are the "smallest" kinematic chains. Yet, even at this level, joints could actually be composites. As an example, one model of a spherical joint is a sequence of three revolute joints. Then, solvers dispatch specialized computations depending on the joint type. For example, one-dimensional joints such as revolute and prismatic joints allow for computationally efficient solutions that rely on scalars instead of full matrices.

We have presented an example of incremental computations in dynamics solvers in Schneider and Bruyninckx (2019): the propagation of the so-called articulated-body inertia matrix (see Featherstone, 2008) is a computationally expensive operation. Additionally, the inertia matrix does not change significantly in neighboring configurations, while its parameterization is prone to measurement noise. Consequently, it is a good candidate for a computation that is performed at a reduced frequency in comparison to the propagation of the other quantities. The articulated-body inertia matrix is then cached and reused across multiple solver invocations.

## 3 Requirements for graph-based solvers

Physical and scientific constraints exist that lead to efficient solvers for kinematic chains and graphical models. It is the top-level tree structure of the underlying graphs that enables the application of dynamic programming. For graphs with cycles, the graph must be pre-processed to establish a tree-structured *view* on the graph, either as a spanning tree or a hierarchical decomposition as in the junction tree algorithm. On the one hand, dynamic programming dictates which data structures should be cached at each node and which operations should be performed on that data. On the other hand, it coordinates or schedules the computations along the graph traversal. Two sweeps, one inward and one outward, decompose the graph's state that can then be flexibly and efficiently recomposed in a final solver sweep to answer queries. The scheduling can depend on various types of state or runtime conditions, such as the availability of data or conflicts in the motion specification. Hence, we can encode such a solver algorithm as a computational graph on top of the underlying structural graph. The latter is the basis of the former's *bookkeeping* (which data structures to use in which operations), *configuration* (which values to fill into the data structures), and *coordination* (in which order to execute the operations). For data-flow networks and expression graphs, these three points are completely at the developers' disposal, who must rely on their insights into the domain to design the algorithms. Nevertheless, the same algorithmic building blocks exist in these approaches.

We can derive various requirements for our approach from the analysis. First, we need explicit models of the graphs' structure and of how behavior is attached to that structure. Here, behavior refers to explicit models of algorithms that consist of data structures, functions, and schedules. The various computational policies, such as caching of intermediate results or varying execution frequencies are then a higher-order composition to the algorithms. A second necessity is flexible tooling that efficiently synthesizes the domain-specific algorithms and attaches them to the underlying graph models. Because the algorithms are merely models, additional tools are required that can execute these models by interpretation or compilation. Finally, all of the above models should be unambiguously understandable by a robot so that it can automatically adapt its software, also at runtime.

## 4 Composable and compositional models for kinematic chains

In this section, we summarize the main results of our prior work from Schneider et al. (2023) to represent the above-mentioned graphs (their structure and their "behavior") as they are a prerequisite for the remainder of this article. In that publication, we have presented an in-depth analysis of existing modeling formats, including the Unified Robot Description Format (URDF)[8] and the Semantic Robot Description Format (SRDF)[9] that originate from the Robot Operating System (ROS) ecosystem (Quigley et al., 2009). Given the lessons learned, we have designed and realized composable and compositional models in JSON-LD (Sporny et al., 2020). **Composability** pertains to *structure* and is an application of two major software design principles to models. The first is the *open-closed principle* (Meyer, 1997), which implies that it should always be possible to extend existing models without a need for modification. The second is the *single-responsibility principle* (Martin, 2003), which implies that each model should represent exactly one concern. In relational databases, the latter principle is known as the third normal form (Codd, 1971; Kent, 1983): each table has a single "topic" and only contains direct dependencies on the table's key; that is, it only represents intrinsic *properties* instead of extrinsic *attributes* (Bruyninckx, 2023, Section 1.5.3). **Compositionality** is

8 http://wiki.ros.org/urdf
9 http://wiki.ros.org/srdf

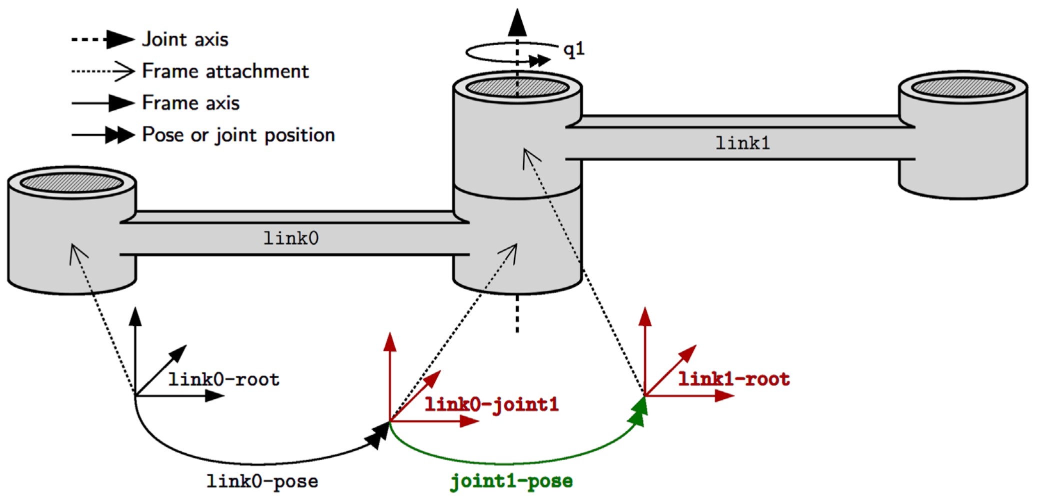


FIGURE 3
The joint `joint1` constrains the relative motion of the two frames `link0-joint1` and `link1-root` that are associated with the links `link0` and `link1`, respectively. For example, the pose `joint1-pose` of the latter frame with respect to the former frame is associated with the joint position `q1` and can change over time. Finally, pose `link0-pose` locates the joint frame on link `link0` with respect to the root frame `link0-root`. For rigid bodies, this pose remains static.

concerned with model *semantics*. It implies that each model must have an unambiguous meaning[10]. For composite models, that meaning must follow from the meaning of its constituents and the composition rules (which are *higher-order models*, that is, a set of relations with other models as arguments) so as to avoid unpredictable "emergent" behavior of the composed system. Both these design goals, composability and compositionality, are highly relevant in complex modern robots that act in open environments and open-ended missions where (i) designers usually cannot foresee all possible applications of their models and, hence, should avoid introducing artificial limitations; and (ii) robots must be able to interpret and reason about the models by themselves without having to rely on human developers to transform the models to code.

JSON-LD models are both JavaScript Object Notation (JSON) (Bray, 2017) documents and Resource Description Format (RDF) (Cyganiak et al., 2014) documents[11]. They support composability and compositionality because all model elements (i) have unique identifiers so that they can be referenced from "external" sources such as files on servers or even executing software binaries; (ii) can refer to complete metamodels that unambiguously define the models' semantics, so that they are free from implicit assumptions; and (iii) are loosely coupled due to the underlying, generic graph structure as well as the support for "symbolic pointers" that are represented by Internationalized Resource Identifiers (IRI) as defined by Duerst and Suignard (2005). In the following subsections, we introduce concrete JSON-LD models of kinematic chains and their behavior as a running example. The proper design of the underlying metamodels can only originate from a detailed and exhaustive domain analysis as we have performed for kinematic chains here and for the additional domains in the supplementary material. As a typographic convention, we indicate model elements in a `monospaced font`. In addition, we designate models by concise and human-understandable identifiers, yet their real meaning must only come from their properties and metamodels.

Figure 3 depicts two links that are constrained in their relative motion by a joint. We consider the most abstract representation of a link or body, its "skeleton," as simply a collection of `simplices`, that is, geometric entities such as points, lines, or frames. These simplices are attachment points for, among others, shape geometry, inertia, motion specifications and also joints as textually represented using JSON-LD in Listing 1. Syntactically, JSON-LD can add one identifier (`@id` keyword), one or more types (`@type` keyword), and one context (`@context` keyword) to any JSON object. The *referenced context*, as a list of IRIs, symbolically points to all *metamodels* that define the meaning of this model. Part of that metamodel is the structural constraint that the `Joint` type demands the `between-attachments` property, as indicated by the matching colors. One way to formally represent such a constraint is via the Shape Constraint Language (SHACL) defined in Knublauch and Kontokostas (2017). Another part of the metamodel defines that the `between-attachments` property symbolically refers to a list of all simplices that are, on the one hand, attached to bodies and, on the other hand, are involved in the joint-constraint relation. Similar to the body, this is the most abstract representation of a joint that captures nothing more than the joint's constituents. The type of joint (e.g., revolute or prismatic), its geometric constraints (e.g., a revolute joint keeps two lines attached to both bodies coincident), or its direction of motion must be composed on top of this model as indicated by the ellipsis. More

10 A counterexample is the rhetoric *metaphor* where the *literal meaning* deviates from the implied, *figurative meaning*.

11 https://w3c.github.io/json-ld-syntax/#relationship-to-rdf

```
{
  "@context": [
    "https://comp-rob2b.github.io/metamodels/kinematic-chain/structural-entities.json",
    ...
  ],
  "@id": "rob:joint1",
  "@type": [ "Joint", ... ],
  "between-attachments": [ "rob:link0-joint1",
                           "rob:link1-root" ],
  ...
}
```

Listing 1. Textual model of joint `rob:joint1`. Colors that match with Figure 3 indicate identical entities.

```
1 {
2   "@context": [ ... ],
3   "@id": "rob:joint1-pose",
4   "@type": "Pose",
5   "of": "rob:link1-root",
6   "with-respect-to": "rob:link0-joint1"
7 }
```

Listing 2. Pose relation in JSON-LD.

complicated kinematic chains are represented by ordered collections of joints. Thus, our models can represent kinematic chains of arbitrary topology: serial, tree-structured, and parallel (that is, with one or more cycles).

The pose in Listing 2 is a relation over the *same* frames as those present in the joint relation to represent the coordinate-free position and orientation of the joint's constrained motion. In the context of a `Pose`, these two frames play the role of an `of` frame and a `with-respect-to` (or `wrt`) frame, respectively. Listing 3 then introduces concrete coordinates in `3D-Euclidean` space (as a unitless direction-cosine matrix and a position vector measured in meters). This model shows an example of *multi-conformance*, meaning that an entity can have more than one type to define its semantics, a feature that is rarely encountered in modeling approaches or general-purpose programming languages. Moreover, JSON-LD helps in distinguishing properties by mapping them to IRIs: the `of` property in Listing 2 Line 5 has a different meaning from the one in Listing 3 Line 11. To this end, the *embedded context* maps the latter to the IRI `coord:of-pose` (Line 5), where `coord` is the prefix (or "namespace") defined in Line 3. Additionally, this context defines the `of` property as a symbolic pointer (Line 6). We again notice the recurrence of the same `rob:link0-joint1` frame in Line 12 that we have already encountered above.

The model of a function (or operator) follows the same pattern: it features an identifier, a type, and its properties, as exemplified in Listing 4. The semantics are defined in the metamodel that is referenced by the type. In this example, the operator represents a map from a joint-space position `rob:q1` to a pose `rob:joint1-pose` in Cartesian space. The metamodel also imposes structural constraints, for example, that the joint position and the pose are associated with the same joint. There are two noteworthy remarks. First, the model *represents* an operator but does not "execute" it; instead, that evaluation is the result of a model transformation via some interpretation or compilation. Second, multiple instances of the same operator, that is, operators with the same type, can exist. In that sense, when compared with general-purpose programming languages, the type defined in the metamodel resembles a function declaration, whereas an instance establishes the connection or binding of data structures, similar to a *closure* in functional programming languages. The execution or invocation of such a function is represented by an entity of type `Schedule` with a single property `trigger-chain`[12], an ordered list of symbolic pointers to operators.

Our approach generalizes the geometric relations semantics (GRS) (De Laet et al., 2012) in two ways. Although the GRS do separate the coordinates from their coordinate-free relation, they do not reify the latter. Here, instead, we assign unique identifiers to both representations, which enables us to properly express the one-to-many relations from the former to the latter. Furthermore, we extend the GRS to the models of kinematic chains and to the dynamics solvers on top. This includes physical quantities such as acceleration, force, and inertia together with their operators. Having symbolic models of kinematic chains allows pre-processing or "normalization." This includes, for example, the extraction of a spanning tree, the conversion of all quantities to matching physical units, the composition of static chains of pose relations, or the transformation of inertia to frames that are most suitable for the solvers.

12 This terminology originates from the microblx framework (Klotzbuecher and Bruyninckx, 2013).

```
{
  "@context": {
    "coord": "https://.../coordinates#",
    "PoseReference": "coord:PoseReference",
    "of": { "@id": "coord:of-pose",
            "@type": "@id" }, ...
  },
  "@id": "rob:joint1-pose-coord",
  "@type": [ "3D", "Euclidean", "PoseReference",
             "PoseCoordinate", "DirectionCosineXYZ", "VectorXYZ" ],
  "of": "rob:joint1-pose",
  "as-seen-by": "rob:link0-joint1",
  "unit": [ "UNITLESS", "M" ]
}
```

Listing 3. Pose coordinate representation.

```
{
  "@context": [ ... ],
  "@id": "slv:fpk1",
  "@type": "JointPositionToPose",
  "joint-position": "rob:q1",
  "pose": "rob:joint1-pose"
}
```

Listing 4. Forward position kinematics operator of a joint.

## 5 Tooling implementation: synthesizer and code generator

In this section, we describe our tooling **to synthesize** concrete algorithm models for kinematics and dynamics solvers and **to generate** correct-by-construction code from such models. Synthesis entails deriving data structures, function instances, and a schedule. The generated code can then be seen as a dispatcher of that schedule. Figure 4 depicts the architecture of our toolchain, which consists of three main tools. The *synthesizer* that consumes a model of a kinematic chain, a query model composed on top of that kinematic chain, and a dedicated solver configuration or a "template" of the solver. It produces as output an algorithm model that can be seen as an instantiation of the template along the kinematic chain. This algorithm is fully linked to the kinematic chain model, meaning it is a graph that symbolically points to elements of the kinematic chain. The *IR generator* lowers the algorithm model to an intermediate representation (IR) that the template-based *code generator* then transforms to code in a general-purpose programming language. It is a best practice to keep any logic out of the code generator. Hence, the IR generator performs any pre-processing required for the code generator. Thus, lowering entails the preparation of the algorithm for the code generator by serializing the graph to a tree and introducing any necessary transformations. Finally, given software libraries that provide the pure solver functions, this code is compiled into an executable with a general-purpose compiler. We provide more details on the implementation of all tools and models in the following discussion.

### 5.1 Synthesizer

The overall process of synthesis is a form of **graph rewriting**, that is, matching patterns in the graph and replacing them with new patterns. In general, due to the subgraph isomorphism problem, this is an NP-complete problem (Cook, 1971). However, we can exploit domain-specific knowledge that enables us to (i) guide the traversal over the graph structure; and hence (ii) reduce graph matching on the overall graph to a local neighborhood or even simply localized graph traversals.

We have implemented the synthesizer using the established RDFLib[13] Python library, which also supports the standardized, powerful, and mature graph query language SPARQL (Harris and Seaborne, 2013). The step change in employing this setup is that (i) SPARQL enables the declarative formulation of complicated graph matching and even graph rewriting queries; (ii) in SPARQL the directionality of edges does not constrain traversability so that a query can follow edges in the "opposite" direction, (iii) RDFLib allows "anchoring" these queries in the underlying graph to drastically

13 https://rdflib.dev/

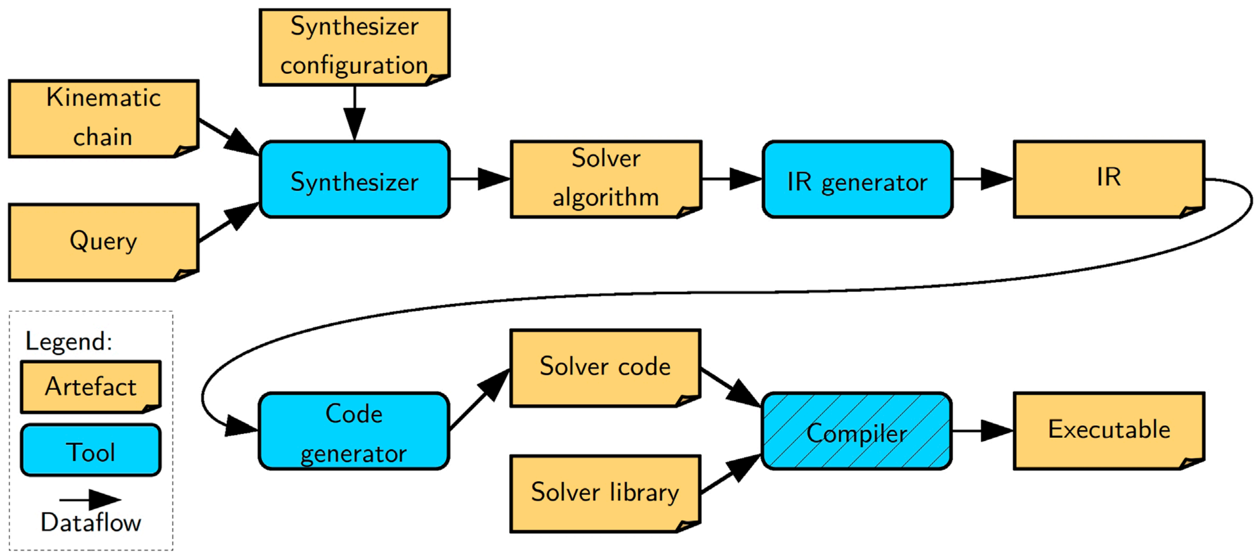


FIGURE 4
High-level architecture of our toolchain showing the three developed tools (blue boxes without hatching), a general-purpose compiler (blue box with hatching), and the artifacts (orange boxes) they consume or produce (arrows).

improve performance by restricting the graph matching to the above-mentioned neighborhood of these "anchor points," and (iv) RDFLib provides a tight interface of custom code with the queries.

The synthesizer features a modular architecture with a small framework core that is complemented by modules to realize the interaction with the graph, for example, by emitting the required data structures and operators for performing the FPK computations. Inspired by the terminology of Gremlin (Rodriguez, 2015), we call each module a *step*. A step *declares* to the framework (i) an expansion query that, on the one hand, determines where the traversal through the graph should continue and, on the other hand, is a first filter criterion to determine when the step applies; and (ii) the functions that implement the graph manipulations at the nodes (either the parent and child or only the child) reached by the expansion. The pure declaration has the benefit that the framework can pre-process and optimize the query execution. Specifically, we have noticed that query execution is a significant contributor to the overall runtime of the synthesizer, but many of these queries tend to be the same. Hence, the framework first clusters all steps with the same expansion query, then executes that query once and afterward dispatches to all steps. The framework also manages a blackboard that it passes to each step. The blackboard is a shared data structure that allows various steps to communicate with each other and incrementally build up the algorithm model. Finally, the framework also realizes the graph traversal as such, with the help of the expansion queries, in a breadth-first manner.

A configuration must be provided to select the types of queries that the synthesizer supports. It consists of a configuration per sweep, an ordered list of steps to be applied during the graph expansion and graph traversal, and the order and direction of these sweeps. As an example, a synthesizer for the FPK problem only requires a single sweep as dictated by physics and evident by Algorithm 1, whereas a hybrid dynamics solver demands three sweeps.

### 5.1.1 Graph expansion by example

We use the FPK solver to exemplify the synthesis in Figure 5. This figure shows an excerpt of a kinematic chain model in the lower box, which is a visual representation of the models from Listings 1 and 2 with an additional joint position `q1` composed on top.

As a first step, the synthesizer determines the traversal, that is, which parts of the graph to visit and in which order. Most computations in the kinematics and dynamics solvers propagate quantities between "local" root frames on adjacent links. The SPARQL query in Listing 5 identifies such frames by a transition over a link and over a joint. Assume that the traversal starts at the frame `link0-root`. This is then the `?node` argument passed to the expansion query. Hence, the query tries to follow the `geom-ent:simplices` first in the "inverse" direction, as indicated by the caret, which would bring it to the `link0` node, and then in the "forward" direction so that it arrives at both the `link0-joint1` node *and* back at the `link0-root` node. Next, the `FILTER` statement eliminates the `link0-root` node. With the same logic applied to the `kc-ent:between-attachments` edges, the traversal arrives at the `link1-root`, which is designated as `?child`. Line 4 finally returns any found child (and also the original input node as the parent) as a result of the query.

### 5.1.2 Graph manipulation by example

Next, we investigate how the position propagation step (Algorithm 1 Line 2) manipulates the graph. At first, the step registers a set of visitors, or callbacks, with the framework. During this registration procedure, the step declares the conditions for when these visitors should be executed. The conditions include the mandatory expansion query and further optional queries, for instance, to check if the traversal is currently visiting a leaf node. Finally, the step can decide whether to visit the edge, in which case it receives the parent and child as argument, or

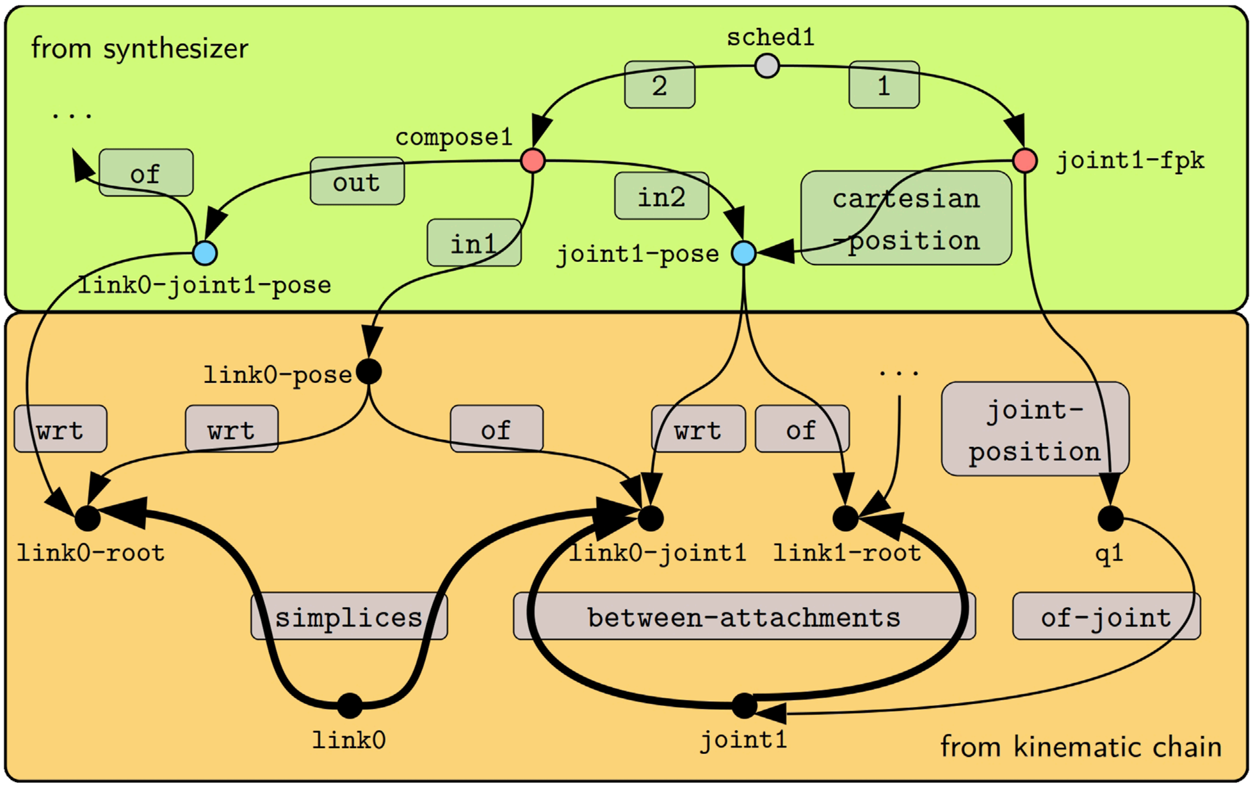


FIGURE 5
Given the kinematic chain model from Listing 1 (lower orange box), the synthesis step for the FPK algorithm emits a model (upper green box) of the data structures indicated by blue circles (see Listings 2 and 3), the operators indicated by red circles (see Listing 4), and the schedules indicated by gray circles. Edges are labeled by gray panels. The thick edges show how the query in Listing 5 traverses the graph from the start node `link0-root` (the `?parent`) to the `link1-root` node (the `?child`).

```
PREFIX geom-ent: <https://.../geometry/structural-entities#>
PREFIX kc-ent: <https://.../kinematic-chain/structural-entities#>

SELECT ?child ?parent WHERE {
    ?node ^geom-ent:simplices
          / geom-ent:simplices ?joint_prox .
    FILTER (?node != ?joint_prox)
    ?joint_prox ^kc-ent:between-attachments
                / kc-ent:between-attachments ?child .
    FILTER (?joint_prox != ?child)
    BIND(?node AS ?parent)
}
```

Listing 5. SPARQL query for frame-to-frame traversal expansion.

whether to only visit the child node. In addition to the expansion query from Listing 5, the position propagation step does not declare any further conditions. Furthermore, this step requires access to the parent's and child's states, so it employs an "edge visitor." The step necessitates two passes: a *configuration* pass to instantiate the algorithm's data structures and a *computation* pass to instantiate the operators and append them to the schedule.

Continuing with the example in Figure 5, we notice that `link0-joint1` is eligible for the position propagation step because it has been reached by the expansion query. During the configuration pass, the step obtains handles to the `link0-root-to-joint1` pose, the `link0-joint1` frame, and the joint position `q1` to then emit the two poses `link0-joint1-to-link1-root` and `link0-root-to-link1-root`.

```
{
    ...,
    "variables": {
        "rob_q1": { ... },
        "rob_joint1_pose": { ... },
        ...
    },
    "closures": {
        ...
        "rob_joint1_fpk": {
            "operator": "joint-position-to-pose",
            "dimensions": 3,
            "joint": "rev_z",
            "joint-position": "rob_q1",
            "pose": "rob_joint1_pose"
        },
        ...
    },
    "schedule": [
        ...
        "rob_joint1_fpk",
        ...
    ]
}
```

Listing 6. IR of algorithm.

Additionally, the joint position and all three poses are added to the blackboard. This enables the computation pass to access them and emit two operators. The first maps the joint position to Cartesian space (`joint1-fpk`). The second composes the pose of the link and the pose over the joint (`compose1`). Finally, the pass adds both operators to the schedule (`sched1`). For this example, we have used human-readable identifiers for all created models; however, in the implementation, we have instead opted for randomly generated universally unique identifiers (UUIDs) as defined by Leach et al. (2005).

### 5.1.3 Parallel kinematic chains

Handling parallel kinematic chains requires the interplay of the graph traversal and the visitors. First, during the traversal, each expanded node is assigned a depth, that is, its minimal distance from the node where the traversal started. Then, the visitors feature conditions to handle the different types of edges: *cross edges* connect nodes with the same depth, *forward edges* connect from nodes with lower depth to nodes with higher depth, and *vice versa* for *back edges*. The concrete graph manipulation to be performed for each type of edge is again part of the step. An example in the context of the FPK is to insert a computation that checks, at runtime, if the poses each way around the cycle are consistent. Alternatively, a dynamics solver could insert acceleration constraints as described in Section 2.

## 5.2 Code generation

We have implemented the code generator using the StringTemplate[14] library and its JSON frontend StringTemplate Standalone Tool[15]. StringTemplate enforces the separation of logic from rendering templates and is one of the few template engines that has scientific justifications for its design and the included and excluded primitives (Parr, 2004). A graphical user interface, the "Inspector," allows visually debugging the generated code by tracing each rendered token back to a template fragment and its input data. Furthermore, the StringTemplate library is extensively used in the ANTLR parser generator (Parr, 2013).

To bridge the gap between the complete graph models and the template engine, we have introduced the IR and its generator. Its objectives are three-fold. First, because the templates are logic-free, the IR generator performs any necessary processing (e.g., filtering strings so that they represent valid identifiers or embedding information for the template into the IR) on the graph model. Second, it transforms the graph into a tree structure by cutting loops and replacing them with symbolic pointers. Finally, it serializes the resulting graph to JSON, as exemplified in Listing 6. The excerpt of an IR model contains (i) the `variables`, which is a dictionary with all required data structures, their types, sizes, or initial values;

14 https://www.stringtemplate.org/

15 https://github.com/jsnyders/STSTv4

```
 1 application(..., closures, schedule) ::= <<
 2 int main() {
 3     <...>
 4     <schedule:statement(closures); separator="\n">
 5     return 0;
 6 }
 7 >>
 8
 9 statement(closure-id, closures) ::= <<
10 <({<closures.(closure-id).operator>})(closures.(closure-id))>;
11 >>
12
13 ...
14
15 joint-position-to-pose(args) ::= <<
16 dyn2b_<args.joint>_to_pose<args.dimensions>(
17     <args.joint-position>, <args.pose>)
18 >>
```

Listing 7. StringTemplate excerpt for generating the solver implementation. Colors align with the IR from Listing 6.

(ii) the `closures`, a dictionary of operators with their connections to data structures as symbolic pointers (`joint-position` and `pose`) and additional properties (`dimensions` and `joint`); and (iii) the `schedule` as an ordered list that contains the symbolic pointers to the `closures` dictionary.

Listing 7 shows an excerpt of a template model for generating C code. The snippet consists of three rules in Lines 1, 9, and 15. As such, these templates align with our objective of composability because (i) *every* rule is labeled by an identifier and, hence, can be referenced, while (ii) higher-level rules dispatch to lower-level rules. This structure mirrors that of parsers for formal languages, but instead of constructing an abstract syntax tree (AST), it renders text from an AST. The top-level `application` rule accepts several parameters and then defines the to-be-rendered text between the double angle brackets << and >> (single angle brackets < and > contain StringTemplate processing directives). Here, only the two arguments `closure` and `schedule` are shown that align with the IR from Listing 6. The application consists of the program's main function, which first defines and initializes all required variables (not shown) from the algorithm model's data structures and then emits the function calls. Line 4 iterates over the `schedule` and applies the `statement` rule to each entry with an implicit argument of the currently visited entry and the `closures` dictionary. Any two generated lines will be separated by a line break. The `statement` shows another StringTemplate pattern: the notation `closures.(closure-id)` performs a lookup in the `closures` dictionary with the value of `closure-id` as key. Here, another dictionary is returned in which the `().operator` retrieves the value associated with the operator. The directive (`{rule-id}`) (...) then dispatches to the rule `rule-id`, which is the `joint-position-to-pose` in this example. This last rule finally renders the function call with the provided arguments.

The real implementation separates the top-level *application* template from the reusable and domain-specific *fragments*. We also see that fragments relate to different domains, such as the algorithm model (`statement` rule) or the kinematics model (`joint-position-to-pose` rule), and are, hence, located in separate files. As a result, we can efficiently compose a variety of applications by relying on StringTemplate's import feature to include only the necessary fragments in a top-level template.

## 5.3 dyn2b: support library for computational building blocks

We have also implemented a C software library called `dyn2b` that realizes the numerical computations for kinematics and dynamics solvers at runtime. `dyn2b` is designed for composability in that it only provides pure functions at the granularity required by the synthesizer and code generator. First, pure functions are free from side effects, which means that any state must be passed into the functions as explicit arguments to allow for their arbitrary, even reentrant, execution. Second, this design prevents the state from remaining hidden behind a private interface. All too often, algorithm or function developers cannot foresee the context in which their artifacts will be used and, hence, should not introduce preliminary decisions to hide the state. As an example, we have noticed this limitation in the development of an online identification procedure for dynamic parameters that relied on the KDL. Here, the inertial parameters are hidden inside a class that prevents them from being updated using an estimator. Most solver libraries, including RBDL and Pinocchio, already follow such a design that avoids encapsulation: the kinematic chain's model and/or the solver's state live in separate yet pre-defined data structures that are publicly accessible. Third, the separation of data from the computations enables (i) the optimization of the data layout for the hardware at hand, including the order of the data structures and their alignment to memory boundaries or cache lines; (ii) state persistence, for instance, by streaming part of the state to a database; or (iii) instrumentation of that state at specific points in time.

The provided functions mainly comprise 3D-Cartesian space kinematics and dynamics to propagate and accumulate the compact representation (cf. Featherstone (2008)) of screws and inertia and functions that map between joint space and Cartesian space for revolute and prismatic joints. In contrast to Featherstone (2008), we do not distinguish between velocity vectors and force vectors but only implement a generic set of functions for screws. The reason is that the type checking, including the checking of additional semantic constraints, is performed on the model level; the code generator then only dispatches to the correct numerical functions. The screw operators act on collections of screws instead of individual screw vectors so that they can efficiently handle the multiple instances of motion drivers. `dyn2b` explicitly excludes functions for highly variable domains, such as operators associated with more sophisticated joints, stiffness, and damping due to their non-linear behavior, numerical integrators, or trajectory generators. All of these warrant their custom set of models, tools, and software libraries.

`dyn2b` is compatible with and is built upon the Basic Linear Algebra Subprograms (BLAS) and Linear Algebra Package (LAPACK) libraries. For now, we rely on the very much unoptimized Netlib reference implementations of both libraries. Hence, a significant performance improvement may be possible with a (highly localized) change to a dedicated BLAS implementation for small-scale linear algebra. Here, the role of a general-purpose language compiler is to generate efficient *numerical* code by program-level optimizations such as code inlining, vectorization, constant propagation, or introducing platform-level details, including calling conventions, instruction scheduling, register allocation, and machine code generation.

### 5.4 Steps for dynamics solvers

Analogous to the synthesis and code generation example above, we have also realized the building blocks for further kinematics solvers up to the acceleration level as well as dynamics solvers. Here, we outline the main challenges and policies that we address in our implementation for such solvers.

The main difference for traversing the graph during the inward sweep relates to the handling of leaves, loop-closures, and branching points. We have already discussed the case of an unconditional edge visitor in Section 5.1.2. In contrast, to handle leaves, a step registers a conditional node visitor with the framework that will be called with only the currently visited node as argument. The condition resembles Listing 5, but (i) is a boolean-valued SPARQL ASK query that (ii) checks whether *no* joint follows the link; that is, it is a leaf. The concrete graph manipulation instructions depend on the quantity or motion driver; for instance, inertial force vectors are initialized to zero vectors, whereas the propagated inertia matrix is initialized with the leaf link's inertia matrix. As for branching points, such as link 1 in Figure 2A, the synthesis step emits operations to accumulate inertia and force over all children of the currently visited segment. Because a serial connection is a special case of a branched connection, we employ the same steps for both in our synthesis tool. The only difference is that the *synthesized* algorithm contains more data structures and operations for branching points.

Next, we turn to the propagation of motion drivers through the kinematic chain. As an example, Figure 2A depicts two instances of external force motion drivers ($\boldsymbol{F}_{ext,3}$ and $\boldsymbol{F}_{ext,4}$). Both instances are propagated inward to arrive at link 1, which now "feels" the propagated effect of both forces as $\boldsymbol{F}'_{ext,3}$ and $\boldsymbol{F}'_{ext,4}$. Traditional solvers would accumulate their effect by adding both forces to minimize the overall number of force variables and, hence, maximize the computational efficiency of the solvers. In contrast, following our recent work (Schneider and Bruyninckx, 2019), the steps for the inward sweep decompose the kinematic chain's state by propagating all forces and their instances in separation. Then their combined effect at link 1 can be represented as a list: $\left(\boldsymbol{F}'_{ext,3} \quad \boldsymbol{F}'_{ext,4}\right)$. In other words, accumulation here means to append. In this setting, it is the role of the synthesis steps to perform the bookkeeping of individual, propagated forces, which includes (i) tracking the sizes of the lists per segment so that their memory can be pre-allocated; (ii) computing the indices into the lists so that each force can be found; and (iii) symbolically associating each propagated force with its original motion driver. For a human consumer, similar names establish the link to the original motion driver, but in the models, a separate relation facilitates the traceability of propagated forces to their original motion specification.

On the one hand, the decomposition during the inward sweep is computationally more expensive than the inward sweep in traditional solvers. On the other hand, it also enables the flexible recomposition of the motion drivers during the final outward sweep or solver sweep. Examples include (i) weighing or prioritizing motion drivers with respect to each other; (ii) avoiding actuator saturation by scaling down some motion drivers in accordance with the motion specification; or (iii) using the decomposed state in model-based controllers (MPC).

## 6 Case study

The objective of the case study is multi-fold. First, it demonstrates the algorithm synthesis and code generation from composable models. Second, it shows the iterative and incremental modeling and development of a concrete application together with its integration into a real robot. Third, it provides evidence of the models' composability because the application is composed of the solvers' algorithm models. Finally, it demonstrates compositionality by performing semantic algorithm manipulation. The case study follows the code-centric tutorial that is available together with the toolchain. The objective is to compose a controller and a robot interface onto a recursive Newton–Euler algorithm (RNEA). Afterward, we systematically inject instrumentation operations into the resulting algorithm. Our target platform is a Kinova Gen3[16] manipulator for which we have created composable models in JSON-LD[17].

To synthesize the RNEA, we configure the toolchain with two sweeps. The first sweep realizes the outward propagation and accumulation of poses, twists, and acceleration twists. The second sweep realizes the inward propagation of Cartesian-space inertial forces that compensate for gravity and velocity-dependent accelerations. The inward sweep also computes the joint-level

16 https://www.kinovarobotics.com/product/gen3-robots

17 https://github.com/comp-rob2b/robot-models

control torques associated with the inertial forces. Both sweeps are built from the available steps and code generator fragments that we have explained above. In practice, a robot will have to synthesize (part of) an application whenever the graph changes structurally. For traditional solutions that rely on manually implemented solvers, this high variability leads to a combinatorial explosion for even moderately sized applications, which makes it challenging to design and verify the developed software in advance. Hence, we tackle such problems with our toolchain to synthesize and generate correct-by-construction solvers from verifiable specifications.

## 6.1 Damping controller and robot interface

The first extension is a Cartesian-space motion controller. Its role is to realize the robot's behavior over a longer time span as contrasted with the solver, which only realizes the instantaneous mappings of force and motion inputs to the control commands of the kinematic chain's actuators. For the case study, we have opted to demonstrate the controller attachment using a simple damping controller. The controller's objective is to move the robot's end-effector while limiting the maximum velocity $v_{max}$ and maximum force $f_{max}$ that it could exert on its environment. The controller computes the commanded force as shown in the following Equation 1.

$$f_{cmd} = f_{max} - \frac{f_{max}}{v_{max}} v_{msr} \tag{1}$$

In this equation, the damping is the intermediate term $D = -\frac{f_{max}}{v_{max}}$ that is scaled by the currently measured velocity $v_{msr}$. Intuitively, this means that the controller (i) at $v_{msr} = 0$ commands the maximum force so that the robot can accelerate if it is not impeded; (ii) at $v_{msr} = v_{max}$ commands no force so that the robot does not accelerate further; and (iii) otherwise linearly interpolates between both cases with $D$ as the proportionality factor. This behavior is depicted in Figure 6. For this case study, we have connected the controller to the robot's linear upward motion (as seen by the world frame). Hence, the robot always tries to move to a fully stretched-out configuration (a workspace singularity) but can manually be displaced. When the operator holds the robot in place, they feel the robot "pushing" upward with the maximum configured force. If they move it upward too fast (beyond the configured maximum velocity), they feel the robot actively counteracting by braking. An alternative application of this controller is to bring the robot into safely controlled contact with its environment and then align it without relying on additional exteroceptive sensors. As a second extension, we have modeled and implemented a robot interface. The robot interface model attaches to the various joint-level quantities to read joint positions and joint velocities from the robot's sensors and to command torques to the actuators. The solver handles these attachments by a step to read measurements in the first sweep and another step to write commands in the last sweep. We support two backends: `robif2b`[18] to operate the real robot and a simple simulation that provides fixed measurements while printing joint-level commands to the terminal.

18 https://github.com/rosym-project/robif2b

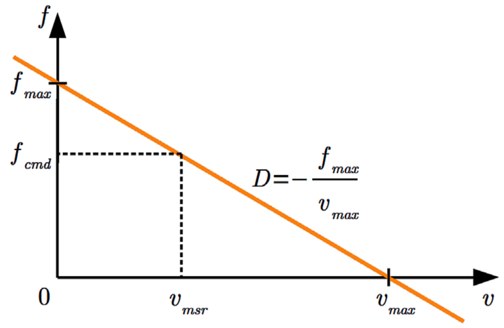


FIGURE 6
Depiction of the damping controller's behavior. The linear damping $D$ is computed from a given maximum velocity $v_{max}$ and a maximum force $f_{max}$, as indicated by the orange line. The force to command the robot $f_{cmd}$ depends on the measured velocity $v_{msr}$. In free space, the force accelerates the robot to maximum velocity, whereas in a rigid contact, the robot does not move while still limiting the commanded force.

## 6.2 Semantic algorithm manipulation

The algorithm model already enables simple queries to gather statistics about an algorithm or even a complete application. A simple example is to count the number of functions or data structures of a particular type. This may provide insights into the expected performance or memory requirements of the algorithm. However, more interesting queries concern the instrumentation of the running software. To this end, Listing 8 demonstrates a complicated query that inserts a logger into an existing schedule. Apart from the prefix definition at the top, the query consists of three main parts: `DELETE` to remove elements from the RDF graph, `INSERT` to extend that graph, and `WHERE` to localize the deletion and insertion points. We start with the latter, which has the goal of finding an operation `?op` of type `Damping` (Line 19) with an input property `velocity-twist` and an output property `wrench`. This operation is part of the `?schedule`'s trigger chain (Line 15), the totally ordered collection of operations. RDF represents such a collection as a singly linked list, a set of anonymous nodes that (i) point to the concrete content (here, the operations) via the `first` property; and (ii) are linked among each other by the `rest` property that points to the next node. The traversal of this list reads as follows: (i) start at the `?schedule` node; (ii) follow one step along the `trigger-chain` property; (iii) follow an arbitrary number of steps (indicated by the asterisk) along the `rest` property to visit any list node; and (iv) for each of these nodes, follow to the content via the `first` property to an operation `?op` that must satisfy the above constraints. The query also creates a new UUID as an identifier for the logger (Line 22). The statement in Line 9 instantiates the new logger with a single property `quantity` that represents an ordered list—indicated by the parentheses—of data structures to log. Notice that apart from the twist and wrench, we also log the joint position of the manipulator's second joint `q2` that experiences the greatest displacement. Finally, Lines 7, 11, and 17 represent, respectively, the cutting, splicing, and bookkeeping of inserting the logger at the correct location into the linked list.

For demonstration purposes, we have implemented a simple backend for this logger model that, at runtime, writes the data into

```
PREFIX rob: <https://comp-rob2b.github.io/robots/kinova/gen3/7dof/>
PREFIX algo: <https://comp-rob2b.github.io/metamodels/algorithm#>
PREFIX ctrl: <https://example.org/ctrl#>
PREFIX log: <https://comp-rob2b.github.io/metamodels/logging#>

DELETE {
    ?ins_ptr rdf:rest ?rest .
} INSERT {
    ?log a log:Logger ;
        log:quantity (?twist ?wrench rob:q2) .
    _:b1 rdf:first ?log ;
        rdf:rest ?rest .
    ?ins_ptr rdf:rest _:b1 .
} WHERE {
    ?schedule a algo:Schedule ;
        algo:trigger-chain / rdf:rest* / rdf:first ?op .
    ?ins_ptr rdf:first ?op ;
        rdf:rest ?rest .
    ?op a ctrl:Damping ;
        ctrl:velocity-twist ?twist ;
        ctrl:wrench ?wrench .
    BIND(UUID() AS ?log)
}
```

**Listing 8. SPARQL update query to insert a logger into a schedule.**

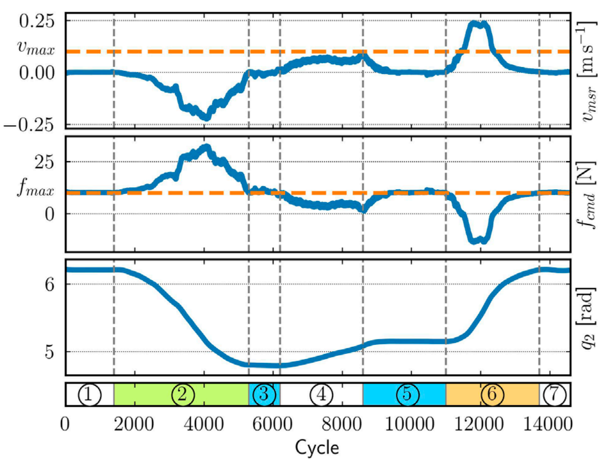


FIGURE 7
Visualization of the logged data recorded on the real robot. The plot shows the relationship between the end-effector's measured upward velocity (top), the upward control force (middle), and the second joint's position (bottom). The seven labeled phases comprise the robot being fully stretched out (1 and 7), manually pushed downward (2) and upward (6) or held in place (3 and 5) by a human operator, and moving up without contact (4). The controller is configured with $v_{max}$ = 0.1 m s$^{-1}$ and $f_{max}$ = 10 N.

a comma-separated value file. The drawback of this approach is that it potentially introduces high amounts of jitter into the real-time control loop. Hence, a more sophisticated approach would rely on a realtime-capable communication infrastructure, such as ring buffers, to send the data to a dedicated log writer. Figure 7 depicts part of this logged data as recorded from the real robot that is executing the above application while intermittently being impeded by a human operator (indicated by colored segments).

The robot starts in a stretched-out configuration, a workspace boundary, so that it points vertically upward. Because it cannot move further in that direction, the controller commands a maximum force of $f_{max}$ = 10 N. Next, the operator physically displaces the robot until it is parallel to the ground. During this second phase, we see that the robot moves faster than $v_{max}$ = 0.1 m s$^{-1}$ and, hence, the controller counteracts with a control force greater than $f_{max}$. After releasing the arm, in the fourth phase, its velocity gradually increases so that the commanded force reduces. Then, the operator pushes the robot upward so that it exceeds $v_{max}$. Here, the controller actively brakes to counteract that upward motion. Not surprisingly, due to the affine control law (scaling and translation), the control command is similar to the measured velocity. Finally, the arm reaches the initial, stretched-out configuration again.

## 7 Discussion

A major inspiration for the toolchain originates from the *strategic programming* paradigm (Lämmel et al., 2003). Although mostly targeted at tree structures, its objective is to separate the *traversal control* from the *logic* that is applied at visited nodes. Here, the traversal strategies are composed of atomic traversal steps and higher-order functions that are called *combinators*. Especially *adaptive programming*, as found in the DJ library (Orleans and Lieberherr, 2001), employs a similar approach as our toolchain by defining graph visitors and their declarative traversal specifications that are then dispatched on a graph of Java objects. With those insights, we refactored our synthesis tool by separating the graph traversal, the synthesis steps, and the blackboard that contains the shared state. Additionally, we realized the query declaration and implemented the caching to reduce the amount of overall query executions.

As shown in the high-level architecture diagram in Figure 4, the individual tools rely on explicit models instead of language-specific APIs. On the one hand, this design allows each tool to be realized with the most suitable programming language and choice of software libraries. On the other hand, also the model representation between two tools can be changed if the producing and consuming tools are adapted. In either case, such a change remains very localized. Here, we will review further variation points and their associated one-to-many mappings with respect to model representation, graph querying languages, template engines, and the execution backend.

The same models can be serialized in various interchange formats, with XML (Bray et al., 2006) and JSON as the most common options. Two models can even be equivalent semantically yet differ structurally. For instance, a `RigidBody` constraint could be part of an entity's type or, alternatively, tagged to that entity by a relation. In any case, if the models represent the same information, they can always be model-to-model transformed into each other. In the Semantic Web, the Web Ontology Language (OWL) defined by Bock et al. (2012) provides multiple concepts to perform such transformations.

The decision to use SPARQL was mainly driven by its support for declarative graph matching and its vendor-independent and mature standard, together with the availability of RDFLib. RDFLib eased the integration with our Python toolchain, in particular, due to the in-memory, in-process database that avoids a dedicated database setup. However, in our prior work (Hochgeschwender et al., 2016), we also employed the declarative graph query language Cypher (Francis et al., 2018) that originates from the Neo4j database. Cypher supports graph matching and, as of now, is in the process of being standardized. The imperative language Gremlin (Rodriguez, 2015) is another popular alternative for graph querying that supports graph matching.

We prototyped the code generation with the more popular template engine Jinja[19] in Schneider et al. (2023). Although the integration with the Python-based toolchain was easy, we quickly noticed the problem of interleaving logic with the templates and a tight coupling of the templates with the execution environment. Examples include the injection of Python functions into the templates as processors or filters and exposing the database interface to the templates. Additionally, in Jinja, the entry point is always an "anonymous" template (not a rule as in StringTemplate) that terminates the composition hierarchy at the top. Developer discipline and macros (akin to rules in StringTemplate) can help, but "clean" templates are not enforced.

Especially when the applications grow more complicated, it may be worthwhile to explore different starting points for the algorithm synthesis. Currently, every synthesis execution starts from scratch. On a computer with an Intel Core i7-4790K CPU and 16 GB RAM, the synthesis takes approximately 1.5 s, while the code generator finishes in approximately 0.3 s. However, it is important to note that this is a design-time cost and *not* part of the real-time path during the running application; the generated code itself *is* real-time capable, that is, it always performs a maximum amount of operations, each with a deterministic runtime. Still, when only some models change, it may be computationally more efficient to reuse previously synthesized algorithms and specialize or modify them in a post-processing step. Another variation point comprises the types of generated artifacts. Because the case study is only an excerpt of the overall application, it suffices to generate completely static code. A simple extension to efficiently change the software's behavior is to expose and adapt some of the data structures at runtime. Another extension is to generate runtime-composable functions, such as the cascades in a controller, which can be hooked into an application-level event loop to be executed at different cycle rates.

There exists an overlap of our approach and toolchain with functionality in the ROS ecosystem. As mentioned above, the structural models in ROS systems are represented in URDF and SRDF. The `robot_state_publisher`[20] and *tf* (Foote, 2013)[21] packages provide the software to bring these models to life by realizing the runtime behavior. However, only the FPK computation from Algorithm 1 is realized by these packages: the `robot_state_publisher` evaluates Line 2, whereas *each* instance of a *tf* listener computes Line 4. These two types of computation are tightly coupled to the ROS communication infrastructure that serializes, sends, receives, and deserializes all pose relations, even if the involved nodes run on the same computer. Once all poses have been accumulated, the *tf* listener can answer queries that require the transformation of points or vectors between coordinate frames. *tf* effectively only supports position-level kinematics: twists can only be approximated by discrete differentiation of poses. Acceleration twists, dynamic quantities and their operators, and maps from Cartesian space to joint space remain completely absent. Additionally, the transformation graph in *tf* must always form a tree. This is caused by the stateless publish-subscribe communication: any node can provide new transforms at any point in time so that the *tf* listener must always construct and evaluate the transform graph anew, which requires a tree as an efficient yet limiting data structure. For similar reasons, *tf* does not support ahead-of-time or just-in-time validation to answer questions such as "*Do the frames in a query exist?*" or "*Does the transform graph actually form a tree?*"

Our approach compares favorably to the currently hyped large language models (Vaswani et al., 2017): it is an engineered solution that is explicit in the represented knowledge, which enables explainability. In other words, composable models represent *exactly* what is necessary, nothing more and nothing less. These properties are also required to certify such a toolchain for safety-critical systems.

## 8 Conclusions and future work

In this article, we present the *graph-based solver* pattern that recurs in various seemingly unrelated robotic domains and is the foundation of various efficient algorithms that act on

19 https://palletsprojects.com/p/jinja/

20 http://wiki.ros.org/robot_state_publisher

21 We use *tf* and *tf2* synonymously.

graph structures. We delve into the details of the pattern by performing an in-depth analysis of solvers on kinematic chains. The supplementary material extends the analysis for probabilistic networks and factor graphs, data-flow models, and expression graphs. We complement the composable models with a model-based engineering toolchain to synthesize such algorithms from the algorithmic building blocks: the data structures, pure functions that act on these data structures, and schedules that describe control flows as sequences of functions. A core element of this toolchain is the synthesizer that accommodates various concerns, including (i) multiple, structured traversals over potentially cyclic graphs; (ii) dispatching computations at specialized node types, both in terms of the graph structure (e.g., at branch nodes, leaf nodes or cycle edges) but also the domain-specific semantics (e.g., at geometric frames, rigid bodies, or kinematic joints); (iii) the algorithm management such as performing memory allocation or triggering computations; and (iv) the incremental construction of the overall algorithm where the operations must have access to a prior *state* from the same sweep or previous sweeps. The synthesizer is an application of higher-order, graph-based reasoning that relies on established standards and mature software libraries. We generate correct-by-construction code from the synthesized algorithm that is complemented by a low-level numeric library to perform the computations required for kinematics and dynamics solvers of rigid-body systems. In a case study, we evaluate our approach on a real robot and demonstrate how the explicit algorithm model facilitates semantic algorithm manipulation.

The proposed approach paves the way to have models for the robot's complete life-cycle, including runtime aspects of the system. Hence, as future work, we foresee the exploitation of *all* of the above models so that robots can adapt their software themselves even at runtime. To this end, we have already developed a proof-of-concept tool using the llvmlite[22] library, a Python interface to LLVM's[23] just-in-time (JIT) compiler. Additionally, we plan to apply and extend the models and the tools to more complicated applications involving multi-robot systems that must cooperate in challenging manipulation tasks. It is in such systems that the models will pay off the most, due to the complicated robot models and world models that are connected by task or motion descriptions. Here, the graph structure helps in coordinating and configuring a wide range of algorithms, including monitors, controllers, or estimators that are associated with the many relations in the graph.

## Data availability statement

The code and models for this article can be found in the following repositories: Synthesis tool and code generator templates: https://github.com/comp-rob2b/kindyngen. Metamodels that define themodel semantics: https://github.com/comp-rob2b/metamodels. Models for the Kinova Gen 3manipulator: https://github.com/comp-rob2b/robot-models. Kinematics and dynamics software library: https://github.com/comp-rob2b/dyn2b.

## Author contributions

SS: conceptualization, data curation, formal analysis, investigation, software, methodology, visualization, writing–original draft, writing–review and editing. NH: conceptualization, funding acquisition, methodology, project administration, supervision, and writing–review and editing. HB: conceptualization, funding acquisition, methodology, project administration, supervision, and writing–review and editing.

## Funding

The author(s) declare that financial support was received for the research, authorship, and/or publication of this article. Sven Schneider and Nico Hochgeschwender gratefully acknowledge the ongoing support of the Bonn-Aachen International Center for Information Technology. This work was supported by the European Union's Horizon 2020 project *SESAME* (H2020-101017258, *Secure and Safe Multi-Robot Systems*). Nico Hochgeschwender gratefully acknowledges the support of the SOPRANO (Horizon Europe – 101120990) Project. Herman Bruyninckx gratefully acknowledges the support of *RobMoSys* (H2020-732410, *Composable Models and Software for Robotics Systems-of-Systems*) and *Esrocos* (H2020-730080, *European Space Robot Control Operating System*).

## Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## Publisher's note

## Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/frobt.2024.1363150/full#supplementary-material

22 http://llvmlite.pydata.org/
23 https://www.llvm.org/

## References

Aertbeliën, E., and De Schutter, J. (2014). "eTaSL/eTC: a constraint-based task specification language and robot controller using expression graphs," in *Proc. IEEE/RSJ international conference on intelligent robots and systems (IROS)*.

Bock, C., Fokoue, A., Haase, P., Hoekstra, R., Horrocks, I., Ruttenberg, A., et al. (2012). "OWL 2 Web Ontology Language structural specification and functional-style syntax," in *W3C recommendation*. Second Edition (W3C standard). Available at: https://www.w3.org/TR/owl2-syntax/.

Bray, T. (2017). The JavaScript object notation (JSON) data interchange format. *RFC 8259, Internet Eng. Task Force (IETF)* (IETF standard). Available at: https://datatracker.ietf.org/doc/html/rfc8259.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cowan, J. (2006). *Extensible markup language (XML) 1.1*. Second Edition. W3C Recommendation, World Wide Web Consortium W3C. Available at: https://www.w3.org/TR/2006/REC-xml-names11-20060816/.

Bruyninckx, H. (2023). in *Building blocks for complicated and situational aware robotic and cyber-physical systems* (KU Leuven: Department of Mechanical Engineering).

Carpentier, J., Saurel, G., Buondonno, G., Mirabel, J., Lamiraux, F., Stasse, O., et al. (2019). "The Pinocchio C++ library – a fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives," in *IEEE/SICE international symposium on system integration (SII)*.

Chignoli, M., Adrian, N., Kim, S. b., and Wensing, P. (2023). "Improving contact-rich robotic simulation with generalized rigid-body dynamics algorithms," in *Embracing contacts – workshop at ICRA 2023*.

Codd, E. F. (1971). Further normalization of the data base relational model. *Tech. Rep*. IBM Research Laboratory.

Cook, S. A. (1971). "The complexity of theorem-proving procedures," in *Proc. ACM symposium on theory of computing*.

Cyganiak, R., Wood, D., and Lanthaler, M. (2014). *RDF 1.1 Concepts and abstract syntax*. W3C recommendation. *World Wide Web Consort. (W3C)*. Available at: https://www.w3.org/TR/rdf11-concepts/.

De Laet, T., Bellens, S., Smits, R., Aertbelien, E., Bruyninckx, H., and De Schutter, J. (2012). Geometric relations between rigid bodies (Part 1): semantics for standardization. *IEEE Robotics & Automation Mag.* 20, 84–93. doi:10.1109/mra.2012.2205652

Dellaert, F. (2012). "Factor graphs and GTSAM: a hands-on introduction," in *Tech. rep.* (Atlanta: Georgia Institute of Technology).

Duerst, M., and Suignard, M. (2005). Internationalized Resource identifiers (IRIs). *RFC 3987, Internet Eng. Task Force (IETF)*. Available at: https://datatracker.ietf.org/doc/html/rfc3987.

Featherstone, R. (2008). *Rigid body dynamics algorithms*. Springer.

Felis, M. L. (2016). RBDL: an efficient rigid-body dynamics library using recursive algorithms. *Aut. Robots* 41, 495–511. doi:10.1007/s10514-016-9574-0

Foote, T. (2013). "Tf: the transform library," in *Proc. IEEE international conference on technologies for practical robot applications (TePRA)*.

Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., et al. (2018). "Cypher: an evolving query language for property graphs," in *Proc. International conference on management of data (SIGMOD)*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

Harris, S., and Seaborne, A. (2013). SPARQL 1.1 query language. W3C recommendation. *World Wide Web Consort. (W3C)*. Available at: https://www.w3.org/TR/sparql11-query/.

Hochgeschwender, N., Schneider, S., Voos, H., Bruyninckx, H., and Kraetzschmar, G. K. (2016). "Graph-based software knowledge: storage and semantic querying of domain models for run-time adaptation," in *Proc. International conference on simulation, modeling, and programming for autonomous robots (SIMPAR)*.

Hunt, A., and Thomas, D. (2019). *The pragmatic programmer*. Reading: Addison Wesley.

Jain, A. (2012). Multibody graph transformations and analysis Part II: closed-chain constraint embedding. *Nonlinear Dyn.* 67, 2153–2170. doi:10.1007/s11071-011-0136-x

Kent, W. (1983). A simple guide to five normal forms in relational database theory. *Commun. ACM* 26, 120–125. doi:10.1145/358024.358054

Klotzbuecher, M., and Bruyninckx, H. (2013). "microblx: a reflective, real-time safe, embedded function block framework," in *Proc. Real time linux workshop (RTLWS)*.

Knublauch, H., and Kontokostas, D. (2017). *Shapes Constraint Language (SHACL)*. W3C recommendation. *World Wide Web Consort. (W3C)*. Available at: https://www.w3.org/TR/shacl/.

Lämmel, R., Visser, E., and Visser, J. (2003). "Strategic programming meets adaptive programming," in *Proc. International conference on aspect-oriented software development (AOSD)*.

Leach, P. J., Salz, R., and Mealling, M. H. (2005). A universally unique IDentifier (UUID) URN namespace. *RFC 4122, Internet Eng. Task Force (IETF)*. Available at: https://datatracker.ietf.org/doc/html/rfc4122.

Mansard, N., Khatib, O., and Kheddar, A. (2009). A unified approach to integrate unilateral constraints in the Stack of tasks. *IEEE Trans. Robotics* 25, 670–685. doi:10.1109/tro.2009.2020345

Martin, R. C. (2003). *Agile software development principles, patterns, and practices*. Upper Saddle Hill: Prentice Hall.

Meyer, B. (1997). Object-oriented software construction

Orleans, D., and Lieberherr, K. (2001). "DJ: dynamic adaptive programming in Java," in *Metalevel architectures and separation of crosscutting concerns*.

Parr, T. (2013). *The definitive ANTLR 4 reference* (pragmatic bookshelf).

Parr, T. J. (2004). "Enforcing strict model-view separation in template engines," in *Proc. International conference on world wide Web (WWW)*.

Pearl, J. (1982). "Reverend Bayes on inference engines: a distributed hierarchical approach," in *Proc. International joint conference on artificial intelligence*.

Popov, E. P., Vereshchagin, A. F., and Zenkevich, S. L. (1978). *Manipuljacionnyje roboty: dinamika i algoritmy*. Moscow Nauka.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., et al. (2009). "ROS: an open-source robot operating system," in *IEEE international conference on robotics and automation (ICRA). Workshop on open source software*.

Rodriguez, M. A. (2015). "The Gremlin graph traversal machine and language," in *Proc. Of the ACM database programming languages conference (DBPL)*.

Schneider, S., and Bruyninckx, H. (2019). "Exploiting linearity in dynamics solvers for the design of composable robotic manipulation architectures," in *Proc. IEEE/RSJ international conference on intelligent robots and systems (IROS)*.

Schneider, S., Hochgeschwender, N., and Bruyninckx, H. (2023). "Domain-specific languages for kinematic chains and their solver algorithms: lessons learned for composable models," in *Proc. IEEE international conference on robotics and automation (ICRA)*.

Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Champin, P.-A., and Lindström, N. (2020). JSON-LD 1.1. A JSON-based serialization for linked data. W3C recommendation. *World Wide Web Consort. (W3C)*. Available at: https://www.w3.org/TR/json-ld/.

Tanenbaum, A. S., and Bos, H. (2014). *Modern operating systems (peason PLC)*. fourth edn.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., et al. 2017). "Attention is all you need," in *Proc. International conference on neural information processing systems (NIPS)*.

Vereshchagin, A. F. (1989). Modelling and control of motion of manipulational Robots. *Soviet J. Comput. Syst. Sci.*, 125–134. Originally published in Izvestiia Akademii nauk SSSR, Tekhnicheskaya Kibernetika, No. 1.

frontiers | Frontiers in Robotics and AI

TYPE Original Research
PUBLISHED 02 April 2025
DOI 10.3389/frobt.2025.1548250

Check for updates

OPEN ACCESS

EDITED BY
Lorenzo Carnevale,
University of Messina, Italy

REVIEWED BY
Jonathan M. Aitken,
The University of Sheffield, United Kingdom
Roberto Marino,
University of Messina, Italy

*CORRESPONDENCE
Michel Albonico,
michelalbonico@utfpr.edu.br

RECEIVED 19 December 2024
ACCEPTED 29 January 2025
PUBLISHED 02 April 2025

CITATION
Albonico M, Cannizza MB and Wortmann A (2025) Energy efficiency in ROS communication: a comparison across programming languages and workloads. *Front. Robot. AI* 12:1548250. doi: 10.3389/frobt.2025.1548250

# Energy efficiency in ROS communication: a comparison across programming languages and workloads

Michel Albonico[1]*, Manuela Bechara Cannizza[1] and Andreas Wortmann[2]

[1]IntelAgir Research Group, Informatics Department, Federal University of Technology, Paraná (UTFPR), Francisco Beltrão, Brazil, [2]Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW), University of Stuttgart, Stuttgart, Germany

**Introduction:** The Robot Operating System (ROS) is a widely used framework for robotic software development, providing robust client libraries for both C++ and Python. These languages, with their differing levels of abstraction, exhibit distinct resource usage patterns, including power and energy consumption–an increasingly critical quality metric in robotics.

**Methods:** In this study, we evaluate the energy efficiency of ROS two nodes implemented in C++ and Python, focusing on the primary ROS communication paradigms: topics, services, and actions. Through a series of empirical experiments, with programming language, message interval, and number of clients as independent variables, we analyze the impact on energy efficiency across implementations of the three paradigms.

**Results:** Our data analysis demonstrates that Python consistently demands more computational resources, leading to higher power consumption compared to C++. Furthermore, we find that message frequency is a highly influential factor, while the number of clients has a more variable and less significant effect on resource usage, despite revealing unexpected architectural behaviors of underlying programming and communication layers.

KEYWORDS

**ROS, energy efficiency, programming language, ROS communication, robotic**

## 1 Introduction

Robots play an important role in many areas of our society. They are commonly used in manufacturing, medicine, transportation (including self-driving vehicles), and as domestic allies (e.g., vacuum cleaners) (Ciccozzi et al., 2017). A great part of those robots depends on increasingly complex software, for which the Robot Operating System (ROS) (Stanford Artificial Intelligence Laboratory, 2024; Steve, 2011) is one of the most important frameworks.

ROS is considered the *de facto* standard for robotic systems in both, research and industry (Koubaa, 2015). It provides an abstraction layer that enables specialists from different areas to integrate their software into one robotic system. In addition, ROS comprises a comprehensive set of open-source libraries and packages. With over half a billion ROS packages downloaded in 2020, it has also significantly encouraged code reuse (Stanford Artificial Intelligence Laboratory et al., 2024). ROS currently has two main versions, ROS one and ROS 2, with end-of-life of ROS 1 being set to 2025. In this paper, we focus only on ROS 2, the only supported distribution in the near future, using ROS as nomenclature.

Software energy efficiency has been a recurrent concern among software developers (Pinto and Castor, 2017). This is stimulated by factors that include environmental impact, budget, and battery-dependent devices (Steve, 2011; Swanborn and Malavolta, 2020), which also applies to the robotic domain. Simple software architectural decisions can make an impact on the energy efficiency of robotic software (Chinnappan et al., 2021), where the programming language is known to be a determinant factor (Pereira et al., 2017; Albonico et al., 2024). In the case of ROS, C++, and Python are the two main programming languages thoroughly supported and documented by the community. Therefore, practitioners tend to start by choosing one of them, which currently must be done with a limited understanding of their impact on ROS systems' energy efficiency.

In **this paper**, we conduct a systematic analysis of the energy consumption associated with message exchanges among ROS nodes implemented in C++ and Python. This study builds upon our previous work (Albonico et al., 2024), which presented initial findings on the energy impact of implementing ROS nodes in different programming languages, motivating further investigation. In that study, we observed two key challenges: (i) Python nodes exhibited higher resource usage, resulting in reduced energy efficiency, and (ii) high message frequencies constrained scalability across multiple nodes. However, the experiments were limited in scope, with only a few independent variables, which were randomly defined. To address these limitations, this paper extends the investigation by exploring four independent variables: (i) the programming language of the ROS nodes; (ii) the ROS communication pattern[1] (e.g., topic, service, or action); (iii) the frequency of message exchange; and (iv) the number of clients/subscribers per server/publisher. Each algorithm in the study is adapted from concrete examples on the ROS tutorials Wiki page[2], carefully adapted for this study. The experimental **results** revealed the programming language and message frequency as consistent key factors influencing energy efficiency across different ROS communication patterns. Additionally, the number of clients had an impact on power consumption, particularly for server/publisher nodes, although to a lesser degree. Interestingly, increasing the number of clients/subscribers sometimes resulted in unexpected behaviors, such as reduced power consumption in client nodes. This observation raises important questions that foster further investigation.

The **target audience** for this study includes researchers and practitioners involved in developing ROS-based systems. This work provides valuable insights to help optimize ROS systems, make informed design decisions, and conduct experiments in energy-efficient robotic systems. It encourages researchers to focus their further studies, which may consider other ROS architectural models, such as multi-node composition within single processes. Additionally, it supports practitioners in selecting suitable programming languages for their specific robotics projects, thereby contributing to the development of greener robotic software.

This paper **contributes** with insights into the energy consumption and power of ROS nodes communication across different paradigms and programming languages. It can used as a source of inspiration for developing greener robotic software, promoting environmentally conscious practices in robotic software development. It also provides a methodological framework and practical guidance for conducting further experiments in this field. Additionally, we provide a complete replication package and experimental data to benefit both researchers and practitioners. Finally, despite the relevant energy-related results, some combination of independent variables resulted in unexpected behaviors that must be shared with ROS community and can lead to important improvements ROS 2 software layers.

## 2 Background

This section presents the fundamental concepts of ROS, its communication and programming premises, and discusses Running Average Power Limit (RAPL)[3] for energy consumption measurements.

### 2.1 Robot operating system (ROS)

ROS is a standard robotics framework in both, industry and research, for the effective development and building of a robot system (Santos et al., 2016). Currently, there are many distributions of ROS available, grouped into two main versions (ROS 1 and ROS 2). ROS 2 completely changed the architecture compared to the first version, which now massively relies on the decentralized Data Distribution Service (DDS) (Pardo-Castellote, 2003).

A ROS software architecture consists of four main types: *nodes*, *topics*, *actions*, and *services*. *Nodes* are executable processes, usually implementing a well-defined functionality of a ROS system, which can communicate asynchronously or synchronously. Asynchronous communication relies on the *publisher/subscriber* pattern, while asynchronous communication can be implemented over *services* or *actions*. All three communication patterns are presented in the sequence.

Figure 1 depicts a Unified Modeling Language (UML) sequence diagram that represents *publisher/subscriber*-based ROS communication. In this communication model, the *Publisher* sends messages to a *topic*, and the *ROS Middleware* routes these messages to subscribed nodes (represented by the *Subscriber* component). This is a unidirectional flow commonly used for continuous data streams, such as sensor data[4].

Figure 2 depicts a Unified Modeling Language (UML) sequence diagram that represents *service*-based ROS communication. The

1 https://wiki.ros.org/ROS/Patterns/Communication

2 https://docs.ros.org/en/galactic/Tutorials.html

3 https://greencompute.uk/Measurement/RAPL

4 https://answers.ros.org/question/295426/why-is-pubsub-the-ideal-communication-pattern-for-ros-or-robots-in-general-instead-of-requestresponse/

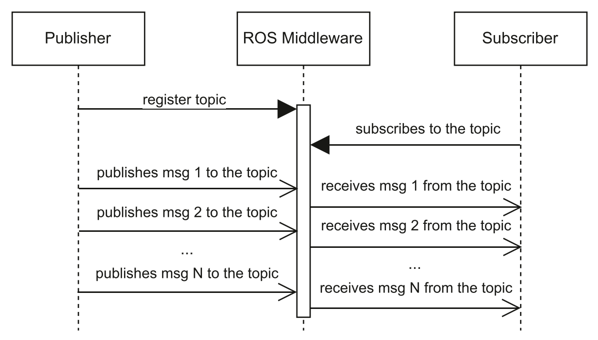

Publisher
ROS Middleware
Subscriber
register topic
subscribes to the topic
publishes msg 1 to the topic
receives msg 1 from the topic
publishes msg 2 to the topic
receives msg 2 from the topic
...
publishes msg N to the topic
...
receives msg N from the topic

FIGURE 1
UML sequence diagram for *publisher/subscriber* communication.

diagram captures the synchronous nature of services, where a *client* sends a one-time request to the *server* via the *ROS Middleware*. The *server* processes the request and sends the result back to the *client*. This direct kind of interaction makes services suitable for operations that trigger specific robotic actions, such as manipulating an object with a gripper, which requires a synchronous response to identify whether the operation was successful or not.

Figure 3 depicts a Unified Modeling Language (UML) sequence diagram that represents *action*-based communication in ROS. The diagram features two primary components: the *action client* and the *action server*. The *client* sends a *goal* task to the *server*, which optionally accepts it. Once the goal is accepted, the *client* requests the *result* of the task. While the task is in progress, the *server* can send periodic *feedback* to the *client*, providing updates on the task's status. When the task is completed, the *server* sends the final *result* to the *client*. This interaction can be applied in navigation scenarios, where a navigation goal is sent to the robot. During the navigation process, the robot provides status updates, and once the task concludes, it notifies whether the goal was reached or the task failed.

#### 2.1.1 ROS programming

ROS is recognized for its flexibility in supporting multiple programming languages, allowing developers to choose the language that best suits their needs. As depicted in Figure 4, all client libraries in ROS share the same underlying software layers. From a bottom-up perspective, this architecture begins with the communication *middleware* and *rmw adapter* (ROS Middleware Adapter), which together enable the use of various middleware solutions without requiring modifications to ROS 2 itself. Above the *rmw adapter*, the *rmw* layer serves as an interface between the lower and upper layers. At the top of this stack, the `rcl` layer provides a high-level API for programming ROS applications. Finally, language-oriented libraries, such as *rclcpp*[5] and *rclpy*[6], lie over the *rcl* layer, enabling developers to create ROS 2 algorithms in their chosen language.

### 2.2 Running average power limit

Modern processors provide a Running Average Power Limit (RAPL) interface for power management, which reports the processor's accumulated energy consumption, and allows the operating system to dynamically keep the processor within its limits of thermal design power (TPD)[8]. RAPL is a recurrent profiling tool in previous related work (Zhang and Hoffman, 2015; Hähnel et al., 2012; Khan et al., 2018; von Kistowski et al., 2016). It keeps counters that can provide power consumption data for both, processor and primary memory. CPU is proven to be one of the most energy-consuming parts of a computer system (Hirao et al., 2005; von Kistowski et al., 2016; Pereira et al., 2017). Despite the primary memory usage not being a usual determinant factor in other studies (von Kistowski et al., 2016; Pereira et al., 2017), it is one of the main RAPL metrics and in this work will be used to determine whether it is still the case for ROS programming.

There are different RAPL-based energy profilers publicly available, among which PowerJoular (Noureddine, 2022) stands out. It has been recurrent in energy-efficiency studies in the literature (Thangadurai et al., 2024; Noureddine, 2024; Yuan et al., 2024). PowerJoular offers real-time insights into energy consumption patterns across diverse hardware components, such as CPUs, GPUs, and memory subsystems. Additionally, it facilitates granular energy measurements of running processes, enabling precise analysis of the energy consumption of individual ROS 2 components.

## 3 Experiment definition

The experiment of this paper is defined after the Goal Question Metric (GQM) model (Basili et al., 1994). It starts with a well-defined *goal*, which is then refined into *research questions* that are answered by measuring the software system using objective and/or subjective *metrics*.

### 3.1 Study goal

This study ***goal*** is to ***analyze*** *ROS programming with C++ and python languages* ***for the purpose*** *of understanding the extent* ***with respect to*** *energy efficiency* ***from the point of view*** *robotics researchers and practitioners* ***in the context of ROS*** *nodes communication patterns.*

5 https://github.com/ros2/rclcpp

6 https://github.com/ros2/rclpy

7 https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html

8 https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html

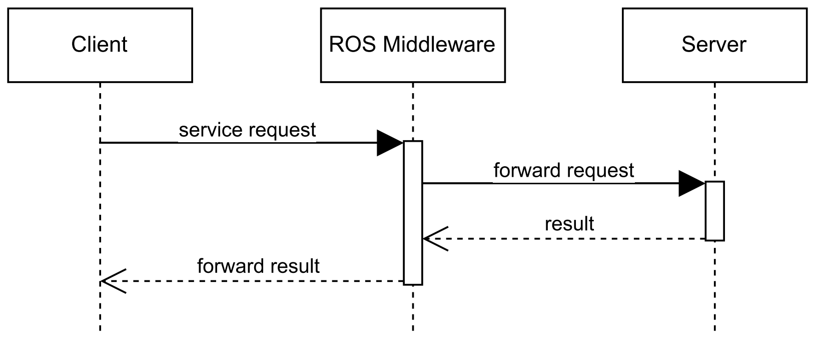


FIGURE 2
UML sequence diagram for *service* communication.

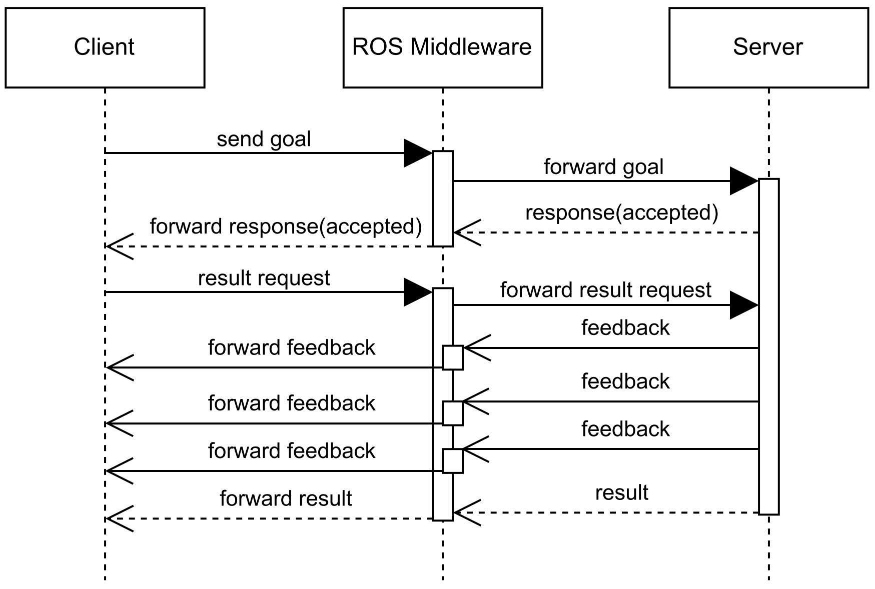


FIGURE 3
UML sequence diagram for *action* communication.

## 3.2 Questions

From our goal, we derive the following research questions.

- RQ1: How is the energy efficiency of each ROS communication pattern?

In ROS, the asynchronous pattern implemented through *topics* is a common and straightforward method for message exchange among nodes. However, the other two synchronous patterns, *service* and *action*, provide essential features enabling advanced synchronization and reliability. Since synchronous communication patterns rely on session-oriented connections, they are expected to consume more computational resources. However, the impact of these design choices on the energy efficiency of ROS systems remains unexplored.

- RQ2: How do the `C++` and `Python` implementations affect resource usage and energy consumption when handling different communication patterns among ROS nodes?

The motivation for this research question is that each language depends on its canonical client library, i.e., `rclcpp` and `rclpy`. Despite those libraries being developed following the same design principles, both languages have distinct concepts, such as compiled vs. interpreted, multi-threading management, abstraction level, etc., that may lead to particular implementations, and impact resource usage and energy consumption.

- RQ3: How does language efficiency scale over different frequencies of communication and number of clients?

Since communication is largely managed by lower-level layers, such as the DDS (see Figure 4), the efficiency differences between

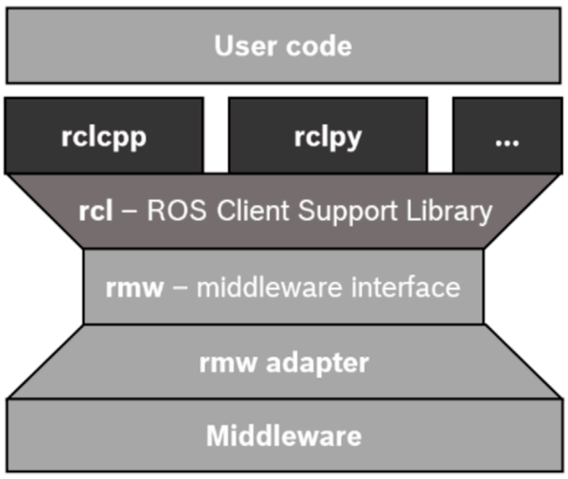


FIGURE 4
Underlying layers of a ROS node programming[7].

languages in simple examples may be minimal. However, message packing and unpacking are processed locally, which can impact both, resource usage and energy efficiency. Additionally, the number of clients can trigger multi-threading, a feature worth investigating, particularly given Python's limitations. Python native multi-threading is limited by its Global Interpreter Lock (GIL)[9], so achieving full parallelism often requires external libraries.

## 3.3 Metrics

Table 1 describes the metrics used for measurements during the experiments. *Energy consumption*, *power* and *execution time* are the key metrics used to assess the energy efficiency of a ROS node, while *CPU* and *memory usage* are metrics that help us to understand how intensive is the ROS node in terms of computational processing, and then reason about the measured *energy efficiency*.

All the measurements refer to the ROS node operating system process. The energy consumption measurements take into account the two main processing factors: CPU and memory. After the energy consumption is measured, we calculate the *power* with the following formula: $P = \frac{EC}{t}$, where $P$ is power, $EC$ is energy consumption, and $t$ is the total ROS node execution time in seconds. Power measurements help identify transient effects that energy consumption (a cumulative metric) might mask. We give more details of the measurement process and tools in Section 5.3.

## 4 Experiment planning

The experiment depends on six algorithms that cover the three ROS nodes' communication patterns (i.e., *topics*, *services*, and *actions*), implemented in both languages, Python and C++. The algorithms are based on ROS Tutorials Wiki pages[10], which provide concise examples. They are all independent from a physical robot, seeking full controllability during the experiments.

### 4.1 ROS 2 algorithms

Table 2 depicts the six algorithms, with a short description, details of their implementation, their dependencies, and their complexities (i.e., logical lines of code–LLOC, and the algorithm McCabe's cyclomatic complexity–MCC), the last two, for a matter of illustration of the compatibility between Python and C++ algorithm implementations. For the implementation, we began with the Python version of each algorithm, as it is the language we are most familiar with. Subsequently, we used the ChatGPT tool[11] (GPT-4o version) to generate compatible C++ versions, which we manually reviewed to ensure compatibility and correctness.

It is evident that the C++ implementations resulted in greater LLOC, particularly for the *action server* and *action client*, where the difference compared to Python nearly doubled, as highlighted in red. It is important to note that exact equality is not possible due to the inherent differences between the languages. Despite variations in code size, all the algorithms exhibit similar complexity (and exactly the same for *service server*, as highlighted in blue), reflecting their overall similarity. The larger difference observed in the size and complexity of *action client* implementation is due to that node being folded into two *services* (one for sending the task and another for retrieving the result) as well as *topic* communication (for receiving task feedback). The size difference is compatible with the other algorithms if we consider the sum of the difference between the *service client* and *subscriber*, for example,. The complexity is similar to the *service client*, and could not be reduced due to the complexity of synchronizing the actions' execution callbacks in C++. Furthermore, all the algorithms lie in the complexity range 1–10 which classifies them as simple (Thomas, 2008).

The table presents the algorithms in pairs, as they execute in the experiments (see Section 5.2). Algorithms 1 and 2 implement the *publisher* and *subscriber* pair, which enable the *publisher* to exchange different message types with the *subscriber* over a specific *topic*. Algorithms 3 and 4 implement the *service server* and *service client* pair, where the *service client* requests the *service server* to do a simple calculation of adding two integer numbers and receives its response. Algorithms 5 and 6 implement the *action server* and *action client* pair, where the *action client* sends a *task* to the *action server* (i.e., to calculate a Fibonacci sequence) via a *service goal*, receives each value of the sequence via a *feedback topic*, and at the end, receives the notification of the task completeness via a *result service*.

### 4.2 Experiment variables

Table 3 summarizes the variables used in the experiments. It categorizes the variables into three main groups: *independent*, *static*, and *dependent* variables.

9 https://realpython.com/python-gil/

10 http://wiki.ros.org/ROS/Tutorials

11 https://chatgpt.com/

TABLE 1 Experiment metrics.

| Metric | Unit | Description |
|---|---|---|
| Energy Consumption | `Joules (J)` | Amount of energy necessary to run the ROS node |
| Power | `Watts (W)` | Energy consumption rate when running the ROS node |
| Execution time | `Milliseconds (ms)` | Total time spent to run a ROS node |
| CPU usage | `Percentage (%)` | Average CPU percentage used during a ROS node execution |
| Memory usage | `Kilobytes (KB)` | Amount of memory used during a ROS node execution |

TABLE 2 ROS2 algorithms subject of investigation with their dependencies and complexity measurements.

| Node | Description | Main dependencies | LLOC python | LLOC C++ | MCC python | MCC C++ |
|---|---|---|---|---|---|---|
| *1.Publisher* | ROS node that continuously sends messages to a *topic* | `rclpy/rclcpp, std_msgs` | 45 | 58 | 2.3 | 1.7 |
| *2.Subscriber* | ROS node that subscribes to the *topic* and reads the published messages | `rclpy/rclcpp, std_msgs` | 52 | 67 | 2.2 | 1.8 |
| *3.Service Server* | ROS node that provides a service | `rclpy/rclcpp, example_interfaces` | 40 | 67 | 1.8 | 1.8 |
| *4.Service Client* | ROS node that consumes the *server* service | `rclpy/rclcpp, example_interfaces` | 52 | 82 | 2.2 | 3.5 |
| *5.Action Server* | ROS node that receives a goal and returns its lifetime state feedback | `rclpy/rclcpp, action_tutorials_ interfaces` | 53 | 85 | 2 | 1.3 |
| *6.Action Client* | ROS node that sends the goal to the *server* | `rclpy/rclcpp, action_tutorials_ interfaces` | 36 | 96 | 1.2 | 3.2 |

1. *Independent variables*: these are the variables that we control during the experiment. They include the *ROS algorithm pair*, which refers to the specific pairs of algorithms that implement different communication patterns in ROS; the *message interval*, which defines the time gap between message exchanges; the *number of clients*, which specifies how many subscribers or clients are interacting with the server or publisher; and the *programming language* used to implement the ROS algorithms.
2. *Static variables*: these are the variables that do not change during the experiments. They include the *ROS distribution* in the Docker containers (where the ROS algorithms run), and the *environment setup*, which refers to the computer and Docker setup for the experiments.
3. *Dependent variables*: these are the measurements during experiment execution, which use the metrics previously described in Table 1. They include *CPU usage*, *memory usage*, and *energy consumption*. They reflect the system's performance in terms of resource utilization, providing insight into how different algorithm pairs and configurations affect the overall efficiency of the system.

Table 4 presents the values of the experimental variables. The *pairs of algorithms* and the *programming languages* have been discussed previously. The *message interval* ranges from 0.05 (20 messages per second) to 1.0 (1 message per second). These intervals have been selected with real-world applications in mind, where critical robotic tasks such as navigation and telemetry typically require short intervals (e.g., the *joystick* package by default relies on 20 messages per second[12]). In contrast, less time-sensitive applications, such as monitoring systems, can tolerate moderate rates (0.5–1.0 s). Longer intervals, which might be suitable for logging applications, are not considered as extended message intervals tend to lead to inexpressive resource usage. The *number of clients* increases gradually from 1 to 3, a range that is realistic for small to medium-sized robotic applications on GitHub[13]. This range also allows us to get insights into how the algorithm's efficiency scales.

12 https://index.ros.org/p/joy/

13 https://github.com/IntelAgir-Research-Group/frontiers-robotics-ai-rep-pkg/blob/main/data-analysis/repos.csv

TABLE 3 Experiment variables.

| Type | Name | Category | Description |
|---|---|---|---|
| *Independent variables* | ROS Algorithm Pair | Nominal | The pairs of algorithms subject to this study, which implement the different ROS communication patterns |
| | Message Interval | Ratio | The interval between each message exchange |
| | Number of Clients | Ratio | The number of *clients/subscribers* for each *server/publisher* |
| | Programming Language | Nominal | The programming language used to implement the ROS algorithm pairs |
| *Static variables* | ROS distribution | Nominal | ROS distribution on the Docker containers used for running the experiments |
| | Environment Setup | Nominal | The computer machine and Docker environment where the experiments are run |
| *Dependent variables* | CPU usage | Ratio | Average percentage of CPU usage during a experiment run |
| | Memory usage | Ratio | Average percentage of memory usage during a experiment run |
| | Energy consumption | Ordinal | Average energy consumption during a experiment run |

TABLE 4 Independent variable values.

| Variable name | Values |
|---|---|
| ROS Algorithm Pair | [*publisher, subscriber*], [*service server, service client*], [*action server, action client*] |
| Message Interval (s) | 0.05, 0.1, 0.2, 0.5, 1.0 |
| Number of Clients | 1, 2, 3 |
| Programming Language | *Python, C++* |

The factors of this study are the four independent variables, with two–three values each, where the number of treatments can be calculated as following:

$$\text{Number of Treatments} = \prod_{i=1}^{k} L_i = L_1 \times L_2 \times \cdots \times L_k$$

where:

$$L_1 = 3\,(\text{ROS Algorithm Pair}),\ L_2 = 5\,(\text{Message Interval})$$
$$L_3 = 3\,(\text{Number of Clients}),\ L_4 = 2\,(\text{Programming Language})$$

Thus, the total number of treatments is:

$$3 \times 5 \times 3 \times 2 = 90$$

The treatments are repeated multiple times (see Section 5.2) to enable statistical inference from the measurements. We provide additional details regarding the experiment execution in the following section.

## 5 Experiment execution

In this section, we define hardware and software components used in the experiments and detail how the algorithms are orchestrated. For a matter of transparency and reuse, we also provide a public replication package[14].

### 5.1 Instrumentation

Figure 5 illustrates the deployment of the experimental artifacts on a single desktop computer with the following specifications: Linux Ubuntu 22.04 operating system, kernel version 6.2.0–33-generic, 20 GB of RAM, and an Intel(R) Core(TM) i5-10210U CPU at 1.60 GHz. Each algorithm was implemented as a ROS 2 node using the ROS Humble distribution[15], which has an end-of-life (EOL) date in May 2027, and distributed as part of a single ROS 2 package with all the implementations. In the experiments, each ROS node runs in a separate Docker (version 24.0.7) container. All the procedures are inside the node's callback functions, so ROS can spin them, taking care of underlying threading[16]. Algorithm executions are orchestrated by the `ros2 run` command, which speeds up automation and guarantees the same underlying layers for every execution.

To eliminate concurrency, all experiments were conducted on a dedicated machine, ensuring no other end-user applications were running. The operating system's power-saving mode was set to `performance`, ensuring unrestricted power usage. This configuration was crucial to maintain a controlled environment, providing consistent priority for each execution. Additionally, we assigned a priority level of 0 (the highest as non-root) to the processes corresponding to the algorithms under experimentation, granting them priority access to the machine's resources. Between each experiment, a 30-s interval was observed to allow the

14 https://github.com/IntelAgir-Research-Group/frontiers-robotics-ai-rep-pkg

15 https://docs.ros.org/en/humble/index.html

16 http://wiki.ros.org/roscpp/Overview/Callbacks%20and%20Spinning

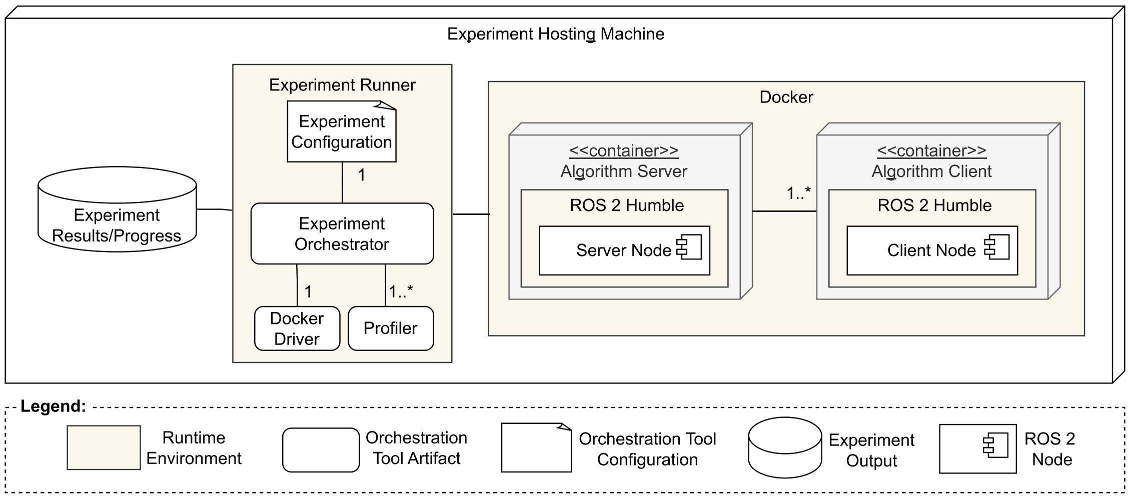


FIGURE 5
Adapted UML deployment diagram of experiment instrumentation.

machine to cool down, which by experimental observation is enough waiting time for the CPU to return to its baseline usage percentage.

## 5.2 Algorithms execution

The algorithms are implemented in pairs, as shown in Table 2, consisting of a *publisher/server* and a *subscriber/client*. Each pair executes repeatedly according to the defined *message interval* until reaching a total run-time of 3 min. The total run-time has been carefully chosen so the ROS nodes have time to capture transient effects like initialization overhead, start-up energy spikes, and system state changes, and to average out possible transitional background processes that may insert noise to the measurements. It also makes the experiment repetitions be completed in a couple of days and enables enough data points for statistical analysis. When multiple *subscriber/clients* are present, each performs the same communication with the *publisher/server* in parallel. Each round typically takes ≈4.5 minutes on average to complete, where a complete round of the 90 treatments takes ≈405 minutes (or 6.75 h). To ensure statistical significance, each treatment is repeated 20 times, resulting in an overall execution time of ≈135 hours (≈5.6 days).

- For nodes 1 and 2 (cf. Table 2), the *publisher* continuously sends a preset message to the *subscriber* at the specified interval. To avoid messages to be lost in high frequency, the *subscriber* is set with a message querying of 10.
- For nodes 3 and 4, the *service client* establishes a connection with the *service server*, uses its service (e.g., performing a calculation with two integers), and receives the result. The connection remains active throughout the experiment to focus on evaluating communication exchanges.
- For nodes 5 and 6, the *action client* connects to the *action server* once at the start of the experiment. It continuously sends goals (e.g., calculating a Fibonacci sequence), receives intermediary feedback, and obtains the final sequence as the result.

## 5.3 Resource profiling

We profile energy consumption using *PowerJoular* (Noureddine, 2022), which leverages Intel's Running Average Power Limit Energy Reporting (RAPL)[17], measuring both, CPU utilization and energy consumption. PowerJournal is an energy monitoring tool that leverages the RAPL interface available in Intel processors to measure power consumption. RAPL provides energy estimations at different levels, such as the package (CPU socket) and the DRAM. These estimations are derived from internal processor models rather than direct physical measurements but have been shown to be accurate for comparative energy consumption analysis. PowerJournal interacts with RAPL via the powercap framework in Linux, periodically reading energy counters exposed through*/sys/class/powercap*. This allows us to measure energy consumption at fine-grained intervals with minimal overhead. For the experiments, *PowerJoular* is configured to monitor the energy usage of each ROS node's processes individually, capturing data at a fixed rate of one measurement per second (with no option to increase the frequency). To gather more granular data on memory usage and CPU utilization, we developed a customized Python script using the *psutil*[18] library. This script records measurements at a rate of 10 samples per second.

17 https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html
18 https://pypi.org/project/psutil/

## 5.4 Data analysis

We begin the data analysis by visually exploring the distribution of power consumption across different combinations of experimental factors: the programming languages `Python` and `C++`, message exchange frequency, and the number of clients. After examining the visual data representation, we proceed with a rigorous statistical testing strategy to assess and validate the primary interpretations. In the following section, we detail the statistical testing approach applied to our data, which can be replicated via replication package[19].

### 5.4.1 Statistical tests

The process of statistical testing starts with an evaluation of key assumptions necessary for parametric tests, such as the distribution of the data and the equality of variances across groups. This approach involves four phases, each dedicated to confirming these assumptions and determining the most appropriate test.

#### 5.4.1.1 Normality assessment

The first step is to verify whether the data follows a normal distribution, as many parametric tests, including ANOVA (St and Svante, 1989), rely on this assumption. To assess normality, we use the Shapiro-Wilk test (Shapiro and Wilk, 1965), which is particularly effective for small sample sizes. If the data does not meet normality, we apply Box-Cox transformations (George and Cox, 1964) to adjust it. Once normality is nearly achieved, we proceed with detecting and removing outliers using the Interquartile Range (IQR) method. We performed a *post hoc* analysis to assess the impact of outlier removal, and observed that this step removes only extreme values (less than 5%), where a representative part of the core dataset is still available for statistical tests.

#### 5.4.1.2 Homogeneity of variance evaluation

Next, we examine the assumption of equal variances across groups, another important condition for tests like ANOVA. Ensuring that the variances within groups are similar allows for valid comparisons. Levene's test is applied here, as it is still consistent even with violations of normality.

#### 5.4.1.3 Parametric and non-parametric testing

When both normality and homogeneity of variance are satisfied, we proceed with the one-way ANOVA to test for differences in means across groups. If the analysis involves just two groups, the t-test is applied instead. With the violation of any of the assumptions, we rely on non-parametric alternatives, such as Welch's ANOVA (Bernard, 1951) and the Kruskal–Wallis test (Kruskal and Wallis, 1952), which do not require normality or equal variances.

#### 5.4.1.4 Post-hoc analysis

If statistical test results suggest significant group differences, *post hoc* analysis is conducted to pinpoint where the differences occur. For parametric tests, we rely on Tukey's Honestly Significant Difference (HSD) test (Abdi and Williams, 2010), as it accounts for multiple comparisons, reducing the risk of false positives. In cases where non-parametric tests were used, we rely on Dunn's test (Olive Jean Dunn, 1961), which offers robustness in the face of normality violations.

19 https://github.com/IntelAgir-Research-Group/frontiers-robotics-ai-rep-pkg/tree/main/

# 6 Results

In this section, we present the key results of the three studied communication patterns and provide a concise discussion of the observed data based on statistical tests. Finally, we compare the measurements across the different communication patterns. All the results presented in this section, have been carefully and manually inspected, and the runs that result in unexpected measurements have all been confirmed by re-execution.

## 6.1 *Publisher* and *subscriber*

We begin the analysis with the *publisher* and *subscriber* data, first describing the mean/total values of each measurement across different configurations. Next, we illustrate the primary data distributions, followed by the presentation of the statistical testing results.

### 6.1.1 Publisher

Table 5 summarizes the measurements of the *publisher* node across the experiments. Across all metrics, C++ demonstrates consistently superior efficiency than Python, particularly in power/energy consumption and CPU utilization (both being directly related). Python exhibits higher resource overhead, especially at high message frequencies of 0.05 and 0.1 s. Memory usage for both remains stable over different configurations, with Python resulting at approximately 41,000 KB on average, nearly double that of C++, which averages around 21,000 KB. The little memory variation across different configurations for both languages is comprehensible since the algorithms remain the same and there is only message replication, with no special pre/post-processing. Increasing the number of clients seems to raise resource consumption for both implementations slightly, and the effect appears to be less significant compared to variations caused by message interval. Furthermore, at the highest frequency (0.05-s message interval), the number of clients does not result in a consistent increasing in power consumption for both languages, despite the grow in CPU usage. However, it is not possible to observer an important decrease either. We carefully investigated the execution logs, and we could not identify any issues. Therefore, we assume this is due to the overhead of such a high-frequency message exchange.

Figure 6 illustrates the distribution of average power consumption for the *publisher* across all repetitions. All figures show C++ with consistently lower power consumption compared to Python. It is also visually evident that shorter message intervals are associated with higher power consumption. Additionally, in most cases, power consumption tends to increase slightly with the number of clients. An exception to this trend is observed at 0.05-s message intervals (Figure 6A, where Python exhibits an anomalous behavior previously highlighted in Table 5, with a slight reduction

TABLE 5 Results of *publisher* with different frequencies and number of clients over 20 executions.

| Language | # Clients | Msg. Interval (s) | Avg. CPU utilization (%) | Avg. Memory utilization (KB) | Total CPU energy (J) | CPU power (W) |
|---|---|---|---|---|---|---|
| *C++* | 1 | 0.05 | 0.59 | 21,009 | 48.82 | 0.27 |
| | | 0.1 | 0.32 | 20,837 | 25.45 | 0.14 |
| | | 0.25 | 0.13 | 20,908 | 11.1 | 0.06 |
| | | 0.5 | 0.07 | 20,862 | 6.19 | 0.03 |
| | | 1.0 | 0.05 | 20,856 | 4.05 | 0.02 |
| | 2 | 0.05 | 0.63 | 21,067 | 52.03 | 0.29 |
| | | 0.1 | 0.33 | 21,041 | 26.48 | 0.15 |
| | | 0.2 | 0.15 | 21,023 | 12.33 | 0.07 |
| | | 0.5 | 0.08 | 20,991 | 6.95 | 0.04 |
| | | 1.0 | 0.06 | 21,003 | 4.52 | 0.03 |
| | 3 | 0.05 | 0.68 | 21,086 | 54.53 | 0.3 |
| | | 0.1 | 0.37 | 20,018 | 29.11 | 0.16 |
| | | 0.2 | 0.16 | 21,100 | 13.58 | 0.08 |
| | | 0.5 | 0.09 | 21,004 | 7.74 | 0.04 |
| | | 1.0 | 0.06 | 20,985 | 4.96 | 0.03 |
| *Python* | 1 | 0.05 | 2.1 | 41,033 | 127.83 | 0.71 |
| | | 0.1 | 1.02 | 41,093 | 61.20 | 0.34 |
| | | 0.2 | 0.45 | 41,026 | 37.5 | 0.21 |
| | | 0.5 | 0.24 | 40,993 | 19.64 | 0.11 |
| | | 1.0 | 0.13 | 40,987 | 11.17 | 0.06 |
| | 2 | 0.05 | 2.25 | 41,174 | 118.80 | 0.66 |
| | | 0.1 | 1.15 | 41,139 | 70.21 | 0.39 |
| | | 0.2 | 0.49 | 41,090 | 39.09 | 0.22 |
| | | 0.5 | 0.26 | 41,080 | 21.19 | 0.12 |
| | | 1.0 | 0.15 | 41,083 | 11.79 | 0.07 |
| | 3 | 0.05 | 2.31 | 41,278 | 122.42 | 0.68 |
| | | 0.1 | 1.28 | 37,131 | 79.23 | 0.44 |
| | | 0.2 | 0.53 | 41,139 | 41.5 | 0.23 |
| | | 0.5 | 0.28 | 41,158 | 22.64 | 0.13 |
| | | 1.0 | 0.16 | 41,164 | 12.66 | 0.07 |

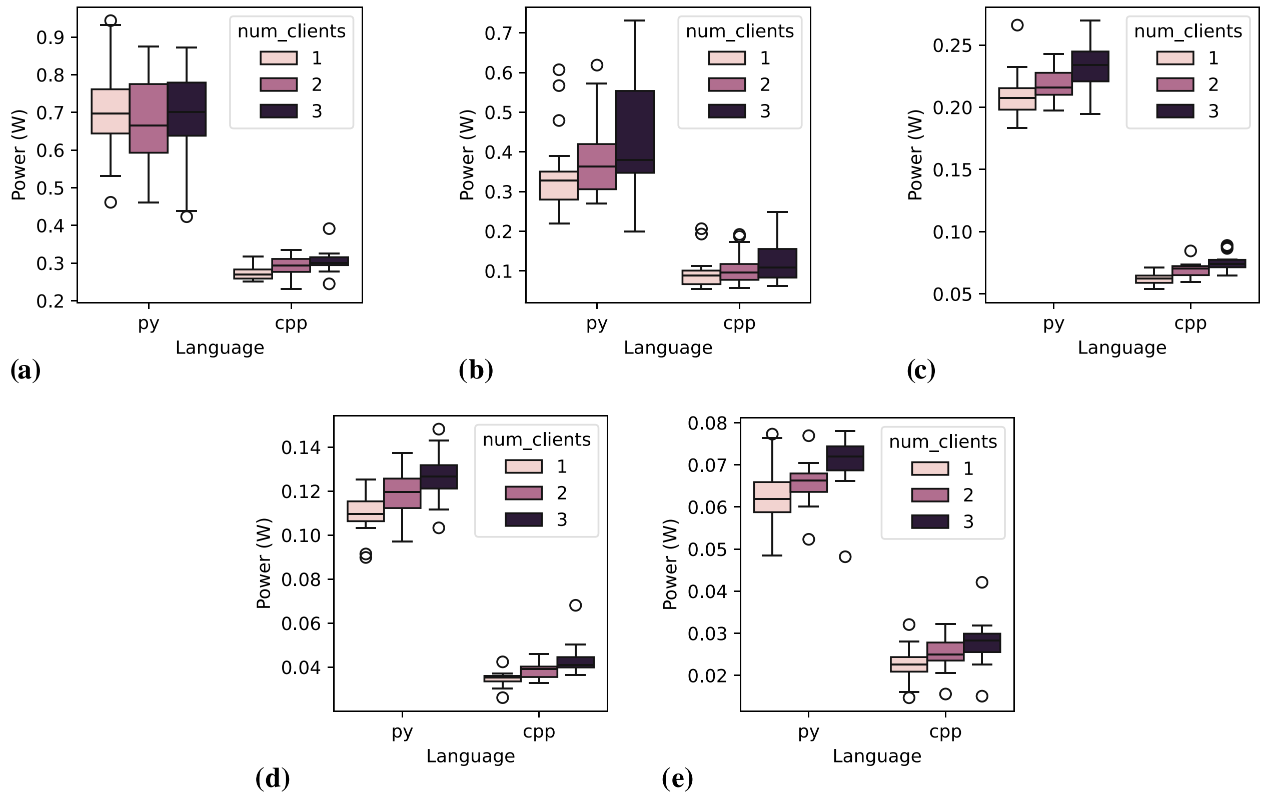


FIGURE 6
Power distribution for the *publisher* node across 20 executions, varying the number of clients and message interval. **(A)** Message interval: 0.05 s, **(B)** Message interval: 0.1 s, **(C)** Message interval: 0.2 s, **(D)** Message interval: 0.5 s, **(E)** Message interval: 1.0 s.

of power consumption with two clients, and then an increase with three clients (which mean value is close to the one with one client).

### 6.1.1.1 Statistical tests

Shapiro-Wilk tests reveal a significant deviation from normality in the data when grouped by a single independent variable. For instance, in the case of the variable language, the test statistic of 0.7147 with a p-value of $3.576 \times 10^{-11}$ falls far below common significance thresholds (e.g., 0.05), strongly rejecting the null hypothesis that the data follows a normal distribution. This pattern is consistent across other independent variables. This outcome aligns with the boxplots in Figure 6, which illustrate a substantial difference between Python and C++ languages. For instance, as shown in Figure 7A, when the data is grouped by a specific message interval of 0.2 s, two distinct clusters of measurements emerge. Figure 7B further reveals that these clusters correspond to groups of measurements based on different programming languages and the number of clients. This is a pattern among groups of other independent variables (what can be inspected in the replication package), where the clusters correspond to the programming languages and, when isolated, appear to follow a normal distribution. These observations strongly suggest that programming language directly influences energy efficiency. This is confirmed by Kruskal–Wallis test on groups by language (non-parametric test since data is not normally distributed), which results in an high $H$ value (205.24) and a p-value ($1.49 \times 10^{-46}$) very distant from the significance threshold. Therefore, for comparative analysis assuming normal distribution, it is essential to group other variables with the programming language.

Considering the programming language as a determining factor, we conduct statistical tests involving message intervals and the number of clients, filtering the data by language (i.e., statistical tests are run for each language separately). We start by testing the effect of message intervals with Kruskal–Wallis test since not every group is normally distributed. The test reveals a significant difference among the groups for both Python ($p = 6.53 \times 10^{-60}$) and C++ ($p = 1.9 \times 10^{-60}$). Table 6 summarizes the results of Dunn's *post hoc* tests for the C++ language across the different message interval groups, with measurements for all numbers of clients. A notable observation is that comparisons between message intervals consistently display increasing differences between groups as the message interval grows. This pattern is also observed for the Python language and for both languages when operating with only one client (which helps avoid bias due to the number of clients). This confirms that the power consumption of the *publisher* node tends to be heavily influenced by the message interval.

Kruskal–Wallis tests on group by programming language and number of clients revealed no significant differences between the groups for Python ($p = 0.36$) and C++ ($p = 0.13$). However, when additionally grouping the data by message intervals, most groups exhibited statistically significant differences. The exceptions were the

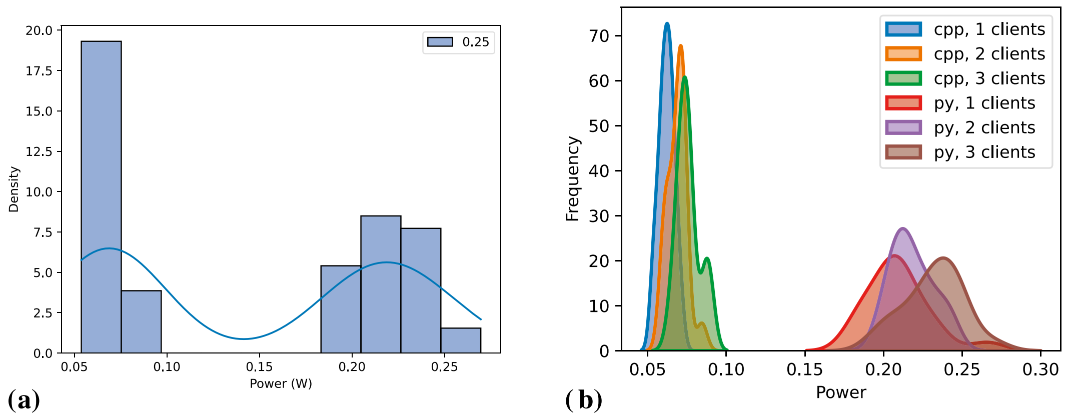


FIGURE 7
Power distribution for the *publisher* node across 20 executions, varying the number of clients at 0.2-s message interval. **(A)** Overall distribution at 0.2 s interval, **(B)** Distribution at 0.2 s interval by number of clients.

**TABLE 6 Dunn's *post hoc* test results for language C++ and different message intervals, with cells in gray representing no significant statistical difference.**

| | 0.05 | 0.10 | 0.2 | 0.50 | 1.00 |
|---|---|---|---|---|---|
| 0.05 | $1.000000\times10^{0}$ | $1.393\times10^{-3}$ | $3.484\times10^{-13}$ | $3.487\times10^{-29}$ | $2.570\times10^{-50}$ |
| 0.10 | $1.393\times10^{-3}$ | $1.000000\times10^{0}$ | $1.636\times10^{-3}$ | $2.820\times10^{-13}$ | $2.067\times10^{-28}$ |
| 0.2 | $3.484\times10^{-13}$ | $1.636\times10^{-3}$ | $1.000000\times10^{0}$ | $1.246\times10^{-3}$ | $6.858\times10^{-13}$ |
| 0.50 | $3.487\times10^{-29}$ | $2.820\times10^{-13}$ | $1.246\times10^{-3}$ | $1.000000\times10^{0}$ | $2.585\times10^{-3}$ |
| 1.00 | $2.570\times10^{-50}$ | $2.067\times10^{-28}$ | $6.858\times10^{-13}$ | $2.585\times10^{-3}$ | $1.000000\times10^{0}$ |

groups corresponding to the Python and C++ languages at a 0.05-s message interval, suggesting an overhead of the *publisher* node.

### 6.1.2 Subscriber

Table 7 presents the performance results for a single ROS-based *subscriber* node across the experiments. Similar to the findings from the *publisher* node analysis, the Python implementation demonstrates higher CPU and memory usage compared to C++, along with greater energy and power consumption. For both languages, resource usage generally decreases with increasing message frequency, although not linearly. Exceptions are also observed at frequencies of 0.05 and 0.1 s, which exhibit an unstable trend consistent with the *publisher* results. Unlike the *publisher*, increasing the number of clients does not significantly impact resource consumption, which is comprehensible since there should be no additional work to be processed as a *subscriber*. However, especially for C++, we observe a slightly increasing pattern as the number of clients increases, which may be the result of extra synchronization work. Memory usage remains stable across all scenarios and aligns closely with the measurements for the *publisher* node.

Figure 8 depicts the distribution of power consumption means for a single *subscriber* across 20 executions. The instability highlighted in Table 7 is evident in Figures 8A, B. In Figures 8C, E, we observe a pattern for C++ where power consumption increases with the addition of a second client but stabilizes with a third *subscriber*. However, the difference appears minor, supporting the assumption that this is caused by overhead in the *publisher* node managing multiple subscribers. Our analysis of the code, including the *rclcpp* library, revealed no explicit synchronization mechanisms in C++ *publisher* handling multiple subscribers. It remains possible that this overhead originates from underlying layers, such as the DDS middleware. For instance, DDS might require additional internal structures and resources to manage the second *subscriber*, resulting in a one-time setup cost with no further increase when adding a third. However, further investigating this potential behavior falls outside the scope of this paper.

#### 6.1.2.1 Statistical tests

The statistical tests reveal that the data distributions closely follow the ones of the *publisher*; however, it tends to be less normally distributed which leads to some different statistical tests across groups of independent variables, as discussed in Section 5.4.1.

For programming languages, the Kruskal–Wallis test results in an $H$ statistic of 205.42 and a *p-value* of $1.37\times10^{-46}$, indicating a significant difference between Python and C++ measurements. This finding aligns with the observations presented in Table 7 and Figure 8. A similar pattern is evident across

TABLE 7 Comparative results of one *subscriber* node with different frequencies and number of clients over 20 executions.

| Language | # Clients | Msg. Interval (s) | Avg. CPU utilization (%) | Avg. Memory utilization (KB) | Total CPU energy (J) | CPU power (W) |
|---|---|---|---|---|---|---|
| *C++* | 1 | 0.05 | 0.57 | 21,291 | 47.4 | 0.26 |
| | | 0.1 | 0.31 | 21,202 | 24.89 | 0.14 |
| | | 0.2 | 0.13 | 21,234 | 10.96 | 0.06 |
| | | 0.5 | 0.07 | 21,183 | 6.16 | 0.03 |
| | | 1.0 | 0.05 | 21,144 | 3.9 | 0.02 |
| | 2 | 0.05 | 0.6 | 21,380 | 49.65 | 0.28 |
| | | 0.1 | 0.33 | 21,355 | 26.09 | 0.15 |
| | | 0.2 | 0.14 | 21,349 | 11.87 | 0.07 |
| | | 0.5 | 0.08 | 21,356 | 6.59 | 0.04 |
| | | 1.0 | 0.05 | 21,432 | 4.45 | 0.02 |
| | 3 | 0.05 | 0.66 | 21,475 | 53.28 | 0.3 |
| | | 0.1 | 0.36 | 21,438 | 28.28 | 0.16 |
| | | 0.2 | 0.15 | 21,388 | 12.81 | 0.07 |
| | | 0.5 | 0.09 | 21,401 | 7.25 | 0.04 |
| | | 1.0 | 0.06 | 21,349 | 4.71 | 0.03 |
| *Python* | 1 | 0.05 | 2.79 | 41,063 | 160.21 | 0.89 |
| | | 0.1 | 1.43 | 41,053 | 90.35 | 0.5 |
| | | 0.2 | 0.6 | 41,023 | 50.08 | 0.28 |
| | | 0.5 | 0.31 | 40,889 | 25.39 | 0.14 |
| | | 1.0 | 0.17 | 40,975 | 14.51 | 0.08 |
| | 2 | 0.05 | 3.08 | 41,120 | 147.60 | 0.82 |
| | | 0.1 | 1.53 | 41,107 | 92.66 | 0.5 |
| | | 0.2 | 0.61 | 40,977 | 48.55 | 0.27 |
| | | 0.5 | 0.32 | 41,014 | 26.25 | 0.15 |
| | | 1.0 | 0.18 | 41,127 | 14.51 | 0.08 |
| | 3 | 0.05 | 3.06 | 41,126 | 154.82 | 0.86 |
| | | 0.1 | 1.7 | 39,193 | 93.21 | 0.52 |
| | | 0.2 | 0.64 | 41,134 | 49.97 | 0.28 |
| | | 0.5 | 0.34 | 41,148 | 27.19 | 0.15 |
| | | 1.0 | 0.19 | 41,080 | 14.78 | 0.08 |

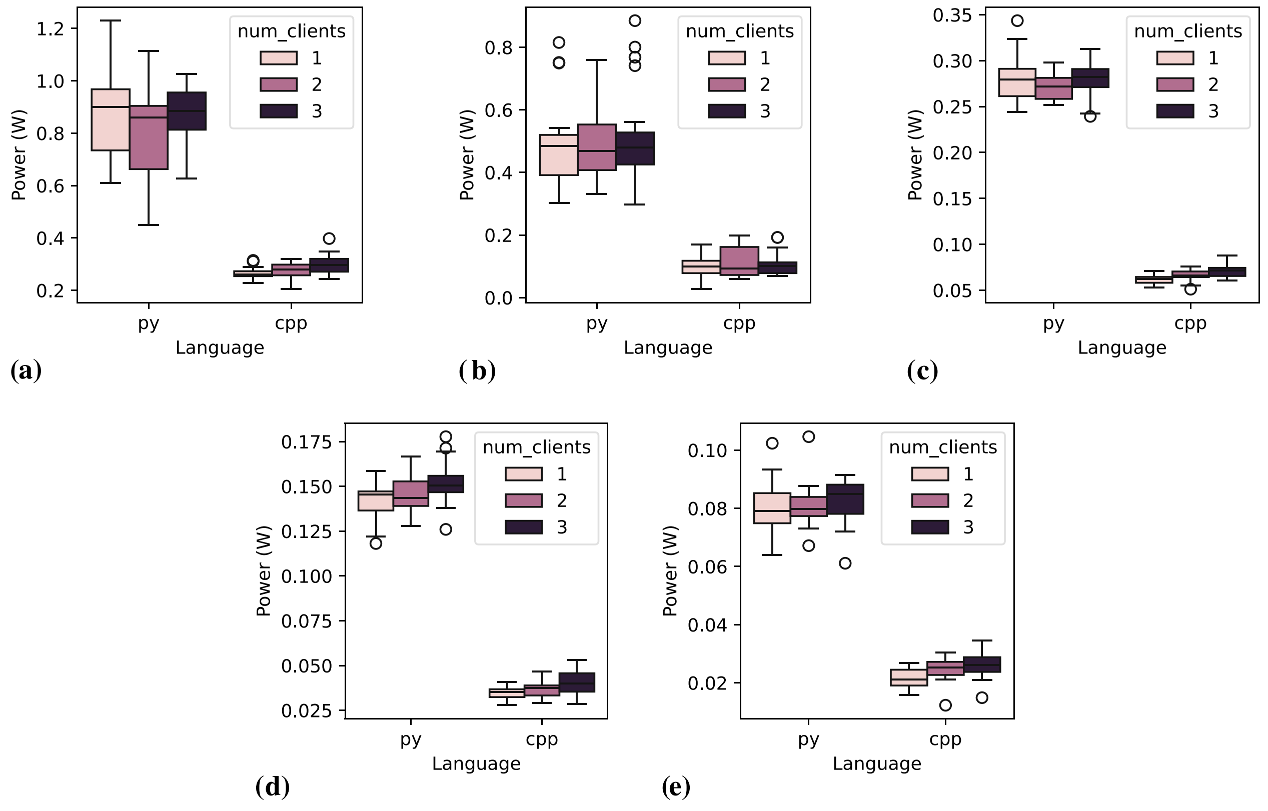


FIGURE 8
Power consumption distribution for one *subscriber* node across 20 executions, while varying the number of clients and message interval. **(A)** Message interval: 0.05 s, **(B)** Message interval: 0.1 s, **(C)** Message interval: 0.2 s, **(D)** Message interval: 0.5 s, **(E)** Message interval: 1.0 s.

groups of programming language and message intervals, where Kruskal–Wallis test results in an $H$ statistic of 286.68 and a *p-value* of $8.09 \times 10^{-61}$ for Python, and in an $H$ statistic of 284.88 and a *p-value* of $1.97 \times 10^{-60}$ for C++. The statistical difference is also observed when grouping programming language and different number of clients. For Python language, one-way ANOVA test results in a $p = 2.52 \times 10^{-8}$, while for C++ language it results in $p = 0.005$.

Upon further analysis of message intervals, the one-way ANOVA reveals a statistically significant difference only for the Python language at the 0.05-s message interval ($p = 2.62 \times 10^{-8}$). However, this finding may be inconclusive, given the anomalies previously observed in the results table and Figure 8A. For the other intervals, Kruskal–Wallis tests show no significant differences among groups, with $p = 0.54$ for the 0.1-s message interval and $p = 0.38$ for the 1.0-s interval. Similarly, one-way ANOVA test across 0.2-s and 1.0-s message intervals indicates no statistical difference among groups, with $p = 0.21$ for both message intervals.

Distinctly from Python, C++ *subscriber* nodes exhibit statistical differences among groups for all message intervals except the 0.1-s interval. Interestingly, the 0.1-s interval also shows the highest variation with two clients, as seen in Figure 8B. We have repeated this experiment to guarantee that this was not added by any noise, and the result is consistent among both executions. Post-hoc tests revealed that, in most cases where there is a statistical difference, it occurs between groups with one and three clients, where gradual increases in the number of clients do not result in significant statistical differences (i.e., from one to 2, and from two to three clients). The only message interval showing statistical differences across all groups is 0.2 s. Analyzing Figure 8C, this interval visually demonstrates the least variation in measurements, which likely influences the statistical outcomes.

The results and statistical tests confirm that programming language and message interval significantly impact the energy efficiency of *subscriber* nodes. In contrast, the number of clients shows only a slight impact on energy consumption, which is expected since the measurements refer to a single *subscriber* node, and the amount of messages received by that node should be independent of the number of clients.

## 6.2 Service

In this section, we present the results for *service server* and *service client* across the experiments with different independent variable combinations.

### 6.2.1 Service server

Table 8 presents the mean values of *service server* measurements across the different combinations of independent variables. Unlike the *publisher* node in the previous results, for all the combinations, CPU usage and energy measurements increase as the number of clients grows. This behavior can be attributed to the nature of the nodes: the *publisher* node relies heavily on underlying layers, such as *rmw*, for message replication, with minimal computation in the node itself. In contrast, the *service server* node involves additional calculations, which may contribute to increased processing and, consequently, higher power consumption. Additionally, as observed in the previous communication pattern, memory usage remains stable and is approximately doubled for the Python implementation compared to the C++ implementation. At high message frequencies, the mean power consumption for C++ is less than one-third of that of Python, a difference that is also reflected in the CPU usage measurements.

Figure 8 depicts the distribution of *subscriber* power measurements. The graphs show that the programming language and message interval are key factors influencing the results. The number of clients seems to affect the two languages differently. For Python, the measurements are less predictable at high message frequencies (0.05-s and 0.1-s intervals) while we observe a clear trend for other message intervals, with power consumption increasing as the number of clients grows. In contrast, for C++, power consumption rises between one and two clients but remains relatively stable between two and three clients, suggesting a one-time effect once multiple clients are involved.

#### 6.2.1.1 Statistical tests

The statistical tests confirm the main findings observed in the results table and graphs. The Kruskal–Wallis test for different message intervals shows a highly significant difference for the C++ node, with $p = 3.49 \times 10^{-55}$, and for the Python node, with $p = 1.98 \times 10^{-58}$. These results indicate a significant difference between groups. For both languages, all pairwise comparisons in the *post hoc* Dunn's test reveal significant statistical differences, with each group differing from the others. On the one hand, when grouping by the number of clients for Python, the one-way ANOVA fails to reject the null hypothesis ($p = 0.19$), suggesting no statistical difference between the groups. On the other hand, for C++, the Kruskal–Wallis test shows a statistically significant difference when grouping by the number of clients ($p = 2.32$). For C++ groups, however, *post hoc* Dunn's test indicates no statistically significant difference between group 2 and group 3, only between group 1 and the others. The observation about C++ groups is also evident in Figure 9, confirming the one-time effect when adding multiple clients. That figure also supports the assumption that Python's statistical unpredictability may be directly linked to its high variation in measurements. However, as a future work, it is important to further investigate the effect of a higher number of clients.

### 6.2.2 Service client

Table 9 presents the mean values of *service client* measurements across various combinations of independent variables. An unexpected trend is observed for both languages, where power consumption and CPU usage decrease as the number of clients increases. This effect is more pronounced at higher message frequencies, with both measurements becoming more stable or showing no significant differences between the 0.2-s and 1.0-s message intervals. Notably, Python likely for *publisher/subscriber* pattern exhibits a larger variation at higher frequencies, suggesting that it tends to be less stable when handling demanding robotic communication.

Figure 10 shows the distribution of mean power consumption across the 20 executions for each combination of independent factors. This confirms the observation from Table 9, where Python exhibits a noticeable reduction in power consumption as the number of clients increases. In contrast, for C++ *service clients*, it is only visually evident that there is an increase in the power consumption from one to two clients, while is observed a reduction when increasing the number of clients from two to three. Additionally, Python measurements display a significant number of outlier data points, whereas C++ measurements do not show this issue. We conducted a careful investigation into the causes of the Python node's unstable behavior but found no issues in the execution logs. We repeated the experiment without the Experiment-Runner orchestrator to avoid any possible noise, which did not change the results. Additionally, we experimented with an alternative Quality of Service (QoS) strategy, inspired by a recently reported issue on GitHub[20]. However, this adjustment did not affect the distribution of the measurements either. Based on these findings, we assume that the nodes functioned correctly and that the instability originates from a Python-related issue, which must motivate further investigation as part of future work.

#### 6.2.2.1 Statistical tests

Kruskal–Wallis test reveals a significant statistical difference between groups of message intervals for both languages ($p = 2.89 \times 10^{-53}$ for Python and $p = 1.44 \times 10^{-57}$ for C++). Post-hoc Dunn's test confirms that all groups are statistically different for both languages. For Python, when grouped by the number of clients, the Kruskal–Wallis test indicates a significant difference among groups ($p = 0.015$). However, the *post hoc* Dunn's test shows that the statistical difference is only evident between group 1 and the others, with no significant difference between groups 2 and 3. Similarly, for C++, groups based on the number of clients also exhibit significant differences (Kruskal–Wallis test, $p = 0.00024$). However, the pairwise *post hoc* test (Tukey HSD) suggests that the difference between groups 1 and 3 is not statistically significant. This observation aligns with the trends depicted in Figure 10.

## 6.3 Actions

In this section, we present the results for the *action server* and *action client*. As for the other communication pattern pairs, we provide related plots and perform statistical tests to validate our visual observations from the data representations.

### 6.3.1 Action server

Table 10 summarizes the mean measurements of the action server node across various message frequencies and numbers of clients over 20 executions. The key observations are as follows: *CPU usage*, and consequently *CPU power*, are consistently influenced by message frequency, with notable increases at high frequencies

20 https://github.com/ros2/rmw/issues/372

TABLE 8 Comparative results on *service server* node with different frequencies and number of clients over 20 executions.

| Language | # Clients | Msg. Interval (s) | Avg. CPU utilization (%) | Avg. Memory utilization (KB) | Total CPU energy (J) | CPU power (W) |
|---|---|---|---|---|---|---|
| *C++* | 1 | 0.05 | 0.52 | 19,768 | 41.5 | 0.23 |
| | | 0.1 | 0.29 | 20,696 | 23.11 | 0.13 |
| | | 0.2 | 0.13 | 19,722 | 11.13 | 0.06 |
| | | 0.5 | 0.09 | 20,856 | 7.16 | 0.04 |
| | | 1.0 | 0.06 | 19,698 | 4.4 | 0.02 |
| | 2 | 0.05 | 0.94 | 19,915 | 70.8 | 0.4 |
| | | 0.1 | 0.5 | 20,939 | 38.1 | 0.21 |
| | | 0.2 | 0.22 | 20,825 | 17.52 | 0.1 |
| | | 0.5 | 0.12 | 20,793 | 10.04 | 0.06 |
| | | 1.0 | 0.08 | 20,810 | 6.25 | 0.03 |
| | 3 | 0.05 | 1.0 | 19,657 | 72.83 | 0.41 |
| | | 0.1 | 0.53 | 20,965 | 40.61 | 0.23 |
| | | 0.2 | 0.24 | 21,016 | 18.84 | 0.11 |
| | | 0.5 | 0.14 | 20,959 | 10.95 | 0.06 |
| | | 1.0 | 0.09 | 20,914 | 6.5 | 0.04 |
| *Python* | 1 | 0.05 | 2.43 | 41,041 | 156.92 | 0.88 |
| | | 0.1 | 1.26 | 39,008 | 90.76 | 0.51 |
| | | 0.2 | 0.55 | 41,063 | 40.73 | 0.23 |
| | | 0.5 | 0.29 | 40,984 | 22.78 | 0.13 |
| | | 1.0 | 0.17 | 41,053 | 13.41 | 0.08 |
| | 2 | 0.05 | 2.61 | 39,111 | 159.29 | 0.89 |
| | | 0.1 | 1.39 | 41,049 | 97.23 | 0.54 |
| | | 0.2 | 0.6 | 39,117 | 44.68 | 0.25 |
| | | 0.5 | 0.32 | 41,156 | 25.04 | 0.14 |
| | | 1.0 | 0.19 | 41,117 | 13.99 | 0.08 |
| | 3 | 0.05 | 2.91 | 39,245 | 171.27 | 0.96 |
| | | 0.1 | 1.52 | 41,215 | 100.04 | 0.56 |
| | | 0.2 | 0.66 | 41,137 | 49.11 | 0.27 |
| | | 0.5 | 0.36 | 41,176 | 27.58 | 0.15 |
| | | 1.0 | 0.21 | 41,201 | 15.9 | 0.09 |

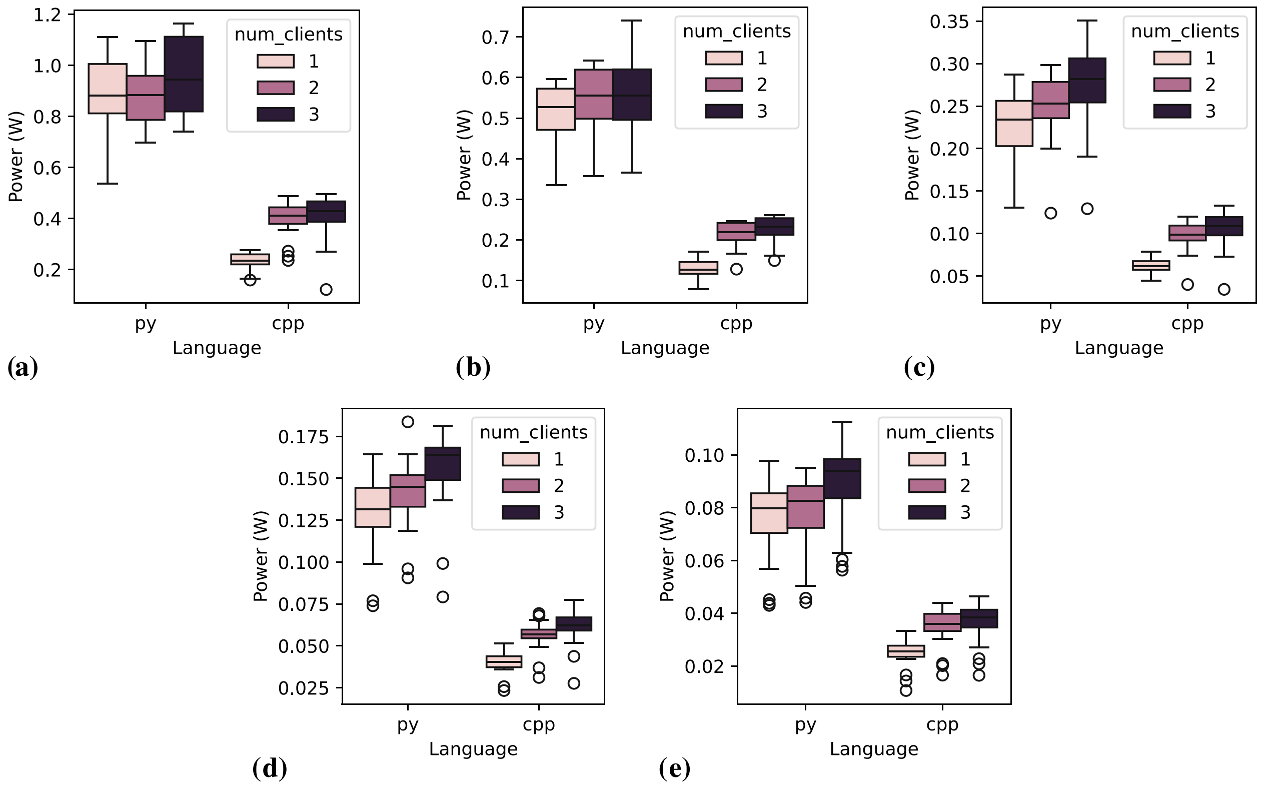


FIGURE 9
Power consumption distribution for the *service server* node across 20 executions, varying the number of clients and message interval. **(A)** Message interval: 0.05 s, **(B)** Message interval: 0.1 s, **(C)** Message interval: 0.2 s, **(D)** Message interval: 0.5 s, **(E)** Message interval: 1.0 s.

corresponding to 0.05-s and 0.1-s intervals. Additionally, memory usage follows a pattern similar to that observed in previous server nodes. Interestingly, at a 0.05-s interval, power consumption decreases when the number of clients increases from two to three, despite CPU usage not exhibiting a corresponding decrease. Given the very low CPU power measurements, this anomaly could be attributed to external noise.

Figure 11 depicts the power consumption distribution for the action server node across 20 executions, varying the number of clients and message interval. At higher frequencies, it is visually evident that both the programming language and message frequency remain key determinants of power consumption. This trend is still noticeable at a 1.0-s message interval, although the difference between the two languages becomes less pronounced, varying by less than 0.1 W in executions with three clients. Unlike other communication server nodes, the C++ implementation appears to be more affected. However, this is inconclusive since it may be a visual misinterpretation, as the overall power consumption for each execution is lower compared to other communication patterns. The most plausible explanation is that, in other communication patterns, the client disconnects and reconnects to the server for every message exchange, whereas in *action* communication, only a new goal is sent, and feedback is received. It is important to note that this behavior is not an implementation issue but rather an inherent characteristic of this communication pattern.

#### 6.3.1.1 Statistical tests

The statistical tests indicate that, as for other communication patterns, both programming languages and message frequencies result in statistically significant differences in power consumption. The Kruskal–Wallis test results in $p = 1.70 \times 10^{-44}$ for Python and $p = 1.35 \times 10^{-54}$ for C++. However, varying the number of clients shows a slight statistical difference among Python groups, which is explained by Dunn's test results, where groups 1 and 2 do not indicate a statistical difference, and the difference for group 3 is less expressive than for the communication patterns. In contrast, for C++, there is a statistically significant difference among all the groups, except at 0.2-s message interval. Further analysis, additionally grouping the data by message frequency reveals that all frequency measurements are statistically different among C++ groups, whereas none of the Python frequency measurements show statistical significance. This indicates that Python either is not holding the concurrent communication properly or it does it precisely well that measurements are not impacted with multi-client executions. The analysis of *action client* figures (Figure 12) gives details of the possible reasons for such behavior, which better aligns with the previous hypothesis.

### 6.3.2 Action client

Table 11 presents the results of a single *action client* node operating at various frequencies and with different numbers of clients across 20 executions. A visual inspection reveals no observable influence of the number of clients on power consumption

TABLE 9 Comparative results of one *service client* node with different frequencies and number of clients over 20 executions.

| Language | # Clients | Msg. Interval (s) | Avg. CPU utilization (%) | Avg. Memory utilization (KB) | Total CPU energy (J) | CPU power (W) |
|---|---|---|---|---|---|---|
| *C++* | 1 | 0.05 | 0.47 | 19,724 | 37.68 | 0.21 |
| | | 0.1 | 0.26 | 20,638 | 20.63 | 0.12 |
| | | 0.2 | 0.12 | 19,660 | 9.81 | 0.05 |
| | | 0.5 | 0.07 | 20,767 | 5.96 | 0.03 |
| | | 1.0 | 0.05 | 19,685 | 4.85 | 0.02 |
| | 2 | 0.05 | 0.61 | 19,781 | 45.96 | 0.26 |
| | | 0.1 | 0.33 | 20,933 | 25.26 | 0.14 |
| | | 0.2 | 0.15 | 20,888 | 12.19 | 0.07 |
| | | 0.5 | 0.09 | 20,805 | 7.04 | 0.04 |
| | | 1.0 | 0.06 | 20,805 | 4.64 | 0.03 |
| | 3 | 0.05 | 0.53 | 20,698 | 39.28 | 0.22 |
| | | 0.1 | 0.28 | 20,934 | 22.05 | 0.12 |
| | | 0.2 | 0.14 | 20,882 | 11.21 | 0.06 |
| | | 0.5 | 0.08 | 20,883 | 7.26 | 0.04 |
| | | 1.0 | 0.05 | 20,902 | 5.13 | 0.03 |
| *Python* | 1 | 0.05 | 3.07 | 40,005 | 221.04 | 1.13 |
| | | 0.1 | 1.74 | 39,038 | 124.8 | 0.7 |
| | | 0.2 | 0.67 | 39,003 | 51.4 | 0.29 |
| | | 0.5 | 0.37 | 40,033 | 28.9 | 0.16 |
| | | 1.0 | 0.2 | 40,063 | 15.73 | 0.09 |
| | 2 | 0.05 | 2.86 | 39,397 | 175.16 | 0.98 |
| | | 0.1 | 1.56 | 41,154 | 108.82 | 0.61 |
| | | 0.2 | 0.65 | 39,129 | 55.62 | 0.28 |
| | | 0.5 | 0.34 | 41,088 | 26.24 | 0.15 |
| | | 1.0 | 0.18 | 40,168 | 13.9 | 0.08 |
| | 3 | 0.05 | 2.67 | 39,734 | 157.21 | 0.88 |
| | | 0.1 | 1.32 | 41,424 | 87.01 | 0.49 |
| | | 0.2 | 0.54 | 41,238 | 39.41 | 0.22 |
| | | 0.5 | 0.3 | 41,208 | 22.67 | 0.13 |
| | | 1.0 | 0.16 | 40,188 | 12.72 | 0.07 |

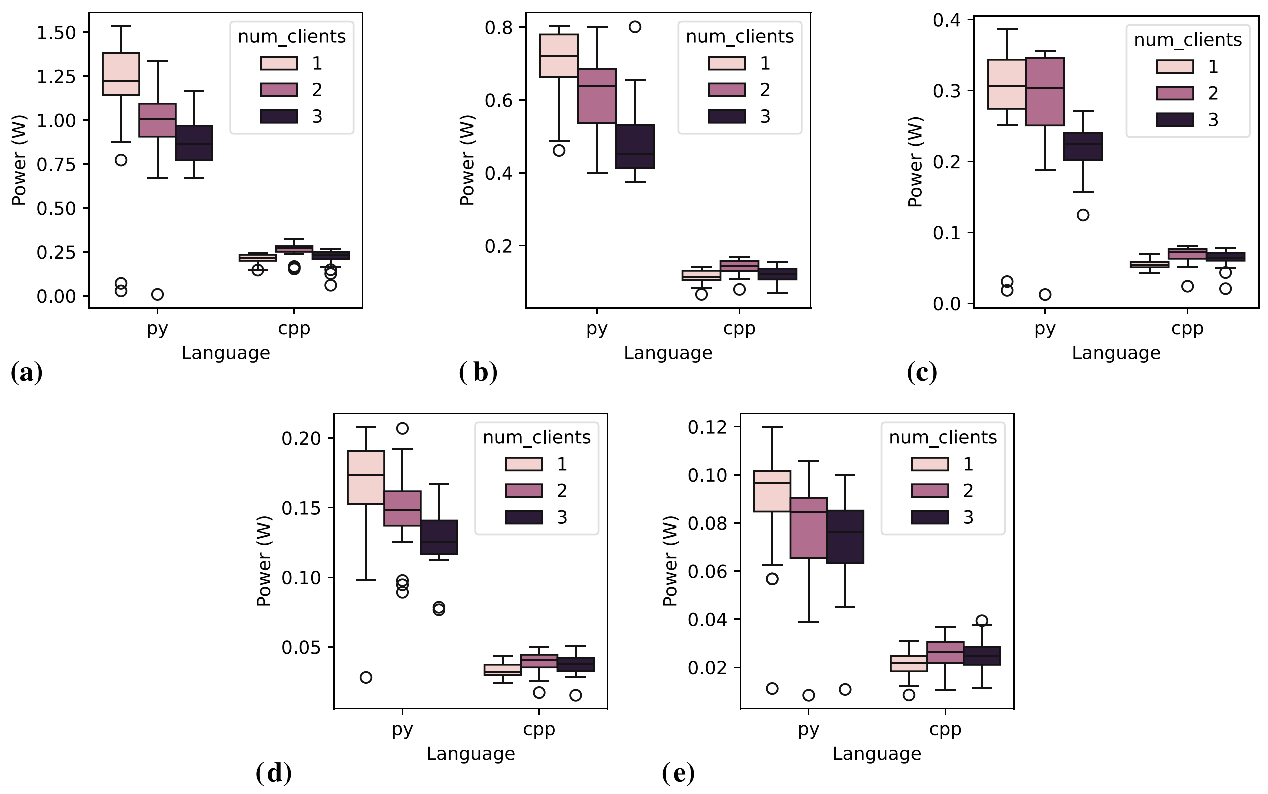


FIGURE 10
Power consumption distribution for one *service client* node across 20 executions, while varying the number of clients and message interval. **(A)** Message interval: 0.05 s, **(B)** Message interval: 0.1 s, **(C)** Message interval: 0.2 s, **(D)** Message interval: 0.5 s, **(E)** Message interval: 1.0 s.

for C++. However, in Python, a consistent decrease in power consumption is observed as the number of clients increases, which differs from *action server*, and even the CPU usage consistently decreases with more clients. Another interesting and related observation is at 1.0-s message interval, when Python language seems to consume less energy than C++, except with runs with a single client. This consistent reduction in Python's power consumption suggests that all measurements with one client require more power compared to those with two or three clients. This behavior is likely linked to multi-client architectural triggering, such as synchronization or multi-threading mechanisms, which are potentially activated in such scenarios.

Figure 12 illustrates the distribution of mean power consumption for a single *action client* node over 20 executions, varying the number of clients and the message interval. The observations from Table 11 are corroborated by the sub-figures. Notably, disruptive measurements are evident for Python nodes when operating with two or three clients, as compared to a single client. Additionally, the power consumption gap between Python and C++ narrows as the message interval increases. At a 1.0-s message interval, Python measurements seem to be compatible with C++ ones.

In a practical context, where an action client might send navigation or manipulation tasks to a robot, having more than one client is generally unnecessary, except for the need for feedback messages by secondary nodes. Therefore, this configuration might be neglected, which could cause the unexpected behavior observed. Since the results focus on a single client (with monitoring limited to the last client), we re-executed the Python experiments to monitor all clients simultaneously. This approach aimed to verify whether any client exhibited anomalous behavior, which was not identified over a thorough log analysis.

#### 6.3.2.1 Statistical tests

The statistical tests confirm that programming language and message frequency are determinant factors, except for C++, where 0.05- and 0.1-s message intervals do not result in statistically significant differences in power measurements. For C++, different numbers of clients result in statistical differences on the measured power consumption, except between groups 2 and 3. For Python, the number of clients also leads to statistically different power measurements among all the groups. An additional Kruskal–Wallis between the two languages at 1.0-s interval indicates no statistical difference ($p = 0.92$) between groups, confirming the visual assumption when analyzing Figure 12E, that both languages result in closely the same power consumption at low message frequency.

## 6.4 Comparison of communication pattern measurements

Since we experiment all the studied communication patterns with a controlled and homogeneous scenario, in this section,

TABLE 10 Comparative results on *action server* node with different frequencies and number of clients over 20 executions.

| Language | # Clients | Msg. Interval (s) | Avg. CPU utilization (%) | Avg. Memory utilization (KB) | Total CPU energy (J) | CPU power (W) |
|---|---|---|---|---|---|---|
| *C++* | 1 | 0.05 | 0.32 | 21,680 | 14.85 | 0.08 |
| | | 0.1 | 0.33 | 21,102 | 15.52 | 0.09 |
| | | 0.2 | 0.13 | 21,152 | 7.41 | 0.04 |
| | | 0.5 | 0.09 | 21,058 | 5.31 | 0.03 |
| | | 1.0 | 0.07 | 20,084 | 3.79 | 0.02 |
| | 2 | 0.05 | 0.37 | 21,884 | 16.93 | 0.09 |
| | | 0.1 | 0.36 | 21,590 | 17.16 | 0.1 |
| | | 0.2 | 0.19 | 20,694 | 8.84 | 0.05 |
| | | 0.5 | 0.12 | 20,582 | 5.26 | 0.03 |
| | | 1.0 | 0.08 | 19,610 | 3.68 | 0.02 |
| | 3 | 0.05 | 0.43 | 21,696 | 19.59 | 0.11 |
| | | 0.1 | 0.41 | 21,408 | 19.14 | 0.11 |
| | | 0.2 | 0.31 | 18,826 | 10.03 | 0.06 |
| | | 0.5 | 0.17 | 19,620 | 5.23 | 0.03 |
| | | 1.0 | 0.11 | 18,630 | 3.49 | 0.02 |
| *Python* | 1 | 0.05 | 2.03 | 42,888 | 85.47 | 0.48 |
| | | 0.1 | 1.0 | 40,891 | 45.9 | 0.26 |
| | | 0.2 | 0.44 | 42,060 | 22.76 | 0.13 |
| | | 0.5 | 0.25 | 39,720 | 13.96 | 0.08 |
| | | 1.0 | 0.16 | 39,654 | 8.14 | 0.05 |
| | 2 | 0.05 | 2.15 | 42,872 | 86.08 | 0.48 |
| | | 0.1 | 1.1 | 42,907 | 47.56 | 0.26 |
| | | 0.2 | 0.47 | 42,104 | 23.84 | 0.13 |
| | | 0.5 | 0.26 | 42,060 | 14.05 | 0.08 |
| | | 1.0 | 0.16 | 41,865 | 8.43 | 0.05 |
| | 3 | 0.05 | 2.21 | 40,737 | 83.3 | 0.46 |
| | | 0.1 | 1.13 | 42,915 | 48.09 | 0.27 |
| | | 0.2 | 0.48 | 42,006 | 23.86 | 0.13 |
| | | 0.5 | 0.27 | 42,030 | 13.97 | 0.08 |
| | | 1.0 | 0.18 | 41,694 | 8.47 | 0.05 |

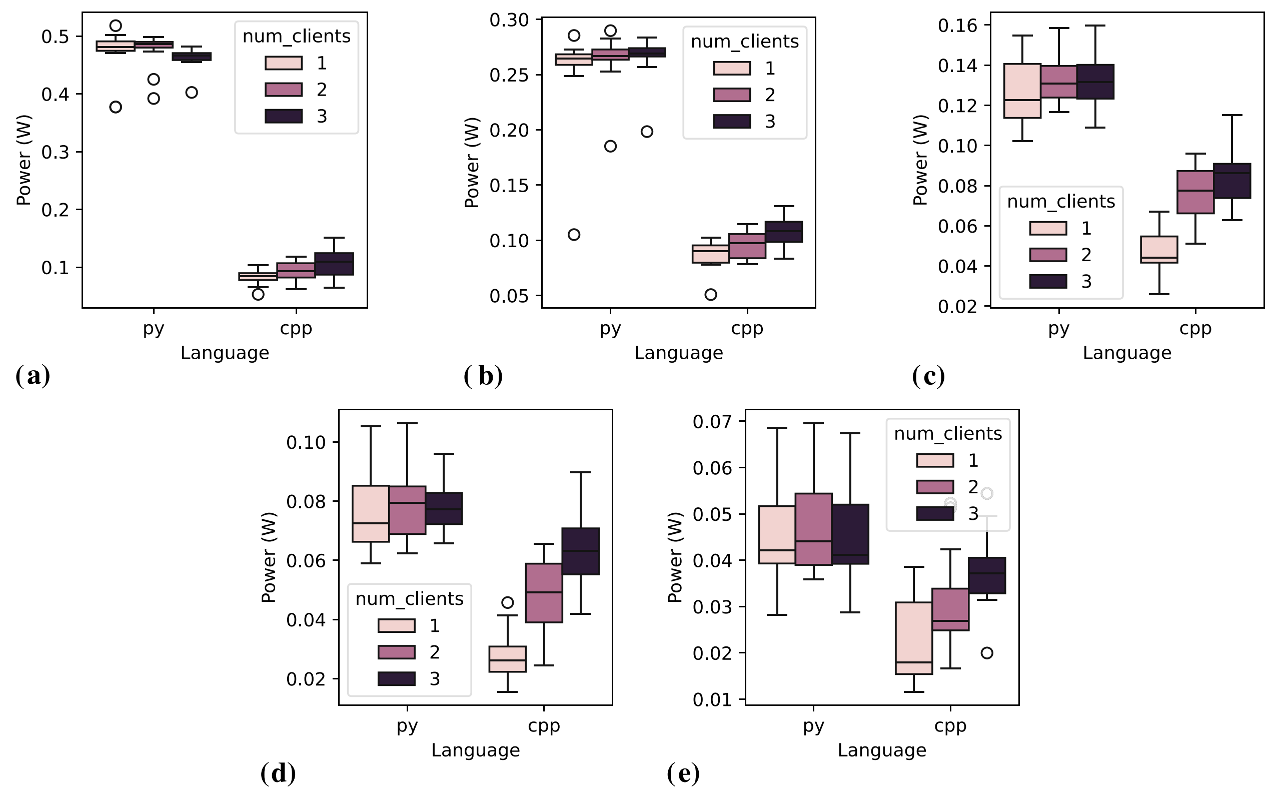


FIGURE 11
Power consumption distribution for the *action server* node across 20 executions, varying the number of clients and message interval. **(A)** Message interval: 0.05 s, **(B)** Message interval: 0.1 s, **(C)** Message interval: 0.2 s, **(D)** Message interval: 0.5 s, **(E)** Message interval: 1.0 s.

we compare their overall measurements, dividing them into *publisher/server* and *subscriber/client*.

### 6.4.1 Publisher/server measurements

Figure 13 illustrates the distribution of power consumption across all combinations of independent variables (configurations) for the *publisher/server* nodes over 20 executions. The plot reveals a consistent mean CPU power consumption across the different nodes, although there is an observable distinction across distribution variations. The *action server* presents the least variation in measurements, while the *service server* shows the greatest fluctuation.

#### 6.4.1.1 Statistical tests

The Shapiro-Wilk test of three groups (*publisher*, *service server*, and *action server*) indicates that their data are significantly non-normally distributed. Following this, the Kruskal–Wallis test was performed to compare the groups, which resulted in a highly significant result ($H$ statistic = 227.22, $p = 4.56 \times 10^{-50}$), indicating substantial differences between the groups. Further analysis with Dunn's *post hoc* test revealed that all pairwise group comparisons were statistically significant, with very small *p-values*. This confirms that despite the consistent mean values observed in Figure 13, the power consumption among the three nodes is significantly different.

### 6.4.2 Subscriber/client measurements

Figure 14 illustrates the distribution of power consumption across all combinations of independent variables (configurations) for *subscriber/client* nodes over the 20 executions. The graphs show a tighter distribution of values than those for *publisher/server* nodes, which also seem to result in closer mean values. Among the plots, the *service client* and *action client* show the most similar distributions, while the *subscriber* displays a slightly different pattern.

#### 6.4.2.1 Statisitcal tests

The Shapiro-Wilk test results indicate that the data for all three groups is likely not normally distributed. Therefore, the Kruskal–Wallis test was performed to assess differences among the groups. The test reveals a significant result ($H$ statistic = 80.48, $p = 3.34 \times 10^{-18}$), indicating differences between the groups. Post-hoc analysis using Dunn's test reveals significant pairwise differences among all the groups, with *action client* showing the most significant differences compared to the others. This confirms our observation about the *action client* distribution, although rejecting the hypothesis of power distributions being close, despite their consistent measurement means.

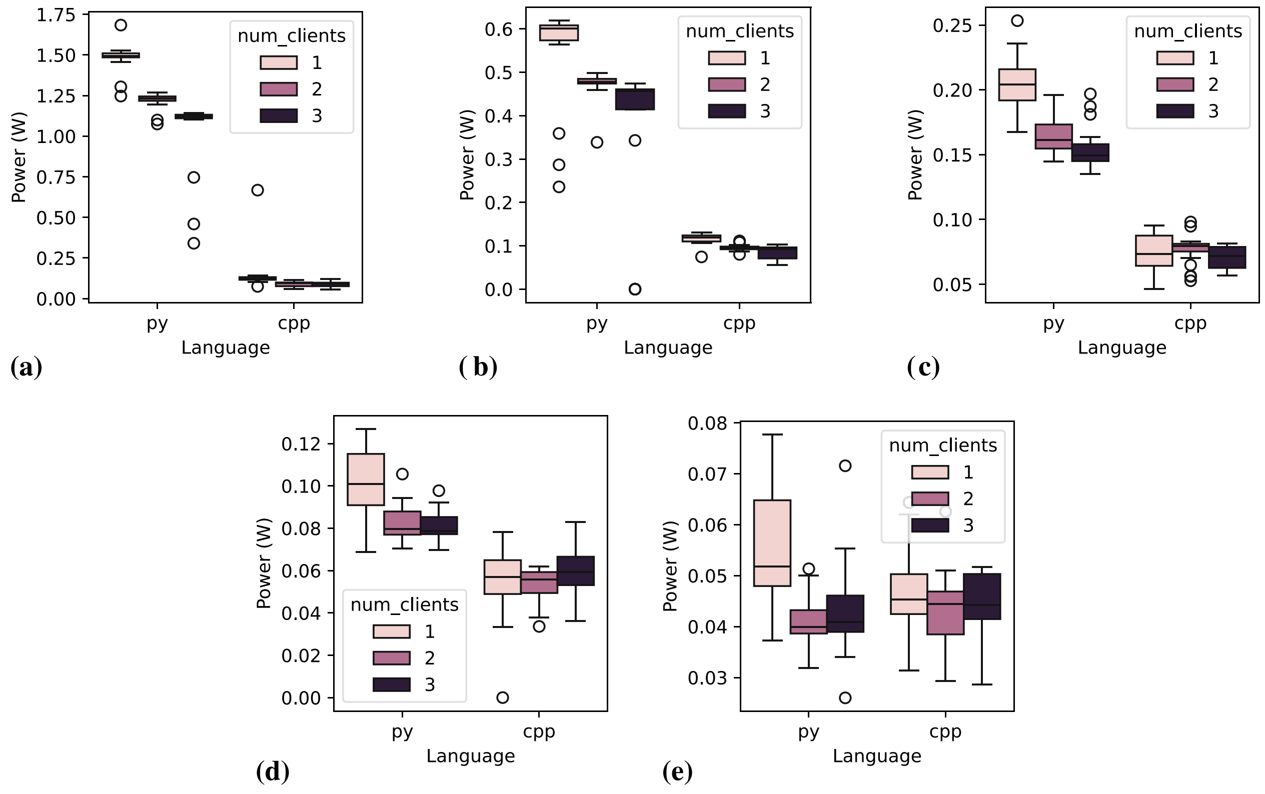


FIGURE 12
Power consumption distribution for one *action client* node across 20 executions, while varying the number of clients and message interval. **(A)** Message interval: 0.05 s, **(B)** Message interval: 0.1 s, **(C)** Message interval: 0.2 s, **(D)** Message interval: 0.5 s, **(E)** Message interval: 1.0 s.

## 7 Discussion

In this section, we summarize and discuss further details of the experiment results, answer the research questions, and ponder about the impact of the findings of our investigation.

### 7.1 Summary of main findings

Here, we summarize the findings discussed as topics, which makes it easier for the reader to navigate through them.

- *C++ is the most efficient programming language*: C++ consistently outperformed Python in terms of energy efficiency and resource usage across all ROS two communication patterns. This was already expected since a previous study that bases this research (Pereira et al., 2021) has already revealed C and C++ superiority regarding energy efficiency. However, in the case of ROS, we expected a shrank difference since both language libraries (*rclpy* and *rclcpp*) share the same underlying programming and communication layers.
- *Message interval is a determinant factor*: Higher message frequencies led to increased power consumption in both C++ and Python, highlighting the importance of optimizing message intervals. The highest message rates experimented, i.e., 0.05-s and 0.1-s message intervals, recurrently led to overhead, indicating the it is important to limit the message exchange to higher rates, starting from four messages per second (0.2-s message interval).
- *The number of clients triggers unexpected behaviors*: Despite being less impactful, the number of clients is still a determining factor that must be considered in the design phase of ROS software systems. It also revealed potential architectural issues, such as for Python *action client* nodes, that result in unexpected low power measurements when scaling from one to two or three clients, which foster further investigation.
- *Python's scalability is unpredictable*: Python exhibited less predictable and often unstable behavior as the number of clients increased, particularly at high frequencies. This suggests potential limitations in Python's ability to handle demanding robotic communication scenarios efficiently.
- *Servers are directly impacted by different independent variables*: The number of clients significantly impacted power consumption on the server-side nodes, but the specific effects varied depending on the communication pattern.
- *Clients are less susceptible to the number of clients*: It is expected that the number of clients do not affect clients directly since it do not result in extra workload. However, some task from the underlying architecture or the server, possibly related to synchronization, seems to affect such nodes as well.
- *Experiments revealed potential issues*: The research suggests that the dependency of programming language (C++ or Python) on

TABLE 11 Comparative results of one *action client* node with different frequencies and number of clients over 20 executions.

| Language | # Clients | Msg. Interval (s) | Avg. CPU utilization (%) | Avg. Memory utilization (KB) | Total CPU energy (J) | CPU power (W) |
|---|---|---|---|---|---|---|
| *C++* | 1 | 0.05 | 0.59 | 2073 | 21.28 | 0.15 |
| | | 0.1 | 0.45 | 2,117 | 21.97 | 0.12 |
| | | 0.2 | 0.25 | 1984 | 14.22 | 0.07 |
| | | 0.5 | 0.19 | 1946 | 10.52 | 0.05 |
| | | 1.0 | 0.16 | 1864 | 9.01 | 0.05 |
| | 2 | 0.05 | 0.35 | 2,175 | 16.32 | 0.09 |
| | | 0.1 | 0.36 | 2,192 | 17.72 | 0.1 |
| | | 0.2 | 0.26 | 1983 | 14.53 | 0.08 |
| | | 0.5 | 4.06 | 1932 | 9.00 | 0.05 |
| | | 1.0 | 0.15 | 1965 | 8.34 | 0.04 |
| | 3 | 0.05 | 0.35 | 2,214 | 16.23 | 0.09 |
| | | 0.1 | 0.33 | 2,152 | 15.83 | 0.09 |
| | | 0.2 | 2.76 | 2073 | 12.60 | 0.07 |
| | | 0.5 | 0.22 | 1975 | 11.37 | 0.06 |
| | | 1.0 | 0.16 | 1837 | 8.56 | 0.04 |
| *Python* | 1 | 0.05 | 6.36 | 4,129 | 273.79 | 1.48 |
| | | 0.1 | 2.08 | 3,817 | 94.33 | 0.55 |
| | | 0.2 | 0.71 | 4,140 | 38.59 | 0.2 |
| | | 0.5 | 0.34 | 3,519 | 19.3 | 0.1 |
| | | 1.0 | 0.19 | 3,496 | 10.46 | 0.06 |
| | 2 | 0.05 | 5.59 | 4,140 | 225.14 | 1.22 |
| | | 0.1 | 1.99 | 4,138 | 87.09 | 0.47 |
| | | 0.2 | 0.58 | 4,142 | 31.23 | 0.17 |
| | | 0.5 | 0.28 | 3,870 | 15.56 | 0.08 |
| | | 1.0 | 0.15 | 3,693 | 7.96 | 0.04 |
| | 3 | 0.05 | 5.15 | 4,138 | 174.65 | 1.03 |
| | | 0.1 | 1.59 | 4,138 | 63.47 | 0.36 |
| | | 0.2 | 0.56 | 3,925 | 29.19 | 0.15 |
| | | 0.5 | 0.29 | 3,688 | 15.77 | 0.08 |
| | | 1.0 | 0.2 | 3,668 | 9.06 | 0.04 |

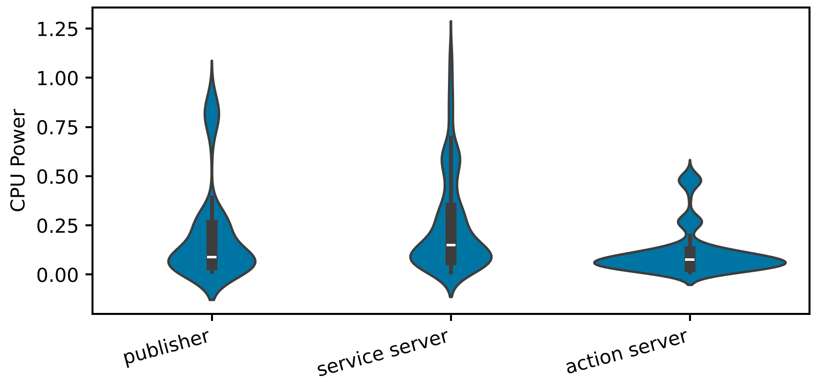


FIGURE 13
Power consumption distribution over all the combinations of independent variables for *publisher/server* nodes across the 20 executions.

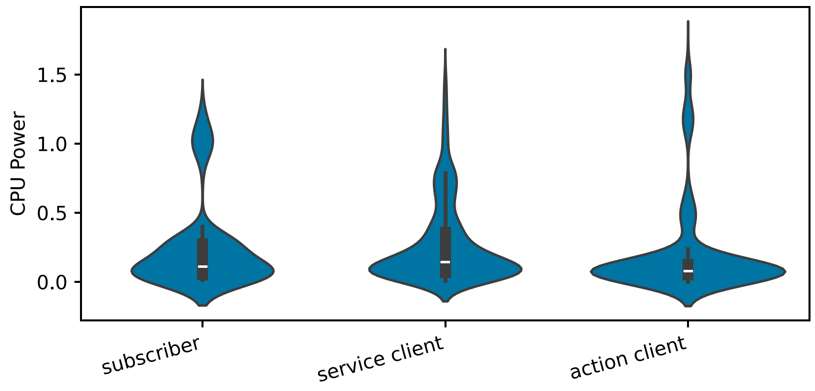


FIGURE 14
Power consumption distribution over all the combinations of independent variables for *subscriber/client* nodes across the 20 executions.

the underlying ROS two architecture (DDS middleware, client libraries) plays a crucial role in energy efficiency. For instance, unexpectedly, power consumption decreased as the number of clients increased on the client-side for services and actions. We could not identify any anomaly in the experiment executions after a careful investigation of logs and measurement data. A quick investigation on *rclpy* and *rclcpp* libraries does not reveal substantial evidence of such a behavior either.

## 7.2 Research question answers

The research questions are repeated here, avoid seeking fro them back in the document while reading their answers.

*RQ1: How is the energy efficiency of each ROS communication pattern?*

The statistical tests described in Section 6.4 demonstrate that the measurement data for the three communication patterns differ significantly, highlighting that each pattern has a distinct impact on energy efficiency. However, the mean values of the measurement distributions for the *publisher/server* and *subscriber/clients* patterns (Figures 13, 14, respectively) are closely aligned. Additionally, the violin plots in these figures reveal that most measurements, particularly for the *subscriber/clients*, are concentrated within a similar range. Given these findings, we recommend a careful design study when selecting a communication pattern. Nonetheless, we acknowledge that the patterns can likely be interchanged without significant consequences for energy efficiency, especially in the case of *publisher/subscriber* and *service* patterns.

*RQ2: How do the `C++` and `Python` implementations affect resource usage and energy consumption when handling different communication patterns among ROS nodes?*

The programming language is a determinant factor, where Python leads to higher power consumption through all the experiment results. This is an expected behavior based on recent research with programming languages for data structure algorithms (Pereira et al., 2021); however, the difference in power consumption among the two studied languages is surprising given the amount of underlying architectural layers shared by both language libraries. After the experiments, we strongly recommend the use of C++ for ROS two implementations, which can benefit scalability, reliability (due to a more predictable behavior), and energy/resource usage efficiency.

*RQ3: How does language efficiency scale over different frequencies of communication and number of clients?*

We observe that message frequency is a strong determinant factor of energy efficiency that should be carefully considered when designing ROS systems. High message frequencies demonstrate to lead to resource overhead, triggering unstable behaviors, particularly in *Python* nodes when more than one client connects with the servers, which significantly compromises scalability. While the number of clients is a less critical factor compared to the programming language and message interval, it remains an important consideration. It impacts not only energy efficiency but also introduces unexpected behaviors, such as those observed with *actions*. These behaviors may indicate poor design choices, especially since multi-client setups are often impractical in most scenarios where such patterns are applied. Therefore, we recommend a careful design when considering a multi-client ROS system.

## 7.3 Impact of the findings

The findings of this research have significant implications for the development and operation of real-world robotic systems using ROS. In the sequence, we discuss some of the key implications. As an excerpt of real-world robotic projects, we analyzed a list of 946 carefully curated ROS two repositories on GitHub[21], obtained from a separate ongoing research project by the authors. Those projects were selected considering quantitative criteria that make the projects to be representative (such as number of forks, number of followers and contributors, and size), which numbers are included in the dataset.

Due to its high level of abstraction, Python is a particularly attractive language for newcomers to programming, which can also be the case of those starting with robotics and ROS. However, as these results suggest, the widespread use of Python can lead to significant energy inefficiencies, impacting projects that rely on battery-powered robots, besides the environmental side-effects. Among the real-world repositories in the referenced dataset, 291 (30%) utilize Python as their primary language. This represents a substantial number of projects, which can be directly used, reused as packages, or serve as models for future implementations. Disseminating this paper's findings to both, academic and industry communities, can lead to more informed decisions regarding programming language selection in robotics projects, which will potentially benefit resource and energy efficiency.

The direct correlation between message frequency and power consumption highlights the critical need for optimization of message intervals within ROS two systems. Robotics developers should attempt to minimize message frequencies while ensuring that essential system functionality remains unimpaired. To understand this impact, we manually queried the first repositories in the real-world dataset. A superficial analysis reveals that service calls tend to be less affected by high frequencies since they are usually triggered by events, such as in the *main_camera_node.cpp* file of *cyberdog_ros2* project[22], which triggers camera-related services on *configure* event and resets it on *clean up* event. However, for *publisher/subscriber* cases, the impact tends to be more critical. A critical example of this is the *webots_ros2* project[23], where the *epuck_node.py* file implements a node with multiple subscriptions to different topics, and do not pace the communication with any delay or sleep. In such cases, message frequency is primarily dictated by factors such as the execution time of the whole algorithm. For simple algorithms, these execution times can result in fractions of seconds, leading to excessively high message frequencies, a significant concern in our experiment results, particularly for Python-based projects.

The final factor, and the least impactful, is the number of clients. This scenario is more commonly associated with the *publisher/subscriber* communication model, given its sensor-based nature. In a manual analysis of the first 20 projects in the dataset, we identified only one project, *ros2_canopen*[24], with more than one subscriber. In this project, the *node_name* + *rpdo* topic is subscribed to by two different test nodes (*test_node* and *simple_rpdo_tpdo_tester*), while the *low_level/joint_states* topic is subscribed to by the *noarm_squat* and *wiggle_arm* nodes. Despite multiple clients seeming to be less common, the repositories' manual inspection supports the legitimacy of our concerns regarding multiple clients and suggests that our observations can also contribute to thoughtful designs that take the number of clients into account.

## 7.4 Open issues

Unfortunately, the results reveal a few issues whose sources we were unable to identify. We outline these issues below to encourage further investigations, as their resolution and deep investigation lie outside the scope of this paper. We confirm that they are not related to issues in our algorithms or their executions, which have already been discussed in the previous sections.

1. The Python *publisher* and *subscriber* mechanisms exhibit high variability at elevated frequencies, suggesting some form of overhead that leads to unpredictable power consumption.
2. The C++ *service server* demonstrates an initial power increase when the number of clients rises from one to two. However, the power consumption stabilizes as the number of clients increases from two to three. This behavior, distinct from that observed in Python, suggests a one-time synchronization method for handling multiple clients.
3. While the Python *service server* shows an increase in power consumption as the number of clients grows, the Python *service clients* exhibit a consistent decrease in power consumption. This may indicate challenges faced by the server in addressing all client requests, although this hypothesis is not supported by manual log analysis.
4. At low message frequencies, the *action* pattern results in similar power consumption for both languages. This consistency is not observed for other communication patterns.

21 https://github.com/IntelAgir-Research-Group/frontiers-robotics-ai-rep-pkg/blob/main/data-analysis/repos.csv

22 https://github.com/MiRoboticsLab/cyberdog_ros2/

23 https://github.com/cyberbotics/webots_ros2

24 https://github.com/ros-industrial/ros2_canopen

## 8 Threats to validity

In this section, we discuss potential threats to the validity of our experiments, outline considerations, and describe how we address each of them.

### 8.1 External validity

One limitation lies in the simplicity of the messages exchanged between ROS two nodes in our experiments. We focused on plain text messages in the *publisher/subscriber* pattern, which are less complex compared to sensor messages like `PointCloud`[25] or geometry-based messages[26]. However, our results show consistent behavior across various configurations and communication patterns, with significant differences between configurations, suggesting that the experimental variables likely impact systems using more complex message types. Additionally, prior work (Albonico et al., 2024) involving more diverse message types indicates that at least the number of clients plays a critical role, reinforcing the applicability of our findings. To facilitate further exploration, our replication package supports extensions to other communication patterns and message types.

To mitigate potential biases due to the representativity of the implemented ROS two nodes, we based our implementation on official ROS two tutorials. This ensures relevance and applicability to a wide range of general-purpose applications since they may work as a template for different types of applications worldwide. Moreover, our study uses ROS 2 Humble, an active distribution supported until 2027, enhancing the relevance and timeliness of our findings.

### 8.2 Internal validity

We ensured internal validity by maintaining a strictly controlled experimental environment, minimizing the influence of variations in system load, background processes, or hardware inconsistencies. All experiments were conducted using the same tools and environment and repeated twenty times to account for variability. For CPU and memory usage measurements, we relied on widely used Python libraries. Energy consumption measurements were conducted using a tool extensively validated in prior studies (Kamatar et al., 2024; Nahrstedt et al., 2024; Makris et al., 2024). This rigorous approach minimizes confounding factors and strengthens the reliability of our conclusions.

The use of Docker in our experimental setup introduces a potential internal threat, as it may slightly influence energy measurements due to its resource isolation and runtime overhead. These effects could introduce systematic measurement biases, affecting the accuracy and reproducibility of our results. This is mitigated by a controlled execution, where we compare the energy usage of different ROS two communication methodologies within a comparable environment. This approach helps mitigate the potential impact of Docker-induced variations, as any overhead introduced by containerization would be present across all experimental conditions. Since we are primarily interested in how different communication patterns compare to each other in terms of energy consumption, rather than the exact power drawn by each, minor variations introduced by Docker do not compromise the validity of our conclusions.

### 8.3 Construct validity

Our experiments were designed with well-established metrics that align with the goals of this research (Pereira et al., 2017; Hähnel et al., 2012). Power consumption, a primary metric, directly measures the rate of energy usage while running ROS nodes, capturing nuanced differences in energy efficiency attributable to language-specific and architectural factors. This metric is independent of confounding variables such as execution time, ensuring that observed effects are solely related to energy efficiency. Furthermore, all findings were validated using a robust statistical testing strategy, ensuring the reliability of our conclusions and alignment with the study's objectives.

## 9 Related work

The energy efficiency of software has received significant attention in recent years, particularly concerning the deployment infrastructures and programming languages utilized in various applications. This aligns closely with the objectives of our research, which aims to investigate the energy efficiency of programming languages within the Robot Operating System (ROS) ecosystem. Notably, a comprehensive search across major research databases revealed a gap in the literature, as no previous studies have specifically addressed the energy efficiency of programming languages in the context of ROS.

The work by Pereira et al. (2017) stands out as the most related to our investigation. Their extensive study evaluated energy consumption across a variety of algorithms implemented in different programming languages, producing a ranking based on energy efficiency. While their findings provide valuable insights, the algorithms they examined were executed natively, without the influence of middleware or frameworks, contrasting with our focus on ROS-specific algorithms that operate within an active ROS stack. This distinction is crucial, as the architecture and operational context of ROS can significantly impact energy consumption metrics.

Other studies have explored the energy efficiency of programming languages in different contexts. For instance, Kholmatova (2020) examined the impact of programming languages on energy consumption for mobile devices, highlighting how language choice can influence energy efficiency. Similarly, Abdulsalam et al. (2014) investigated the effects of language, compiler optimizations, and implementation choices on program energy efficiency, emphasizing the importance

25 https://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/PointCloud.html

26 https://docs.ros2.org/foxy/api/geometry_msgs/index-msg.html

of these factors in software development. Furthermore, research by Holm et al. (2020) focused on GPU computing with Python, analyzing performance and energy efficiency, which underscores the relevance of programming paradigms in energy consumption.

In our previous work (Albonico et al., 2024), we investigated the energy efficiency of ROS nodes implemented in C++ and Python. Their study focused on the publisher-subscriber communication pattern and assessed the power consumption of ROS nodes in both languages. The results demonstrated that C++ nodes exhibit superior energy efficiency compared to Python nodes, particularly in scenarios with multiple subscribers. This difference was attributed to the architecture of the client libraries and the native multi-threading capabilities of C++. However, compared to this paper, their study considered fewer independent variables, followed a less systematic methodology, and provided only preliminary results with limited discussion.

## 10 Conclusion

Our study provides a comprehensive analysis of the energy efficiency of ROS two communication patterns, revealing that C++ implementations consistently outperform Python in terms of energy efficiency and resource usage. Message frequency significantly influences power consumption, while the number of clients has a less predictable impact, particularly for Python. These findings have significant implications for real-world robotic systems, guiding programming language choices, message frequency optimization, and system architecture considerations. Therefore, our research contributes to a better understanding of energy efficiency in ROS 2, promoting the development of greener robotic software.

Despite the consistency of our findings across various configurations and communication patterns, there are still a few open issues that can be further investigated. For example, further research can explore the impact of message type complexity, particularly with other types of messages. We also plan to examine the specific synchronization or multi-threading mechanisms in both C++ and Python service servers and clients to understand the observed power consumption trends as the number of clients increases. Finally, another possible extension of this research, is to analyze the action pattern to determine why it results in similar power consumption for both C++ and Python at low message frequencies.

## Data availability statement

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found in the article/supplementary material.

## Author contributions

MA: Conceptualization, Data curation, Formal Analysis, Investigation, Methodology, Project administration, Software, Supervision, Validation, Visualization, Writing–original draft, Writing–review and editing. MC: Software, Writing–original draft. AW: Funding acquisition, Project administration, Supervision, Writing–review and editing, Conceptualization.

## Funding

The author(s) declare financial support was received for the research, authorship, and/or publication of this article. This research was made possible through the support and funding of the following grants: the National Council for Scientific and Technological Development (CNPq), Grant Process 200006/2024-0; InnovationsCampus Mobilität der Zukunft, Mid-term Fellowships for Experts; and UTPFR/PROPPG 12/04 call for research groups in the computer science field.

## Acknowledgments

We would like to express our gratitude to Paulo Júnior Varela and Adair José Rohling for their contributions to the previous paper, which served as the foundation for this research. Additionally, we acknowledge the use of a generative artificial intelligence tool to assist with English language corrections, with no content generated by the tool.

## Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## Generative AI statement

The author(s) declare that Generative AI was used in the creation of this manuscript. Only to correct the English language writing.

## Publisher’s note

## References

Abdi, H., and Williams, L. J. (2010). Tukey's honestly significant difference (HSD) test. *Encycl. Res. Des.* 3 (1), 1–5.

Abdulsalam, S., Lakomski, D., Gu, Q., Jin, T., and Zong, Z. (2014). "Program energy efficiency: the impact of language, compiler and implementation choices," in *International green computing conference* (IEEE), 1–6.

Albonico, M., Junior Varela, P., Jose Rohling, A., and Wortmann, A. (2024). *Energy efficiency of ROS nodes in different languages: publisher/subscriber case studies (RoSE '24)*. New York, NY, USA: Association for Computing Machinery, 1–8. doi:10.1145/3643663.3643963

Basili, V., Caldiera, G., and Rombach, H. D. (1994). The goal question metric approach. *Encycl. Softw. Eng.*, 528–532.

Bernard, L. W. (1951). On the comparison of several mean values: an alternative approach. *Biometrika* 38 (3/4), 330–336. doi:10.2307/2332579

Chinnappan, K., Malavolta, I., Lewis, G. A., Albonico, M., and Lago, P. (2021). "Architectural tactics for energy-aware robotics software: a preliminary study," in *European conference on software architecture* (Springer), 164–171.

Ciccozzi, F., Di Ruscio, D., Malavolta, I., Pelliccione, P., and Tumova, J. (2017). "Engineering the software of robotic systems," in *2017 IEEE/ACM 39th international conference on software engineering companion (ICSE-C)* (IEEE), 507–508.

George, E. P. B., and Cox, D. R. (1964). An analysis of transformations. *J. R. Stat. Soc. Ser. B Stat. Methodol.* 26 (2), 211–243. doi:10.1111/j.2517-6161.1964.tb00553.x

Hähnel, M., Döbel, B., Völp, M., and Härtig, H. (2012). Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Perform. Eval. Rev.* 40 (3), 13–17. doi:10.1145/2425248.2425252

Hirao, E., Miyamoto, S., Hasegawa, M., and Harada, H. (2005). "Power consumption monitoring system for personal computers by analyzing their operating states," in *2005 4th international symposium on environmentally conscious design and inverse manufacturing*, 268–272. doi:10.1109/ECODIM.2005.1619220

Holm, H. H., Brodtkorb, A. R., and Sætra, M. L. (2020). GPU computing with Python: performance, energy efficiency and usability. *Computation* 8 (1), 4. doi:10.3390/computation8010004

Kamatar, A., Hayot-Sasson, V., Babuji, Y., Bauer, A., Rattihalli, G., Hogade, N., et al. (2024). GreenFaaS: maximizing energy efficiency of HPC workloads with FaaS. *arXiv Prepr. arXiv:2406.17710*. doi:10.48550/arXiv.2406.17710

Kholmatova, Z. (2020). Impact of programming languages on energy consumption for mobile devices. 1693–1695. doi:10.1145/3368089.3418777

Koubaa, A. (2015). ROS as a service: web services for robot operating system. *J. Softw. Eng. Robotics* 6 (1), 1–14.

Kruskal, W. H., and Wallis, W. A. (1952). Use of ranks in one-criterion variance analysis. *J. Am. Stat. Assoc.* 47 (260), 583–621. doi:10.1080/01621459.1952.10483441

Makris, A., Korontanis, I., Psomakelis, E., and Tserpes, K. (2024). "An efficient storage solution for cloud/edge computing infrastructures," in *2024 IEEE international conference on service-oriented system engineering (SOSE)* (IEEE), 92–101.

Nahrstedt, F., Karmouche, M., Bargieł, K., Banijamali, P., Kumar, A. N. P., and Malavolta, I. (2024). "An empirical study on the energy usage and performance of pandas and polars data analysis Python libraries," in *Proceedings of the 28th international conference on evaluation and assessment in software engineering*, 58–68.

Nizam Khan, K., Hirki, M., Niemi, T., Nurminen, J. K., and Ou, Z. (2018). RAPL in action: experiences in using RAPL for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3 (2), 1–26. doi:10.1145/3177754

Noureddine, A. (2022). "Powerjoular and joularjx: multi-platform software power monitoring tools," in *2022 18th international conference on intelligent environments (IE)* (IEEE), 1–4.

Noureddine, A. (2024). "Analyzing software energy consumption," in *Proceedings of the 2024 IEEE/ACM 46th international Conference on software engineering: companion proceedings (lisbon, Portugal) (ICSE-Companion '24)* (New York, NY, USA: Association for Computing Machinery), 424–425. doi:10.1145/3639478.3643058

Olive Jean Dunn (1961). Multiple comparisons among means. *J. Am. Stat. Assoc.* 56 (293), 52–64. doi:10.2307/2282330

Pardo-Castellote, G. (2003). "Omg data-distribution service: architectural overview," in *23rd international conference on distributed computing systems workshops, 2003. Proceedings* (IEEE), 200–206.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., et al. (2017). "Energy efficiency across programming languages: how do energy, time, and memory relate?," in *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*, 256–267.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., et al. (2021). Ranking programming languages by energy efficiency. *Sci. Comput. Program.* 205 (2021), 102609. doi:10.1016/j.scico.2021.102609

Pinto, G., and Castor, F. (2017). Energy efficiency: a new concern for application software developers. *Commun. ACM* 60 (12), 68–75. doi:10.1145/3154384

Sanford Shapiro, S., and Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). *Biometrika* 52 (3-4), 591–611. doi:10.1093/biomet/52.3-4.591

Santos, A., Cunha, A., Macedo, N., and Lourenço, C. (2016). "A framework for quality assessment of ROS repositories," in *2016 IEEE/RSJ international conference on intelligent robots and systems (IROS)* (IEEE), 4491–4496.

St, L., and Svante, W. (1989). Analysis of variance (ANOVA). *Chemom. intelligent laboratory Syst.* 6 (4), 259–272. doi:10.1016/0169-7439(89)80095-4

Stanford Artificial Intelligence Laboratory (2024). Robotic operating system. Available at: https://www.ros.org.

Steve, C. (2011). Exponential growth of ros [ros topics]. *IEEE Robotics and Automation Mag.* 1 (18), 19–20. doi:10.1109/MRA.2010.940147

Swanborn, S., and Malavolta, I. (2020). "Energy efficiency in robotics software: a systematic literature review," in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering workshops*, 144–151.

Thangadurai, J., Saha, P., Rupanya, K., Naeem, R., Enriquez, A., Scoccia, G. L., et al. (2024). "Electron vs. Web: a comparative analysis of energy and performance in communication apps," in *International conference on the quality of information and communications technology* (Springer), 177–193.

Thomas, M. C. (2008). "Software quality metrics to identify risk," in *Department of homel and security software assurance working group*.

von Kistowski, J., Block, H., Beckett, J., Spradling, C., Lange, K.-D., and Kounev, S. (2016). "Variations in CPU power consumption," in *Proceedings of the 7th ACM/SPEC on international Conference on performance engineering (delft, The Netherlands) (ICPE '16)* (New York, NY, USA: Association for Computing Machinery), 147–158. doi:10.1145/2851553.2851567

Yuan, Ye, Shi, J., Zhang, Z., Chen, K., Zhang, J., Stoico, V., et al. (2024). "The impact of knowledge distillation on the energy consumption and runtime efficiency of NLP models," in *Proceedings of the IEEE/ACM 3rd international Conference on AI engineering - software Engineering for AI (lisbon, Portugal) (CAIN '24)* (New York, NY, USA: Association for Computing Machinery), 129–133. doi:10.1145/3644815.3644966

Zhang, H., and Hoffman, H. (2015). A quantitative evaluation of the RAPL power control system. *Feedback Comput.* 6.

TYPE Original Research
PUBLISHED 20 May 2025
DOI 10.3389/frobt.2025.1531743

Check for updates

OPEN ACCESS

EDITED BY
Federico Ciccozzi,
Mälardalen University, Sweden

REVIEWED BY
Nico Hochgeschwender,
University of Bremen, Germany
Davide Di Ruscio,
University of L'Aquila, Italy

*CORRESPONDENCE
Gustavo Rezende Silva,
g.rezendesilva@tudelft.nl

RECEIVED 20 November 2024
ACCEPTED 04 April 2025
PUBLISHED 20 May 2025

CITATION
Rezende Silva G, Päßler J, Tapia Tarifa SL, Johnsen EB and Hernández Corbato C (2025) ROSA: a knowledge-based solution for robot self-adaptation.
*Front. Robot. AI* 12:1531743.
doi: 10.3389/frobt.2025.1531743

# ROSA: a knowledge-based solution for robot self-adaptation

Gustavo Rezende Silva[1]*, Juliane Päßler[2], S. Lizeth Tapia Tarifa[2], Einar Broch Johnsen[2] and Carlos Hernández Corbato[1]

[1]Cognitive Robotics Department, Mechanical Engineering Faculty, TU Delft, Delft, Netherlands, [2]Department of Informatics, University of Oslo, Oslo, Norway

Autonomous robots must operate in diverse environments and handle multiple tasks despite uncertainties. This creates challenges in designing software architectures and task decision-making algorithms, as different contexts may require distinct task logic and architectural configurations. To address this, robotic systems can be designed as self-adaptive systems capable of adapting their task execution and software architecture at runtime based on their context. This paper introduces ROSA, a novel knowledge-based framework for RObot Self-Adaptation, which enables task-and-architecture co-adaptation (TACA) in robotic systems. ROSA achieves this by providing a knowledge model that captures all application-specific knowledge required for adaptation and by reasoning over this knowledge at runtime to determine when and how adaptation should occur. In addition to a conceptual framework, this work provides an open-source ROS 2-based reference implementation of ROSA and evaluates its feasibility and performance in an underwater robotics application. Experimental results highlight ROSA's advantages in reusability and development effort for designing self-adaptive robotic systems.

KEYWORDS

**self-adaptation, knowledge representation, underwater vehicle, robotics, self-adaptive robotic system**

## 1 Introduction

A current challenge in robotics is designing software architectures and task decision-making algorithms that enable robots to autonomously perform multiple tasks in diverse environments while handling internal and environmental uncertainties. This challenge arises because different contexts may demand distinct task logic and architectural configurations. At runtime, certain actions may become unfeasible, requiring the robot to adapt its task execution to ensure mission completion. For example, a robot navigating through an environment might run out of battery during its operation, requiring it to adapt its task execution to include a recharge action. Additionally, actions may require different architectural configurations depending on the context. For example, a navigation action that relies on vision-based localization cannot be executed in environments without lights but could potentially be executed with an alternative architectural configuration that employs a localization strategy based on lidar. This becomes even more challenging when both the robot's task execution and its architectural configuration need to be adapted. For instance, when a robot runs out of battery while navigating, it must simultaneously adapt its architecture to a configuration that consumes less energy and its task execution to include a recharge action and to navigate along paths that are better suited to the new configuration.

To address this challenge, robots can be designed as self-adaptive systems (SASs) with the ability to perform *task-and-architecture co-adaptation* (TACA) (Cámara et al., 2020), i.e., simultaneously adapt their task execution and software architecture dependently during runtime. This work focuses on proposing a systematic solution for enabling TACA that can be reused with different robotic systems.

A common approach to enable self-adaptation in software systems is to design them as two-layered systems containing a managing and managed subsystem (Weyns, 2020), where the managing subsystem monitors and reconfigures the managed subsystem, and the managed subsystem is responsible for the domain logic. This design facilitates the development and maintenance of the system by creating a clear separation between the adaptation and the domain logic. While several solutions have been proposed for solving either architectural (Alberts et al., 2025) or task adaptation in robotic systems (Carreno et al., 2021; Hamilton et al., 2022), there are some works that partially address TACA (Park et al., 2012; Lotz et al., 2013; Gherardi and Hochgeschwender, 2015; Valner et al., 2022), and there are few works that fully address TACA (Braberman et al., 2017; Cámara et al., 2020). More critically, to the best of our knowledge, the existing solutions for TACA require a significant and complex re-programming of the adaptation logic for each different use case, including the creation of multiple modelsbased on different domain-specific languages (DSLs) (Cámara et al., 2020) or implementing the managing subsystem itself (Braberman et al., 2017), hindering the adoption of SAS methods in robotics.

To address the limitations of SAS methods for TACA, this paper proposes to extend traditional robotics architectures with a novel knowledge-based managing subsystem for RObot Self-Adaptation (ROSA) that promotes reusability, composability, and extensibility. The main novelty of ROSA is its knowledge base (KB) which captures knowledge about the actions the robot can perform, the robot's architecture, the relationship between both, and their requirements to answer questions such as “What actions can the robot perform in situation X?” and “What is the best configuration available for each action in situation Y?”, for example, “Can the robot perform an inspection action when the battery level is lower than 50%?” or “What is the best software configuration for the inspection action when the visibility is low?” This results in a reusable solution for TACA in which all application-specific aspects of the adaptation logic are captured in its KB.

In addition to a conceptual framework, this work provides a reference implementation of ROSA as an open-source framework that can be reused for research on self-adaptive robotic systems. ROSA is implemented as a ROS 2-based system (Macenski et al., 2022), leveraging *TypeDB* (Dorn and Pribadi, 2023; 2024) for knowledge representation and reasoning, and *behavior trees* (BT) (Colledanchise and Ögren, 2018) as well as *PDDL-based planners* (Ghallab et al., 1998) for task decision-making.

The *feasibility* of using ROSA for runtime self-adaptation in robotic systems is demonstrated by applying it to the SUAVE exemplar (Silva et al., 2023), and its adaptation *performance* is evaluated in comparison to other approaches available in the exemplar. ROSA's *reusability* is demonstrated by using it to model the TACA scenarios described by Braberman et al. (2017) and Cámara et al. (2020). The *development effort* for using ROSA is evaluated by analyzing the number of elements contained in the knowledge models created to solve the aforementioned use cases and comparing it with the size of a BT-based approach used to solve SUAVE. ROSA's *development effort scalability* is demonstrated by showing how ROSA's knowledge model grows with the addition of extra adaptations in a hypothetical scenario.

In summary, the main contributions of this paper are:

1. a *modular architecture* for self-adaptive robotic systems that extends robotics architectures with a managing subsystem and supports reusability, composability, and extensibility;
2. a *reusable knowledge model* to capture all application-specific aspects of the adaptation logic required for TACA in self-adaptive robotic systems;
3. a reference *open-source implementation* of the framework that can be reused for self-adaptive systems research; and
4. an *experimental evaluation* of ROSA based on simulated robotic self-adaptation scenarios.

The remainder of this paper is organized as follows. Section 2 describes the TACA use case used to exemplify and evaluate this work. Section 3 presents related works. Section 4 describes how this work proposes to extend robotics architectures with ROSA. Section 5 details ROSA's KB. Section 6 describes the proposed reference implementation of ROSA. Section 7 showcases ROSA's evaluation. Section 8 concludes this work and presents future research directions.

## 2 Running example

Throughout this paper, the SUAVE exemplar (Silva et al., 2023) is used as an example to ease the understanding of the proposed solution, and later, it is used to evaluate ROSA.

SUAVE consists of an Autonomous Underwater Vehicle (AUV) used for underwater pipeline inspection. The AUV's mission consists of performing the following actions in sequence: *(A1) searching for the pipeline* and *(A2) simultaneously following and inspecting the pipeline*. When performing its mission, the AUV is subject to two uncertainties: *(U1) thruster failures*, and *(U2) changes in water visibility*. These uncertainties are triggers for parameter and structural adaptation. When *U1* occurs while performing *A1* or *A2*, the AUV activates a functionality to recover its thrusters. When *U2* happens while performing *A1*, the AUV adapts its search altitude.

To demonstrate TACA, this work extends SUAVE with an *(A3) recharge battery* action and a *(U3) battery level* uncertainty. With these extensions, the AUV's battery level can suddenly drop to a critical level, requiring the AUV to abort the action it is performing *(A1 or A2)* and perform *A3*. In this situation, the AUV needs to perform TACA by adapting its task execution and architecture to perform *A3*. To better evaluate ROSA by serving as a baseline for comparison, this work extends the SUAVE exemplar with a managing subsystem where the adaptation logic is implemented with BTs, and the AUV's architectural variants as well as the architectural adaptation execution are realized with System Modes (Nordmann et al., 2021)[1]. Furthermore, this work introduces a new

1 The SUAVE exemplar is already configured to use System Modes.

*reaction time* metric that represents the time a managing system takes to react to uncertainties and adapt the managed subsystem.

## 3 Related work

This work combines principles from self-adaptive systems and knowledge representation and reasoning to design a reusable framework for developing adaptive robotics architectures. Section 3.1 analyzes existing robotics architectures and describes the architectural patterns from robotics architectures adopted in this work. Section 3.2 reviews related research on self-adaptive robotic systems that leverage knowledge representation techniques to promote reusability, as well as studies that consider the relationship between task execution and architectural adaptation. Additionally, it discusses how these works influenced the design of the proposed framework and highlights its distinctions from existing approaches.

### 3.1 Robotics architectures

Numerous approaches have been proposed for programming and designing autonomous robot architectures (Kortenkamp et al., 2016). In recent years, two main trends have emerged: component-based frameworks and middlewares–among which ROS (Macenski et al., 2022) stands out due to its widespread adoption in academia and industry–and layered architectures (Barnett et al., 2022). Barnett et al. (2022) reviewed 21 robotics architectures and concluded that most architectures follow a layered pattern, and even those that do not can still have their elements mapped onto a layered architectural structure. Furthermore, they found that all architectures include a bottom functional layer responsible for interacting with the robot's hardware, an upper task decision layer–whose responsibilities vary across architectures–and an arbitrary number of intermediate layers. This work aims to design a reusable solution for TACA that can be integrated into robotics architectures adhering to these architectural patterns. To achieve this, the proposed solution establishes a clear separation between architectural management and task logic, organizing them into distinct layers, or subsystems, as commonly referred to in the self-adaptive systems community.

The LAAS architecture (Alami et al., 1998) is an example of a three-layered architecture consisting of a functional layer, an executive, and a decision layer. The functional layer contains the robot's control and perception algorithms. The executive layer receives a task plan from the decision layer and selects functions from the functional layer to realize each action in the task plan. The decision layer includes a planner that generates task plans and a supervisor responsible for monitoring plan execution and triggering replanning when necessary. More recent examples of layered robot architectures include AEROSTACK (Sanchez-Lopez et al., 2016), designed for aerial drone swarms, and SERA (Garcia et al., 2018), which is tailored for decentralized and collaborative robots. These architectures build on the layered model but focus on providing domain-specific solutions.

Cognitive architectures (Kotseruba and Tsotsos, 2018), such as CRAM (Beetz et al., 2010; Kazhoyan et al., 2021), focus on generating intelligent and flexible behavior by integrating cognitive capabilities such as planning, perception, or reasoning. However, being integral solutions, these works provide a blueprint for the complete robot control system and are not intended for reuse and integration with other methods, thus making it difficult to adapt and customize for specific applications.

### 3.2 Self-adaptive robotic systems

Despite advances in self-adaptive robotic systems, fully addressing task and architectural co-adaptation (TACA) with reusable and scalable solutions remains an open challenge (see Table 1). While some studies explore the relationship between task execution and architectural adaptation, only a few explicitly address TACA—and those that do face limitations in reusability and practical applicability in robotics. This work aims to bridge this gap by introducing a knowledge-based framework that can capture the necessary knowledge to solve TACA across different use cases, can be directly applied to robotic systems, and supports modular modifications for incorporating different adaptation strategies.

Alberts et al. (2025) recently conducted a systematic mapping study[2] on "robotics software architecture-based self-adaptive systems" (RSASSs), identifying 37 primary studies on RSASSs published since 2011. Among these, Alberts et al. (2025) identified that four studies (Park et al., 2012; Lotz et al., 2013; Gherardi and Hochgeschwender, 2015; Cámara et al., 2020) consider, to varying degrees, the relationship between the tasks a robot performs and architectural adaptation. A non-systematic snowballing of the primary studies identified by Alberts et al. (2025) revealed two additional studies (Braberman et al., 2017; Valner et al., 2022) that also explore this relationship.

In the context of knowledge-based methods, Alberts et al. (2025) identified six studies (Park et al., 2012; Niemczyk and Geihs, 2015; Hochgeschwender et al., 2016; Niemczyk et al., 2017; Bozhinoski and Wijkhuizen, 2021; Silva et al., 2023) that use knowledge representation techniques to capture knowledge required for the adaptation logic. Among these (Bozhinoski and Wijkhuizen, 2021; Silva et al., 2023), do not propose solutions for RSASSs but instead demonstrate the application of Metacontrol (Hernández et al., 2018; Bozhinoski et al., 2022) in different robotics use cases. While these methods do not address TACA, they provide valuable insights for designing knowledge-based approaches to self-adaptation.

Park et al. (2012) introduced the SHAGE framework for task-based and resource-aware architecture adaptation in robotic systems. SHAGE partially solves TACA, as it can adapt the robot's architecture at runtime to specifically realize each action in its task plan when it needs to be performed. However, SHAGE does not support task execution adaptation based on the robot's architectural state. SHAGE promotes reusability by leveraging architectural models and knowledge captured with an ontology to reason about adaptation at runtime. However, to the best of our knowledge, there is no implementation of the SHAGE framework that works with

2 Alberts et al. (2025) do not claim that the mapping study is a complete overview of the literature but rather a characterization of the field.

TABLE 1 Related frameworks for robot self-adaptation.

| Approach | Architectural adaptation | | TACA | | Reusability | | Robotics Middleware | Applied to robot |
|---|---|---|---|---|---|---|---|---|
| | Parameter | Structural | A-t-T | T-t-A | Conceptual | Software Available | | |
| ICE (Niemczyk et al. 2017) | No | Yes | No | No | No | No | None | No |
| Hochgeschwender et al. (2016) | No | Yes | No | No | No | No | None | Real robot |
| Metacontrol (Bozhinoski et al., 2022) | Yes | Yes | No | No | Yes | Yes | ROS 1 and 2 | Real robot |
| SHAGE (Park et al., 2012) | No | Yes | No | Yes | Partially | No | None | No |
| Lotz et al. (2013) | No | Yes | No | Yes | No | No | None | No |
| RRA (Gherardi and Hochgeschwender, 2015) | Yes | Yes | No | Partially | Yes | Yes | ROS 1 | Simulated |
| TeMoto (Valner et al., 2022) | Yes | Yes | No | Yes | Partially | Yes | ROS 1 | Real robot |
| MORPH (Braberman et al., 2017) | Yes | Yes | Yes | Yes | No | No | None | No |
| Cámara et al. (2020) | Yes | Yes | Yes | Yes | Partially | No | ROS 1 | Simulated |
| **ROSA** | **Yes** | **Yes** | **Yes** | **Yes** | **Yes** | **Yes** | **ROS 2** | **Simulated** |

Where A-t-T means "Architectural state triggers task adaptation", and T-t-A means "Task triggers architectural adaptation".

common robotic frameworks. Thus, it is not possible to directly reuse SHAGE.

Lotz et al. (2013) proposed a method to model operational and quality variability using two distinct (DSLs) models. Their work provides a high-level discussion on how these models could be used at runtime to enable architectural adaptation based on the actions executed by the robot. An interesting aspect of their approach is the clear separation between functional and non-functional requirements: one model captures the task deliberation logic, functional requirements, and their variation points, while the other focuses on non-functional requirements and their possible variations. While this separation of concerns simplifies the modeling process, combining task deliberation with functional requirements reduces reusability. Any change in the task deliberation logic directly impacts the modeling of functional requirements, making the approach less flexible. Additionally, they do not provide sufficient details on how these models are used at runtime, nor do they present an evaluation to demonstrate the feasibility of their approach.

Gherardi and Hochgeschwender (2015) proposed RRA as a model-based approach for structural, parameter, and connection adaptation in robotic systems. Their method employs six distinct models to capture all the knowledge required for adaptation. These models represent the robot's architecture and its variability, its functionalities and their variability, the mapping between functions and architecture, the required interfaces (i.e., inputs, outputs, and data types) for the adaptation logic, and the adaptation logic itself. RRA models the dependencies between the tasks a robot can perform and the architectural configurations needed to accomplish them. This is achieved by decomposing each task into multiple functionalities and capturing the available architectural variants for realizing each function. At deployment time, the robot's operator selects a task, and RRA manages only the functionalities required for that task. Although RRA considers the relationship between tasks and architecture to some extent, it cannot be classified as TACA, as this dependence is only accounted for at deployment time. Architectural adaptation occurs based on the selected task rather than dynamically at runtime in response to the individual actions the robot needs to perform. Moreover, RRA does not adapt the task execution based on the robot's architectural state.

Valner et al. (2022) proposed the TeMoto as a general solution for robotic systems' dynamic task and resource management. TeMoto partially solves TACA, as it can adapt the robot's architecture to realize the actions being performed by the robot. TeMoto does not completely fulfill TACA as it cannot adapt the task execution given the robot's architectural state. TeMoto provides reusable mechanisms for resource management, but reusability is limited since the adaptation logic must be implemented for all managed resources, and the knowledge about the dependencies between actions and architecture are programmatically included in the actions' code.

Braberman et al. (2017) proposed MORPH as a reference architecture to enable TACA. They showcased on a conceptual level how MORPH can be applied to enable TACA in an unmanned aerial vehicle (UAV) use case. However, since MORPH is only demonstrated at a conceptual level, it is hard to evaluate the feasibility of applying MORPH to robotic systems at runtime. To the best of our knowledge, there is no framework implementing the complete MORPH architecture. Therefore, it is not possible to directly reuse MORPH.

In the context of TACA, Cámara et al. (2020) developed a method for finding *optimal* task and reconfiguration plans for an autonomous ground vehicle (AGV) navigating in a graph-like environment. To enable optimal planning within reasonable time limits, their method first reduces the search space by finding all possible reconfiguration plans and then computing the shortest N paths the robot can take to reach its goal. Then, it uses this information along with task-specific models that capture mission quality attributes (e.g., energy consumption, collision probabilities) and a preferred utility to apply model checking and determine an optimal reconfiguration plan for each path. Finally, an optimization function selects the best plan based on a predefined utility function (e.g., minimizing energy consumption, time, or collision probability). Although their approach reduces the planning search space to improve planning time, their experiments show that solving the navigation use case still takes an average of 15.1 s, an impractical duration for robots that frequently need to replan at runtime to handle uncertainties. Additionally, while the approach is model-based, it relies on task-specific model transformations (e.g., converting the map or battery model into PRISM model snippets), which require dedicated implementations for different tasks. This limits the reusability of their approach for different types of tasks, as it requires a considerable amount of development effort.

Hochgeschwender et al. (2016) argue that robots should have access to and exploit software-related knowledge about how they were engineered to support runtime adaptation. They demonstrate how labeled property graphs (LPGs) can be used to persistently store and compose different domain models specified with (DSL) to enable runtime architectural adaptation. Although their approach is interesting, its reusability is limited as it does not define a knowledge model that can be reused for other applications: for each different use case and DSL the roboticist is responsible for creating a translation from the DSL to the corresponding LPG.

Niemczyk and Geihs (2015); Niemczyk et al. (2017) propose ICE as a method for adapting the information processing subsystem in multi-robot systems, specifically by adapting the connections between system components. Their approach uses an ontology to define each component's required inputs and outputs, along with quality-of-service information for each connection. This ontology is then translated into answer set programming (ASP), and an ASP solver determines an optimal configuration. While they conceptually demonstrate how their method could be applied to robots, they do not demonstrate it with a robotic system. Moreover, their approach focuses solely on structural and connection adaptation to maintain the functionality of the information processing subsystem, without considering other subsystems of the robotic system.

Hernández et al. (2018); Bozhinoski et al. (2022) proposed Metacontrol as a knowledge-based solution for parameter and structural adaptation. Metacontrol leverages the TOMASys (Hernández et al., 2018) ontology to capture the knowledge required for the adaptation logic and to reason at runtime to decide when and how the system should adapt. Similar to RRA, Metacontrol decomposes the system into functionalities, using TOMASys to represent the robot's functionalities, the architectural variants that implement each functionality, and the non-functional requirements associated with these variants. However, Metacontrol cannot perform TACA as TOMASys does not capture the relationship between the system functionalities and the robot's actions.

In conclusion, existing works that fully address TACA (Braberman et al., 2017; Cámara et al., 2020) face limitations in reusability and practical applicability in robotics. The authors of MORPH (Braberman et al., 2017) note that no complete system has been developed, and they have not demonstrated its feasibility. While the approach proposed by Cámara et al. (2020) provides the most complete solution to TACA in the literature, it also faces reusability limitations by relying on multiple distinct (DSL) models and requiring task-specific implementations for model transformations. To address these limitations, this work proposes capturing all the knowledge required for adaptation logic in a single knowledge model. This approach requires only one model–conforming to the proposed knowledge model–to be designed for each application. Additionally, an open-source reference implementation of ROSA is provided, enabling researchers to reuse, extend, and build upon the proposed solution.

On the other hand, previous knowledge-based methods for RSASS (Park et al., 2012; Niemczyk and Geihs, 2015; Niemczyk et al., 2017; Hochgeschwender et al., 2016; Hernández et al., 2018; Bozhinoski et al., 2022) do not capture all knowledge required to enable TACA. They either lack the ability to capture the relationship between the robot's actions and architecture (Hochgeschwender et al., 2016; Hernández et al., 2018; Bozhinoski et al., 2022), or the knowledge required to decide when and how the robotic system should adapt (Park et al., 2012). Although these works do not capture all the knowledge required for TACA, they are able to capture, to varying degrees, knowledge that supports RSASSs. Thus, this work takes inspiration from them to design ROSA's knowledge model while addressing their limitations. More concretely, ROSA's knowledge model is designed to capture the relationship between the robot's actions and architecture and the knowledge required to decide when and how TACA should be performed.

## 4 Architecture

To enable TACA, this paper proposes to extend traditional robotics architectures with ROSA, using it as a managing subsystem for the robotic subsystem (see Figure 1). This section first describes the assumptions made about the robotics architecture and the requirements to use it alongside ROSA, and then it details ROSA's architecture.

### 4.1 Robotics architecture

This work assumes that the robotics architecture is layered, containing a bottom functional layer, an upper task decision

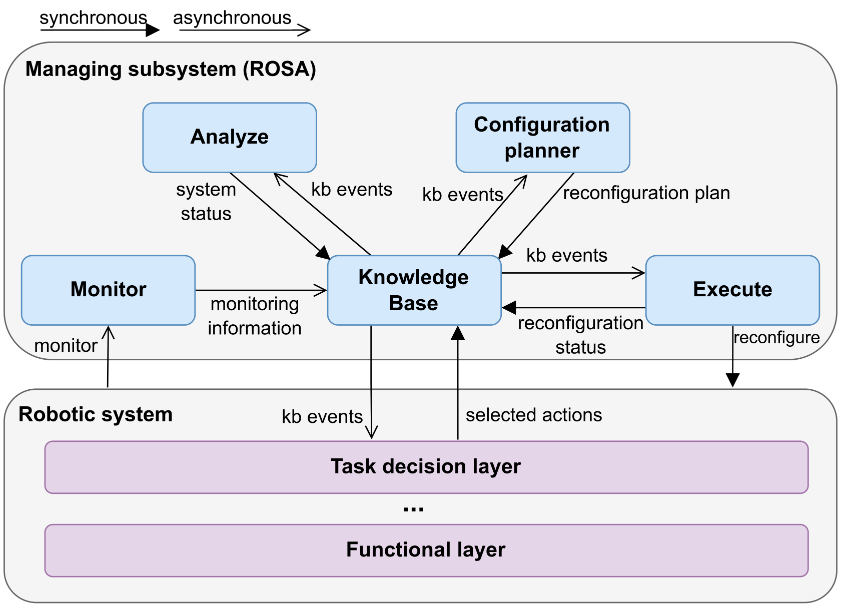


FIGURE 1
The upper layer depicts ROSA's architecture and the bottom layer depicts the robotic system.

layer, and an arbitrary number of layers in between, as common in robotics architectures (Barnett et al., 2022). The functional layer is responsible for interacting with the robots' sensors and actuators, and the task decision layer is responsible for task planning and execution[3]. To enable TACA with ROSA, the task decision layer shall use the knowledge contained in ROSA's KB to decide which actions to perform, and it must update the KB with the actions selected to be performed to enable ROSA to configure the robot's architecture accordingly. To enable architectural adaptation, the robotic architecture must be component-based, its components must be able to be activated and deactivated at runtime, and its components' parameters must be able to be adapted at runtime.

### 4.2 ROSA architecture

ROSA's architecture adheres to the MAPE-K loop (Kephart and Chess, 2003). It *monitors* the managed subsystem, *analyzes* whether adaptation is required, when needed, *plans* how the managed subsystem should be reconfigured, and *executes* the selected reconfigurations. All these steps interact with a central *KB*.

To promote reusability, composability, and extensibility, the architecture is designed with the following premises: *(1) all* knowledge required for the adaptation logic is captured in the central KB, *(2)* there is no inter-component communication between the MAPE components (Weyns et al., 2013), *(3)* the MAPE components insert and read data from or to the KB via standardized interfaces, and *(4)* there is no explicit coordination between the MAPE-K components. Premise *1* promotes reusability by only requiring the modeling of the relevant knowledge for applying ROSA to different applications in a single model. Premises *1* to *4* promote composability and extensibility by allowing the MAPE components to be stateless and self-contained.

## 5 Knowledge base

To fulfill the architectural premise that all knowledge required for the adaption logic should be captured in a central KB, this work proposes a KB component composed of the knowledge model depicted in Figure 2 and the set of rules depicted in Figure 3, described in Section 5.1 and Section 5.2 respectively.

The knowledge model is presented as a conceptual data model (CDM) conforming to a particular case of the enhanced entity-relationship (EER) (Thalheim, 1993; Thalheim, 2000) model, an extension of the entity-relationship model (Chen, 1976) that accounts for subclassing and higher-order relationships, i.e., relations between relationships. The EER model captures information as entities, relationships, and attributes. An entity is a "thing' which can be distinctly identified" (Chen, 1976), a relationship is an association among entities or relationships, and an attribute represents a property of an entity or relationship (Thalheim, 1993; Thalheim, 2000). In addition, entities and relationships play a role in the relationships they are part of, which is identified by the label on the arrows in Figure 2. This CDM was selected since it supports n-ary relationships, many-to-many relationships, higher-order relationships, and attributes for both entities and relationships. Section 6.1 details why these representation capabilities are relevant to the proposed model. The rules are also described at a conceptual level as decision diagrams. Section 6.2 presents the details on how

3 Some architecture have distinct layers for handling task planning and execution, in the context of this work, they can be considered as sub-layers of the task decision layer.

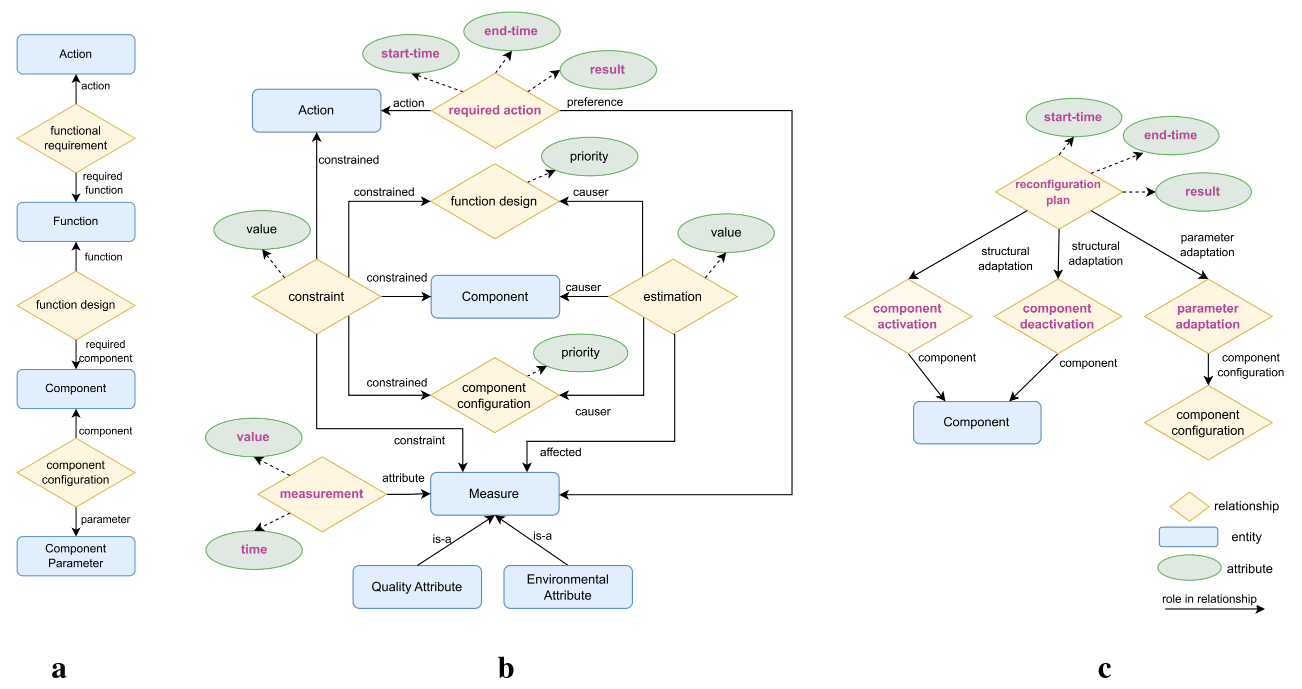


FIGURE 2
ROSA's knowledge model. Architectural knowledge is on the left. Adaptation heuristic knowledge in the center. Reconfiguration Plan knowledge on the right. The labels on the arrows represent which role an entity or relationship plays in a relationship. Instances of the entities, relationships, and attributes with purple font are created at runtime. Instances of the other entities, relationships, and attributes are defined at design time. **(a)** Architectural. **(b)** Adaptation heuristic. **(c)** Reconfiguration plan.

the knowledge model and rules can be implemented and executed and runtime.

## 5.1 Knowledge model

To enable adaptation, the knowledge model captures *what* can be adapted with the architectural knowledge depicted in Figure 2a; *why* to adapt and *how* to select an adaptation with the adaptation heuristics knowledge depicted in Figure 2b; and *how* to execute an adaptation with the reconfiguration plan knowledge depicted in Figure 2c. Tables 2–4 define each element in the model alongside examples based on SUAVE to ease its understanding.

### 5.1.1 Architectural knowledge

The architectural knowledge (see Table 2) captures what actions the robot can accomplish, the set of functionalities the robot needs to realize an action, the set of components required to realize a functionality, and the possible parameters for a component.

The architectural knowledge enables parameter adaptation by capturing each parameter configuration of a component with a distinct `component configuration` relationship, relating one `Component` to a set of `Component Parameters`. It enables structural adaptation by capturing the different possibilities for solving a system functionality as distinct `function design` relationships which relate a `Function` to a set of `Components`. It enables TACA by indirectly capturing the dependencies between the robot's actions and architecture with the `functional requirement` relationship which relates an `Action` to the `Functions` it requires.

At runtime, ROSA performs parameter adaptation by switching the selected `component configurations`. It performs structural adaptation by changing the selected `function designs` and consequently the active `Components`. The task decision layer in combination with ROSA performs TACA by selecting suitable `function designs` and `component configurations` for each `Action` the robot needs to perform, and with the task decision layer selecting different `Actions` to perform according to the feasible configurations.

### 5.1.2 Adaptation heuristic knowledge

This work considers that the robotic system might need to adapt due to changes in the environment, changes in the system's quality attributes (QAs) (Board, 2023), component failures, and changes in the robot's selected actions.

The adaptation heuristic knowledge enables adaptation due to changes in the environment or the system's QAs with the `constraint` relationship by capturing constraints on the selection of `Actions`, `Components`, `function designs`, or `component configurations` in terms of measured values of `Measures`. It enables adaptation due to component failures by capturing a `Component's` status as an attribute and adaptation due to changes in the robot's task execution by capturing what `Actions` need to be performed with the `required`

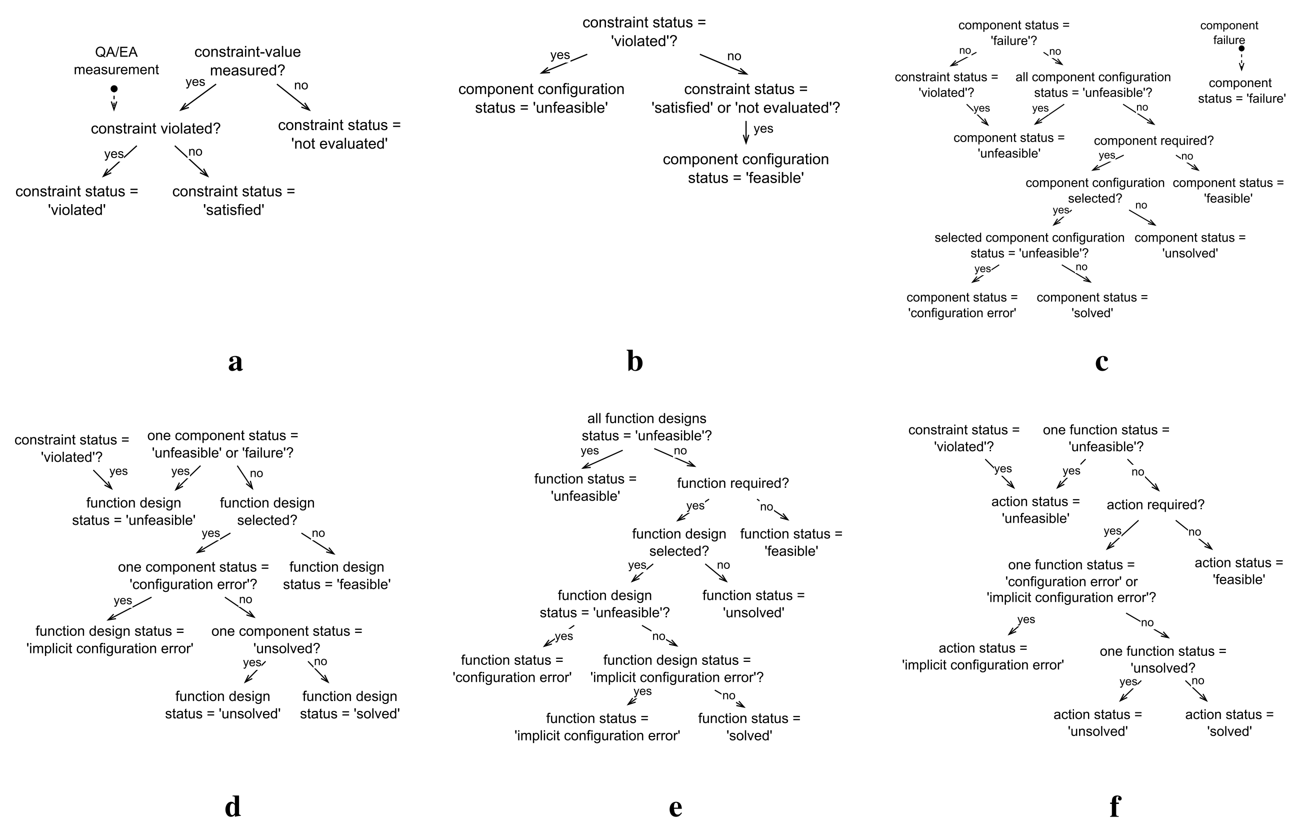

QA/EA measurement
constraint-value measured?
yes
no
constraint violated?
constraint status = 'not evaluated'
yes
no
constraint status = 'violated'
constraint status = 'satisfied'
a
constraint status = 'violated'?
yes
no
component configuration status = 'unfeasible'
constraint status = 'satisfied' or 'not evaluated'?
yes
component configuration status = 'feasible'
b
component status = 'failure'?
no
no
constraint status = 'violated'?
all component configuration status = 'unfeasible'?
component failure
component status = 'failure'
yes
yes
no
component status = 'unfeasible'
component required?
yes
no
component configuration selected?
component status = 'feasible'
yes
no
selected component configuration status = 'unfeasible'?
component status = 'unsolved'
yes
no
component status = 'configuration error'
component status = 'solved'
c
constraint status = 'violated'?
one component status = 'unfeasible' or 'failure'?
yes
yes
no
function design status = 'unfeasible'
function design selected?
yes
no
one component status = 'configuration error'?
function design status = 'feasible'
yes
no
function design status = 'implicit configuration error'
one component status = 'unsolved'?
yes
no
function design status = 'unsolved'
function design status = 'solved'
d
all function designs status = 'unfeasible'?
yes
no
function status = 'unfeasible'
function required?
yes
no
function design selected?
function status = 'feasible'
yes
no
function design status = 'unfeasible'?
function status = 'unsolved'
yes
no
function status = 'configuration error'
function design status = 'implicit configuration error'?
yes
no
function status = 'implicit configuration error'
function status = 'solved'
e
constraint status = 'violated'?
one function status = 'unfeasible'?
yes
yes
no
action status = 'unfeasible'
action required?
yes
no
one function status = 'configuration error' or 'implicit configuration error'?
action status = 'feasible'
yes
no
action status = 'implicit configuration error'
one function status = 'unsolved'?
yes
no
action status = 'unsolved'
action status = 'solved'
f

FIGURE 3
Rules used to infer the status of the elements in the knowledge model, presented as decision diagrams. **(a)** Constraint status. **(b)** Component configuration status. **(c)** Component status. **(d)** Function design status. **(e)** Function status. **(f)** Action status.

`action` relationship. At runtime, adaptation is triggered when a measurement violates a constraint, when a component has a failure status, or when required actions change.

To capture the decision criteria on how to select an adaptation, the `priority` attribute can be used to express the order of priority for selecting each `function design` or `component configuration`. For more complex criteria, the `estimation` relationship can be used to capture the estimated impact of selecting a `function design`, `Component`, or `component configuration` on the measured values of `Measures`. Furthermore, a `required action` can relate to a `Measure` to indicate the preferred `estimation` when selecting a configuration for that action. At runtime, the configuration planner component exploits this knowledge to decide which configuration to select.

### 5.1.3 Reconfiguration plan knowledge

The reconfiguration plan knowledge represents which component parameters must be updated (i.e., how to execute parameter adaptation) and which components must be activated and deactivated (i.e., how to execute structural adaptation). The execute component exploits this knowledge at runtime to reconfigure the managed subsystem.

## 5.2 Rules

To enable ROSA to reason about when the managed subsystem should be adapted and what type of adaptation is needed, the KB contains a set of generic rules that define how the status of the system can be inferred based on monitored information, e.g., how to infer if a constraint is violated and how to propagate a constraint violation. A change in status only occurs when measurements are updated or when a component fails. The complete status inference rules are depicted as decision diagrams in Figure 3.

The status of the required `Components, Functions`, and `Actions` indicate when and what type of adaptation is needed. When a `Component` is "unsolved" or in "configuration error" (see Figure 3c), parameter adaptation is required, i.e., a new `component configuration` needs to be selected. When a required `Function` is "unsolved" or in "configuration error" (see Figure 3e), structural adaptation is required, i.e., a new `function design` needs to be selected. When a required `Action` is "unfeasible" (see Figure 3f), TACA is needed, i.e., the task decision layer must choose a new action to perform, consequently triggering architectural adaptation.

TABLE 2 The elements of the architectural knowledge.

| | Name | Definition | Example | Attributes |
|---|---|---|---|---|
| E | Action | "Action is defined as an operation applied by an agent or team to affect a change in or maintain either an agent's state(s), the environment, or both" (IEEE Approved Draft Standard for Robot Task Representation, 2024). Where in the context of this paper the agent is the robot | An AUV can perform the `Action` *search pipeline*, and *inspect pipeline* | `name:String @key` `status:String` `is-required:Bool` |
| | Function | "A function is defined by the transformation of input flows to output flows" (Board, 2023), i.e., a function represents what the system can do | An AUV can have several `Functions`, like *generate search path* and *generate path to follow pipeline*. The *generate search path* function can be considered a transformation of the AUV's current position (input) to a goal waypoint (output) | `name:String @key` `always-improve:Bool` `status:String` `is-required:Bool` |
| | Component | Hardware and software parts that compose the system | A *thruster* is a hardware component, and a *generate spiral search path node* is a software component | `name:String @key` `always-improve:Bool` `status:String` `is-required:Bool` `is-active:Bool` `pid:Integer` |
| | Component Parameter | A specific parameter configuration of a Component | The *generate spiral search path node* component can be configured with different search altitudes. Each search altitude is represented as a distinct `Component Parameter` | `key:String` `value:String` |
| R | funcional requirement | Represents which `Functions` a certain `Action` requires to be performed | The *search pipeline* action requires the *control motion*, *maintain motion*, *localization*, *detect pipeline*, *generate search path*, and *coordinate mission* functions | N/A |
| | function design | Represents a solution for a `Function` as a set of `Components` (Hernández et al., 2018) | The *generate search path* function can have multiple `function designs`, e.g., one using the *generate spiral search path node* component, and another one using the *generate lawnmower search path node* component | `name:String @key` `priority:Integer` `status:String` `is-selected:Bool` |
| | component configuration | Represents a possible configuration of a `Component` as a set of `Component Parameters` | The *generate spiral search path node* component can be configured in different ways by combining different values of its search altitude and speed parameters. Each combination is represented as an instance of a `component configuration` | `name:String @key` `priority:Integer` `status:String` |

Instances of the elements with purple font are created at runtime. Instances of the other elements are defined at design time. The "@key" expression after an attribute indicates that the attribute is used as a unique identifier. E stands for entity and R stands for relationship.

## 6 Architecture realization

This section details the proposed reference implementation for ROSA. First, the representation requirements imposed by the proposed knowledge model are analyzed to select a suitable knowledge representation technique. Afterward, the proposed implementation is explained.

### 6.1 Representation requirements

The proposed knowledge model was presented as a conceptual data model (CDM) in a non-machine-readable format. To transform the knowledge model into a machine-readable format while maintaining its semantics and structure, its representation requirements must be identified and used to select a suitable technology to implement it. The representation requirements imposed by the proposed knowledge model are:

1. n-ary relationships: the knowledge model contains relationships with different arity[4], e.g., the `constraint` relationship has arity 5, and the `measurement` relationship has arity 1;

4 The number of distinct elements that can be part of a relationship.

TABLE 3 The elements of the adaptation heuristic knowledge.

| | Name | Definition | Example | Attributes |
|---|---|---|---|---|
| E | Measure | "Measure is defined as a function over observations, state variables, and parameters". (IEEE Approved Draft Standard for Robot Task Representation, 2024) | An AUV can have the `Measures` *battery level* or *water visibility* | `name:String @key` |
| | Quality Attribute | Quality Attribute is defined as a function over the system's state variables. Which, according to the Software Engineering Body of Knowledge (Board, 2023), can be considered as "System functional and non-functional requirements used to evaluate the system performance" | An AUV can have the `Quality Attributes` *battery level*, *battery consumption*, and *safety level* | `name:String @key` |
| | Environmental Attribute | Environmental Attribute is defined as a function over observations of the environment. That is, it represents a metric of the environment with respect to a certain attribute | An underwater environment can have *water visibility* as an `Environmental Attribute` | `name:String @key` |
| R | required action | Represents the Actions required at runtime | The *inspect pipeline* `Action` can be required to be performed at runtime, leading ROSA to configure the AUV appropriately to carry out the inspection action | `start-time:Datetime` `end-time:Datetime` `result:String` |
| | measurement | "Measurement is defined as the act of evaluating the measures" (IEEE Approved Draft Standard for Robot Task Representation, 2024) | The *battery level* `Quality Attributes` can have a measurement of 0.5 | `value:Double` `time:Datetime` |
| | constraint | Represents `Measure` constraints for performing a `Action` or for selecting a `function design`, `Component`, or `component configuration` | If the *water visibility* environmental attribute is low, the *high altitude* configuration for the *generate spiral search path node* cannot be selected | `operator:String` `value:Double` `status:String` |
| | estimation | Represents the estimated impact of a `function design`, `Component`, or `component configuration` on a `Measure` | A lamp `Component` is expected to positively impact the *water visibility* when turned on | `value:Double` `type:String` |

Instances of the elements with purple font are created at runtime. Instances of the other elements are defined at design time. The "@key" expression after an attribute indicates that the attribute is used as a unique identifier. E stands for entity and R stands for relationship.

2. many-to-many relationships: the knowledge model contains relationships with different cardinalities[5], e.g., `function design` is a 1 `Function`-to-many `Components` relationship;
3. higher-order relationships: some relationships relate relationships to relationships, e.g., the `constraint` relationship;
4. attributes for entities and relationships: both entities and relationships have attributes, e.g., the `Action` entity and the `constraint` relationship.

These requirements limit which technology can be used. For example, technologies using graph-based or descriptive logic-based knowledge representation techniques (e.g., OWL (Antoniou and van Harmelen, 2004)) do not satisfy the representation requirements above apart from allowing attributes for entities (part of Requirement 4). Although it is possible to transform the proposed knowledge model into one that can be represented with graph-based or description logic-based approaches by reifying the model (Olivé, 2007), each reification applied to the original model can be considered as an introduction of a semantic disparity between the CDM and the machine-readable model, making it harder to understand and reuse it. Thus, this work does not consider applying reification.

A technology that satisfies all representation requirements is TypeDB (Dorn and Pribadi, 2023; 2024). TypeDB is a polymorphic database based on type theory that implements the polymorphic entity-relation-attribute (PERA) (Dorn and Pribadi, 2024) data model. The PERA model subsumes the CDM used as the meta-model for the proposed ROSA model, allowing it to be implemented without modifications. Furthermore, TypeDB has a reasoning system that is able to reason over rules of the form `antecedent` ⇒ `consequent` to infer new facts at query time. Where `antecedent` represents a precondition for inferring the

5 The number of element instances that can be part of a relationship.

TABLE 4 The elements of the reconfiguration plan knowledge.

| | Name | Definition | Example | Attributes |
|---|---|---|---|---|
| R | reconfiguration plan | Represents the reconfiguration plan generated in the plan step | When the AUV completes the *search pipeline* action and starts the *inspect pipeline* action, the `reconfiguration plan` consists of deactivating the *generate spiral search path node* and activating the *follow pipeline node* | `start-time:Datetime` `end-time:Datetime` `result:String` |
| | component activation | Represents the components that should be activated | When the AUV starts the *inspect pipeline* action, the `component activation` relates to the *follow pipeline node* | N/A |
| | component deactivation | Represents the components that should be deactivated | When the AUV completes the *search pipeline* action, the `component deactivation` relates to the *generate spiral search path node* | N/A |
| | parameter adaptation | Represents the component parameters that should be updated | When the water visibility changes, the `parameter adaptation` consists of a parameter adaptation of the component configuration of the *generate spiral search path node* | N/A |

Instances of the elements with purple font are created at runtime. Instances of the other elements are defined at design time. R stands for relationship.

`consequent` and is expressed as a first-order logic expression combining elements from the model (i.e., entities, relationships, and individuals), and the `consequent` is a single new fact inferred when the `antecedent` holds true. The ROSA model rules presented in Figure 3 can be implemented with TypeDB without modifications. For these reasons, this work uses TypeDB to implement the proposed knowledge model and rules.

## 6.2 Implementation

ROSA is implemented as a ROS 2-based system, where the MAPE-K components (depicted in Figure 1) are realized as ROS nodes, and interfaces are implemented using ROS services or topics. The proposed ROSA implementation uses ROS (Robot Operating System) as its robotics framework since ROS is the current *de facto* standard robotics framework, and it has been designed, among other things, to promote software reusability in the robotics ecosystem (Macenski et al., 2022). In this implementation, ROS handles the communication between system components, schedules callbacks for incoming messages and events, and manages the lifecycle of ROS nodes. The full ROSA implementation is available at https://github.com/kas-lab/rosa[6].

6 In addition to ROSA, this work provides a generic ROS package to integrate ROS 2 with TypeDB: https://github.com/kas-lab/ros_typedb.

### 6.2.1 Knowledge base and analyze

The KB component consists of the TypeDB implementation of the proposed knowledge model and inference rules, the ROS interfaces for communicating with the MAPE components, and the logic to manage ROSA's knowledge which is stored in a TypeDB database. In this reference ROSA implementation, TypeDB's reasoner fulfills the role of the analyze component, executing ROSA's inference rules (Figure 3) to infer new data when the KB is queried. Thus, since TypeDB's reasoner is part of TypeDB, there is no separate analyze component.

#### 6.2.1.1 Knowledge model and rules

To exemplify how the knowledge model is implemented, Listing 1 depicts how the `functional-requirement` relationship and the `Action` entity are defined with TypeQL (TypeDB's query language). Line 1 defines the `functional-requirement` relationship, and lines 2-3 define that it can relate elements that play the role of actions and required-functions. Lines 4-6 define the `Action` entity and that it has the attributes "action-name" (its unique identifier) and "action-status". Lines 7-8 define that it can play the action role in a `functional-requirement` relationship and the constrained role in a `constraint` relationship.

Listing 2 exemplifies how the inference rules are implemented with TypeQL. The component-status-configuration-error rule defines that a Component has a "configuration error" status when it is required, it does not have an "unfeasible" or "failure" status, and

```
functional-requirement sub relation,
 relates  action,
 relates required-function;
Action sub entity,
 owns action-name @key,
 owns action-status,
 plays functional-requirement:action,
 plays constraint:constrained;
```

Listing 1. TypeQL query to define functional-requirement and Action.

it is in a component-configuration relationship that is selected and has an "unfeasible" status (see Figure 3c).

```
rule component-status-configuration-error:
 when {
  $c isa  Component, has  is-required true;
  not {
  $c has status $c_status;
  $c_status like "unfeasible|failure";
  } ;
  (component: $c) isa component-configuration,
  has is-selected true,
  has status "unfeasible";
 }then {
  $c has status "configuration error";
 };
```

Listing 2. TypeQL rule to infer whether a component is in "configuration error".

#### 6.2.1.2 Interfaces

The KB component abstracts the details of interacting with TypeDB with the ROS interfaces it implements, enabling the MAPE components to read and write knowledge via the interfaces described in Table 5. When the MAPE components request or send data to the KB component via these interfaces, the KB component queries the TypeDB database to retrieve or write knowledge. For example, when the task decision layer calls the selectable service */action/selectable* to retrieve the name of the selectable Actions (i.e., actions that do not have an "unfeasible" status), the KB component performs the TypeQL query depicted in Listing 3 to retrieve the name (unique identifiers) of the selectable Actions. When data is written in the KB, the KB component publishes a message in the */events* topic specifying which type of data was written, i.e., "monitoring data", "action update", "reconfiguration plan". Additionally, the KB component provides the */query* service, which can be used to perform *any* TypeDB query to the database. It is not used in ROSA's runtime workflow, but it enables users to perform custom queries, for example, to retrieve all reconfiguration plans that were executed.

TABLE 5 ROS interfaces.

| Topics | | |
|---|---|---|
| **Publisher** | **Subscriber** | **Name** |
| KB | Configuration planner | ~/events |
| | Execute | |
| | Task decision layer | |
| Monitor nodes | KB | /diagnostics |
| **Services** | | |
| **Server** | **Client** | **Name** |
| KB | Configuration planner | ~/function/adaptable |
| | | ~/function_ designs/selectable |
| | | ~/function_designs/priority |
| | | ~/component/adaptable |
| | | ~/component_ configuration/selectable |
| | | ~/component_ configuration/priority |
| | | ~/select_configuration |
| | Execute | ~/reconfiguration_ plan/get_latest |
| | | ~/reconfiguration_ plan/result/set |
| | | ~/component/active/set |
| | | ~/component_ parameters/get |
| | Task decision layer | ~/action/selectable |
| | | ~/action/request |
| | User | ~/query |

```
match $a isa Action, has name $name;
 not { $a has status "unfeasible";};
 fetch $name;
```

Listing 3. TypeDB query to fetch selectable actions' names.

### 6.2.2 Monitor

ROSA's implementation does not provide generic monitor nodes. They should be implemented as needed for each application with the requirement that they publish the monitored information in the */diagnostics* topic[7] with the standard ROS *DiagnosticArray* message format. When a monitor node sends measurement updates to the KB, the message field in the *DiagnosticStatus* message needs to be set to "QA measurement" or "EA measurement", and when sending component status updates (e.g., that the component is in failure), the message field must be set to "Component status". When the KB receives monitoring data, it sends an event message in the "/events" topic to inform that monitoring data was written in the KB.

### 6.2.3 Configuration planner

The configuration planner component selects the configurations (i.e., `function designs` or `component configuration`) with the highest priority. When the configuration planner receives an event message indicating that monitoring data was written in the KB or that there was an update in the required actions, it calls the services "/function/adaptable" and "/component/adaptable" to check which `Functions` and `Components` must be adapted. Then, it calls the services "/function/selectable" and "/component/selectable" to check which `function designs` and `component configurations` are available for the `Functions` and `Components` that need to be adapted. Finally, the configuration planner selects the `function designs` and `component configurations` with the highest priority and informs the KB about the newly selected configuration by calling the service "/select_configuration". When this service is called, the KB component checks the current state of the robot, creates a `reconfiguration plan` to bring the robot to the goal configuration, and sends an event message in the "/events" topic to inform that there is a new reconfiguration plan available.

### 6.2.4 Execute

In ROS-based systems, software components are realized either as ROS nodes or as a particular type of ROS nodes called *lifecycle* nodes. The difference between both is that the latter can be set to different states at runtime, such as *active* and *inactive*, and the former cannot. To enable ROSA to leverage ROS 2 mechanisms to adapt the system, the knowledge model was extended to capture knowledge about ROS 2 components as depicted in Figure 4. The execute component performs structural adaptation by starting or killing ROS nodes or switching the state of lifecycle nodes to active or inactive, and it performs parameter adaptation by calling the ROS's parameter API to change the ROS nodes' parameters at runtime.

When the execute component receives an event message indicating that a new reconfiguration plan was added to the KB, it calls the service "/reconfiguration_plan/get_latest" to get the latest `reconfiguration plan`. Then, it adapts the robot's architecture according to the reconfiguration plan.

7 The */diagnostics* topic is a standard topic for publishing system diagnosis information within the ROS ecosystem. See ROS REP 107 for more information.

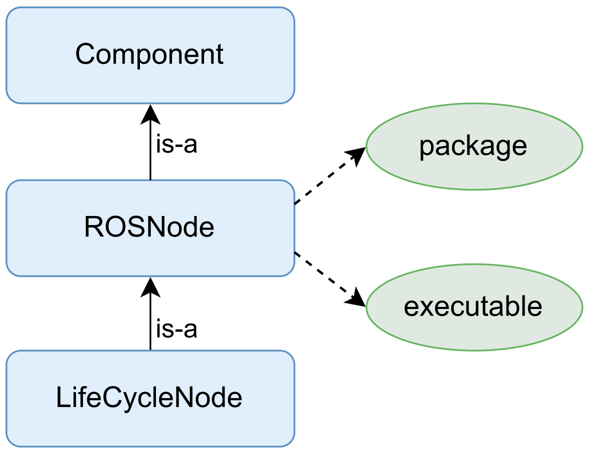


FIGURE 4
ROS specific knowledge.

Finally, it calls the services "/reconfiguration_plan/result/set" and "/component/active/set" to update the KB with the result of the reconfiguration plan and which components are active.

### 6.2.5 Task decision layer

ROSA's implementation provides an integration for the task decision layer for both PDDL-based planners and BTs, which are implemented leveraging the PlanSys2 (Martín et al., 2021) and the BehaviorTree.CPP[8] packages, respectively.

#### 6.2.5.1 Planning

To enable task decision-making and execution with PDDL-based planners in combination with ROSA, the planner and plan executor must consider the runtime feasibility of performing the robot's actions as inferred by the KB component. This work maps the action status from ROSA's knowledge model to PDDL by capturing whether the action's status is feasible as a PDDL predicate of the form "*action_feasible ?action*" and using it as a precondition to select the respective action. An example can be seen in Listing 4 where the action *my_action* can only be selected when it does not have an "unfeasible" status in the KB. At runtime, if an action becomes unfeasible during execution, the plan executor triggers re-planning to generate a new action plan. This results in task execution adaptation and, if the newly selected actions require a different architectural configuration, also in architectural adaptation, i.e., TACA.

To handle the interaction between PlanSys2 and ROSA's KB, this work provides a custom ROS 2 node called *RosaPlanner* and a custom PlanSys2 action called *RosaAction*. The *RosaPlanner* is responsible for querying the KB and updating the PDDL problem formulation with information on whether the ROSA actions are feasible or not using the aforementioned "*action_feasible ?action*" PDDL predicate. The *RosaAction* action is responsible for querying the KB to request or cancel an `Action` when the execution of

8 https://www.behaviortree.dev/

```
(:durative-action my_action
 :parameters (?a - action...)
 :duration (...)
 :condition (and
  (over all(my_action_action ?a))
  (over all(action feasible ?a))
  ...
 )
 :effect (and...)
)
```

Listing 4. PDDL formulation example for ROSA.

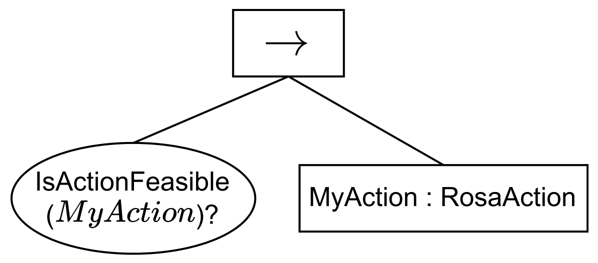


FIGURE 5
BT pattern for TACA with ROSA. The *IsActionFeasible* condition node takes the action name as a parameter. The *MyAction* node derives from the proposed *RosaAction* node and implements the action execution. The action names in the BT must match the names defined in the KB.

an action starts or finishes. Each PlanSys2 action that should be managed by ROSA should derive from *RosaAction*, and it should implement the logic for the specific action execution.

#### 6.2.5.2 Behavior trees

To enable task decision-making and execution with BTs in combination with ROSA, the BTs must consider the runtime feasibility of performing the robot's actions as inferred by the KB component. This work proposes adding before action nodes a condition node that queries the KB to ask whether the following action is feasible. The proposed pattern is depicted in Figure 5, where the action *MyAction* would only be executed when its status in the KB is not "unfeasible".

To enable the use of the BehaviorTree.CPP package to implement BTs for ROSA and abstract away the interactions with the KB, this work implements a reusable custom condition node called *IsActionFeasible* and a custom action node called *RosaAction*. The condition node queries the KB to check whether an `Action` is feasible before selecting it to be executed, and the action node queries the KB to request or cancel an `Action` when the execution of an action starts or finishes, respectively.

## 7 Evaluation

This section evaluates ROSA to answer the following questions:

- *Feasibility*: Is it feasible to use ROSA to enable runtime TACA in ROS 2-based robotic systems?
- *Performance*: How does ROSA perform compared to other managing subsystems for ROS 2-based systems?
- *Reusability*: To what extent can ROSA's knowledge model capture the knowledge required for TACA?
- *Development effort*: What is the development effort of using ROSA for adding adaptation to different robotic systems and how does it compare to other approaches?
- *Development effort scalability*: How does the development effort of using ROSA for adding adaptation to robotic systems scale for more complex systems?

### 7.1 Experimental design

#### 7.1.1 Feasibility

To evaluate the feasibility of applying ROSA at runtime to enable TACA in ROS 2-based robotic systems, it was applied to the SUAVE exemplar described in Section 2. SUAVE was selected since, to the best of our knowledge, it is the only ROS 2-based open-source exemplar for self-adaptive robotic systems.

#### 7.1.2 Performance

To evaluate ROSA's performance, metrics were collected with the SUAVE exemplar. The metrics collected were the ones available in SUAVE, *search time* and *distance of the pipeline inspected*, in addition to the *reaction time* metric introduced in this paper. For the original use case, the experiments were performed with no managing subsystem, with a BT managing system, with Metacontrol[9], and with ROSA. For the extended use case, the experiments were performed with a BT managing system and with ROSA.

#### 7.1.3 Reusability

To evaluate to what extent ROSA's knowledge model can capture the knowledge required for TACA, it was used to model the TACA scenarios presented by Cámara et al. (2020) and Braberman et al. (2017)[10], and we showcase how the captured knowledge can be exploited to enable TACA. The experimental setup for both scenarios is not publicly available. Thus, it was not possible to apply ROSA at runtime to the simulated environments they used.

#### 7.1.4 Development effort

To evaluate the development effort of using ROSA to enable adaptation in robotic systems, we analyze the number of elements contained in the ROSA model created to solve the use cases described in this paper. Furthermore, we compare it to the number of elements modeled with the BT approach to solve the SUAVE use case.

9 Metacontrol is the only managing subsystem packaged with SUAVE. The Metacontrol implementation for this use case is described in detail in the SUAVE paper (Silva et al., 2023).

10 To the best of our knowledge, these are the only two scenarios in the literature in which the authors explicitly claim the need for TACA.

### 7.1.5 Development effort scalability

To analyze how ROSA's development effort scales for more complex use cases, we showcase how many elements must be modeled in ROSA's model to include structural and parameter adaptation in a hypothetical adaptation scenario.

The experimental setup for the *Feasibility* and *Performance* evaluation is open source and reproducible, it can be found at https://github.com/kas-lab/suave_rosa. The models designed to evaluate *Reusability* can be found at https://github.com/kas-lab/rosa_examples.

## 7.2 Feasibility

### 7.2.1 Experimental setup

To solve the adaptation scenarios of the SUAVE exemplar with ROSA, the model depicted in Figure 6 was created.

*Thruster failure* ($U_1$) was solved with structural adaptation by including two possible `function designs` of the *maintain motion* function. *Runtime behavior:* when a thruster fails, the *maintain* function design status is set to unfeasible (see Figure 3d), and it cannot be selected anymore. Then, the *recover* function design is selected, and the *recover thruster node* component is activated. If all thrusters are recovered, the *maintain* function design status becomes feasible and is selected again.

*Changing water visibility* ($U_2$) was addressed with parameter adaptation by including three `component configurations` for the *generate spiral node*, each representing a different altitude for searching for the pipeline. Furthermore, a *water visibility* `constraint` was added to each configuration, representing the minimum water visibility in which the configuration can be used. *Runtime behavior:* if the measured water visibility is higher than 3.25, the component configuration *High* is selected since it has priority number one. If the water visibility drops below 3.25, its constraint status is set to violated (see Figure 3a), and, consequently, its status is set to unfeasible (see Figure 3b). Depending on the water visibility, the component configuration *Medium* or *Low* is then selected. If the water visibility increases again above 3.25, the *High* component configuration status becomes feasible and is selected.

*Critical battery level* ($U_3$), occurring only in the extended SUAVE use case, was solved with TACA by extending the knowledge model with a *recharge* action, and a *battery level* `constraint` to the *search pipeline* and *inspect pipeline* actions, representing the minimum battery level at which they can be selected. *Runtime behavior:* when the battery level drops below 0.25, the status of both the *search pipeline* and *inspect pipeline* actions is set to unfeasible, and the task decision layer cannot select them anymore. Therefore, the task decision layer selects the *recharge* action which also triggers structural adaptation.

Note that extending the solution to solve SUAVE's extended version only required the inclusion of additional knowledge of the *recharge* action and the `constraint` to the *search pipeline* and *inspect pipeline* actions. This demonstrates that an existing application modeled with ROSA can easily be extended to solve additional adaptation scenarios.

To apply ROSA in simulation to the SUAVE exemplar, the knowledge model depicted in Figure 6 was implemented with TypeDB (see an example in Listing 5), and the AUV's mission was implemented with the BT depicted in Figure 7 as well with the PDDL formulation partially shown in Listing 6.

```
# Action search pipeline
$a_search_pipeline isa Action, has action-name
 "search_pipeline";
$f_generate_search_path isa Function, has
function-name
 "generate_search_path";
# functional-requirement relationship
(action: $a_search_pipeline,
required-function: $f_generate_search_path,
required-function: $f_maintain_motion)
isa functional-requirement;
```

**Listing 5. Snippet of SUAVE's use case implementation in TypeDB.**

### 7.2.2 Result

During the mission execution, the AUV was able to overcome all uncertainties: adapting to thruster failures ($U_1$) with structural adaptation, to changing water visibility ($U_2$) with parameter adaptation, and to an unexpected drop in the battery level ($U_3$) with TACA, demonstrating the feasibility of using ROSA to enable runtime TACA in ROS 2-based robotic systems.

```
(:durative-action search_pipeline
 :parameters (?a - action ?p - pipeline ?r -
   robot)
 :condition (and
  (over all(robot_started ?r))
  (over all(search_pipeline_action ?a))
  (over all(action_feasible ?a))
 )
 :effect (and
  (at end(pipeline_found ?p))
 )
)
...
```

**Listing 6. Snippet of the PDDL domain formulation for SUAVE containing the search pipeline action definition.**

## 7.3 Performance

### 7.3.1 Experimental setup

To evaluate ROSA's performance, the SUAVE exemplar was configured with the same parameters as described in the SUAVE paper (Silva et al., 2023). In addition, for the extended use case, the battery was set to discharge within 200 s, and the *search pipeline* and *inspect pipeline* actions were set to require at least 25% of battery to be performed.

### 7.3.2 Results

The results obtained are shown in Table 6. The different managed subsystems had similar performance despite the difference in their

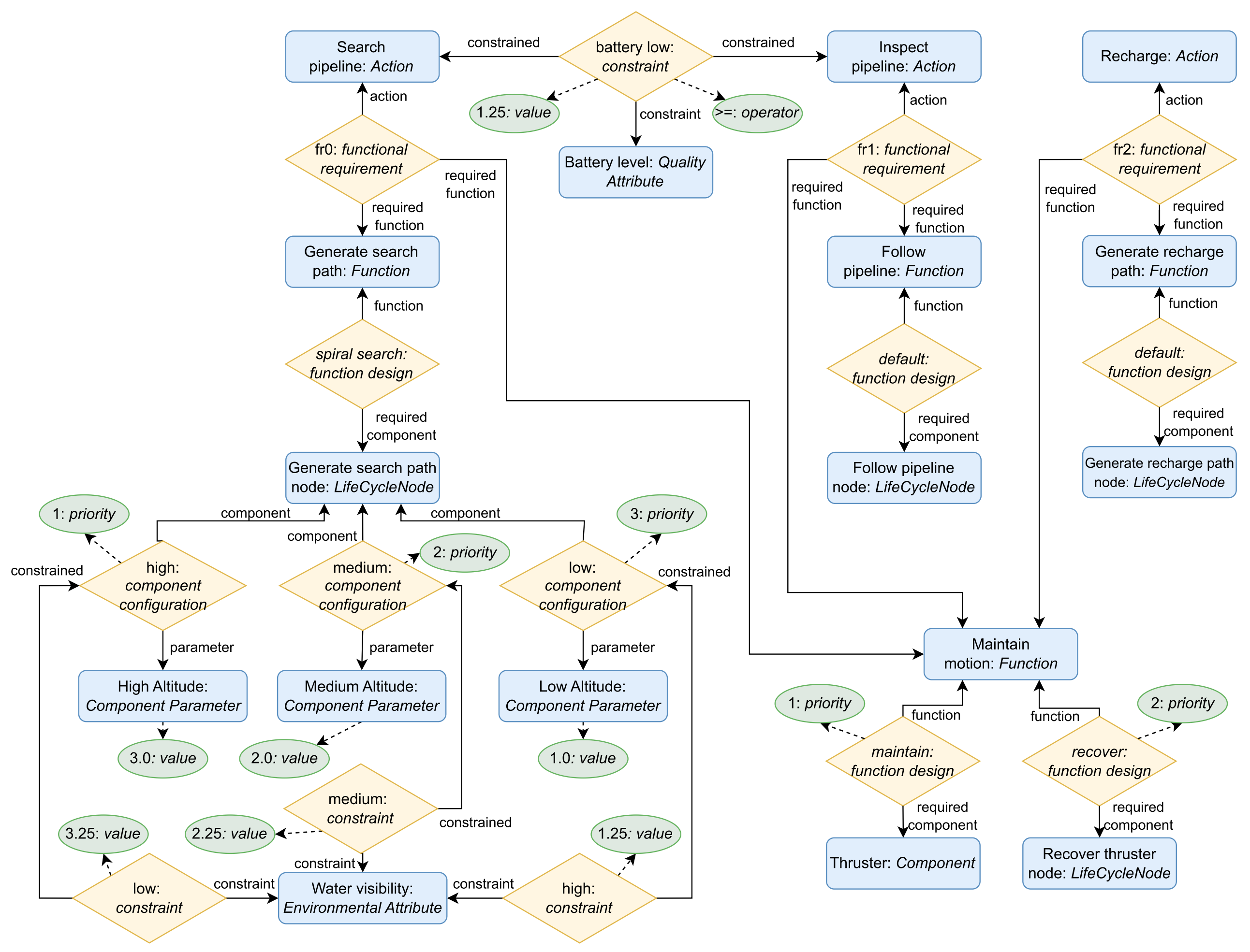


FIGURE 6
ROSA model for SUAVE.

reaction time. Furthermore, since the performance with ROSA was better than without a managing subsystem and close to the other managing subsystems, it can be considered as additional evidence of the feasibility of applying it at runtime to enable self-adaptation in robotic systems.

## 7.4 Reusability

### 7.4.1 Autonomous ground vehicle use case

In this scenario, an AGV has to navigate from an initial to a goal position in a graph-like environment while facing uncertainties such as component failures, corridors with obstacles, and changing light conditions (Cámara et al., 2020).

The AGV has distinct architectural variants available to solve navigation. It has three localization algorithms (AMCL, MRPT, or aruco) and three sensing components (camera, lidar, or Kinect). However, there are some restrictions on how they can be combined. The AMCL and MRPT algorithms can only be combined with lidar or Kinect, and the aruco algorithm can only be combined with a camera. In low-light conditions, the camera can only be used with a lamp. Furthermore, the robot can move at three different speeds. Each configuration has a different energy cost, safety, and accuracy level. This scenario can be solved with ROSA with the knowledge model depicted in Figure 8[11].

When navigating, the robot performs adaptation by selecting which corridors it needs to go through given its feasible configurations, and by selecting a suitable architecture configuration for each corridor it goes through. For example, to go from point $A$ to $B$, the AGV can go directly through a corridor with obstacles $C1$ or through corridors $C4 \rightarrow C3 \rightarrow C2$ without obstacles. Ideally, the AGV should go through $C1$ as it is the shortest path. Considering that the Kinect and AMCL combination is the only one with enough accuracy to go through a corridor with obstacles, in the case that the Kinect fails, the robot needs to perform TACA by adapting its task plan to go through $C4 \rightarrow C3 \rightarrow C2$ while simultaneously adapting its architecture, e.g., to use the lidar as its sensing component.

11 The accuracy estimation for *fd1* and *fd2* and all energy estimations are omitted from the figure to improve readability.

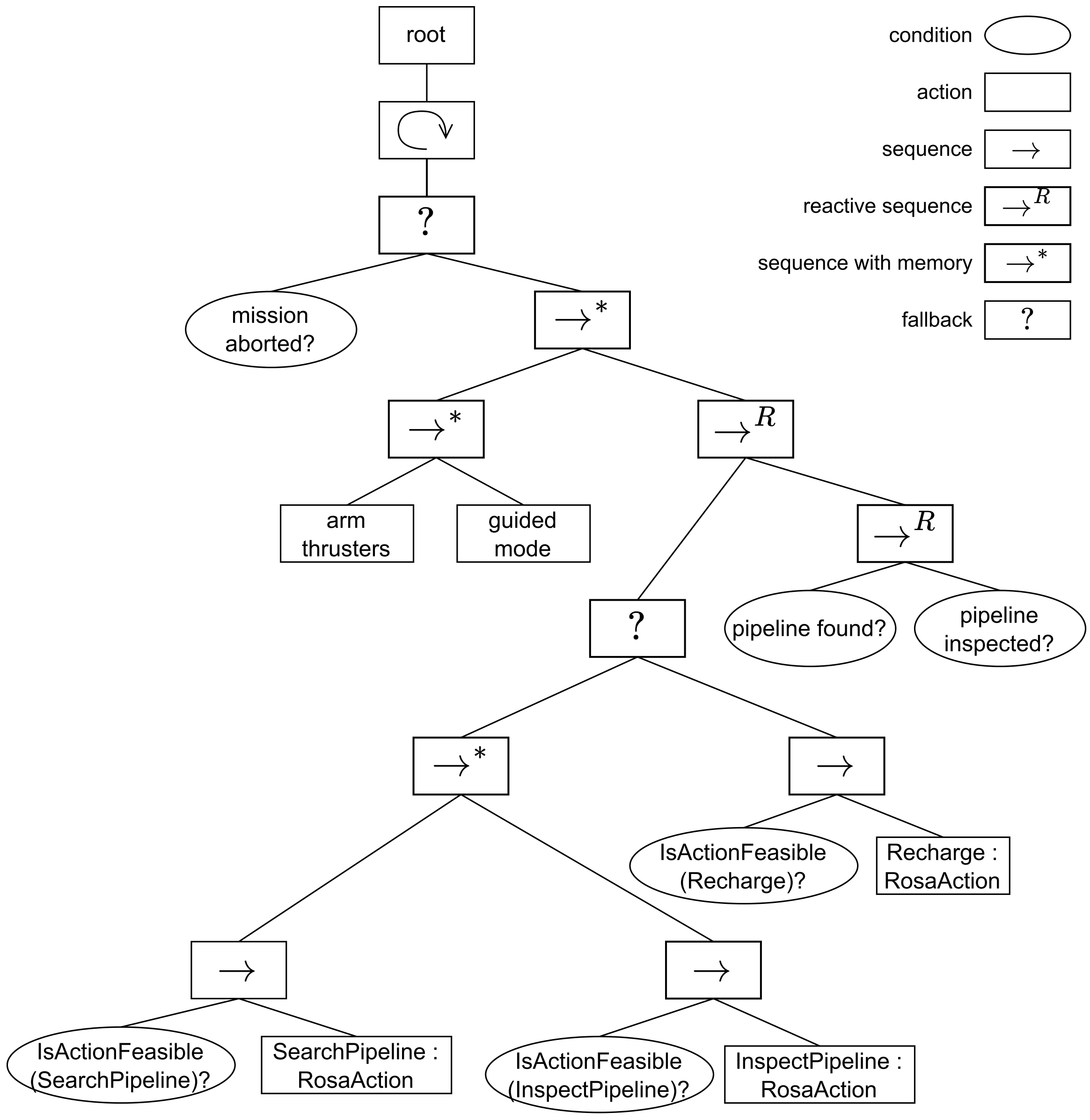


FIGURE 7
Behavior tree for the extended SUAVE use case.

### 7.4.2 Unmanned aerial vehicle use case

In this scenario, a UAV has to search for samples in a predefined area and analyze them (Braberman et al., 2017). To accomplish this mission, the UAV can perform the actions *(A1) search for samples*, *(A2) pick up and analyze samples*, *(A3) analyze samples on site*, *(A4) return to base and recharge*, and *(A5) land and fold gripper*. The analyze action *A2* performs a better analysis than *A3*, however, it consumes more battery. Furthermore, action *A2* requires a *gripper*, while *A3* requires an *infra-red camera*. When operating, the UAV might run out of battery, and its gripper might fail.

There are three adaptation scenarios in this use case: *(1)* if the battery level is insufficient to perform *A1*, the UAV must perform *A4*; *(2)* if the battery level is insufficient to perform *A2* but it is still sufficient to perform *A3*, the UAV must perform *A3*; *(3)* if the *gripper* fails while performing *A2*, the UAV must perform *A3*. Before transitioning from *A2* to *A3*, the UAV must first perform *A5*.

Adaptation scenarios *1* and *2* can be solved with ROSA's knowledge model by capturing the battery level as a `Quality Attribute` and using it as a `constraint` for actions *A1* and *A2*. Adaptation scenario *3* can be solved by capturing that the `Function` required by *A2* requires a *gripper* component. Furthermore, the task decision layer is responsible for guaranteeing that *A3* is only performed when *A5* is finished.

### 7.4.3 Results

This evaluation demonstrates that in addition to SUAVE, the knowledge required for the adaptation logic to solve the AGV and

TABLE 6 Mission results.

| Managing system | Number of runs | Search time (s) | | Distance inspected (s) | | Mean reaction time (s) | | |
|---|---|---|---|---|---|---|---|---|
| | | Mean | Std dev | Mean | Std dev | U1 | U2 | U3 |
| **SUAVE** | | | | | | | | |
| None | 100 | 174.75 | 36.00 | 33.20 | 13.49 | N/A | N/A | N/A |
| BT | 100 | 84.09 | 26.41 | 62.70 | 7.78 | 0.08 | 0.10 | N/A |
| Metacontrol | 100 | 89.24 | 35.57 | 60.57 | 11.17 | 1.55 | 0.82 | N/A |
| **ROSA** | **100** | **85.11** | **32.48** | **60.76** | **10.29** | **1.24** | **1.57** | **N/A** |
| **SUAVE extended** | | | | | | | | |
| BT | 100 | 94.37 | 34.92 | 20.88 | 3.81 | 0.07 | 0.10 | 1.09 |
| **ROSA** | **100** | **92.75** | **35.92** | **18.97** | **3.38** | **1.39** | **1.67** | **2.50** |

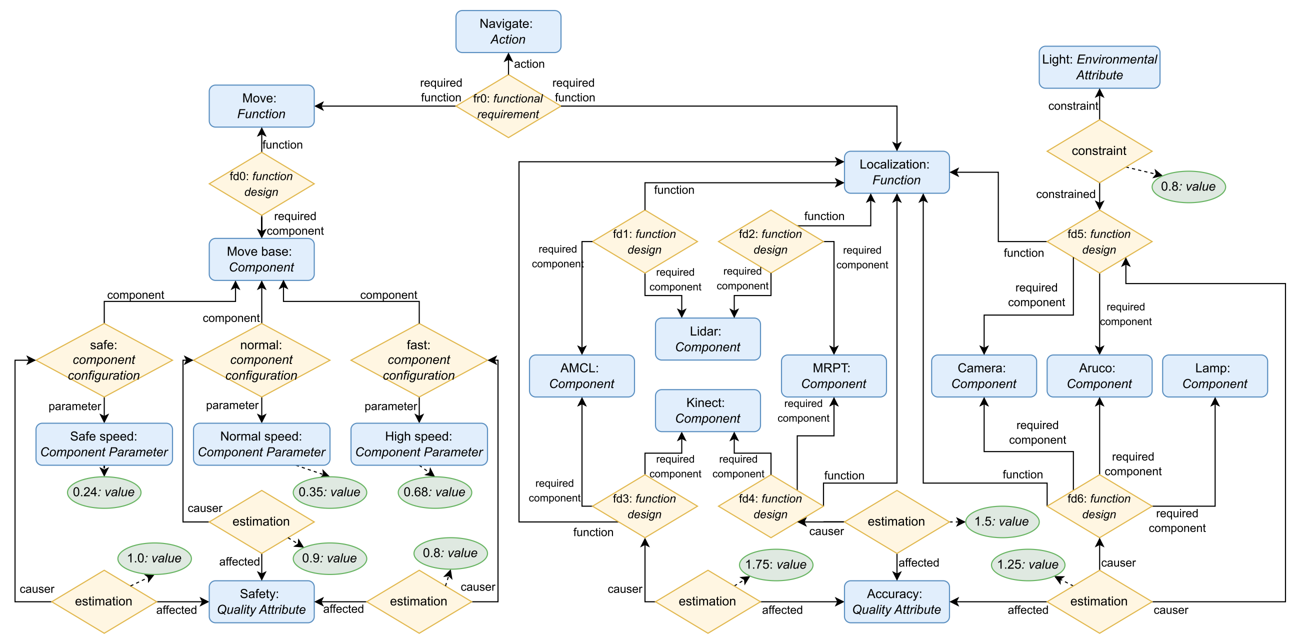


FIGURE 8
ROSA model for the AGV use case (Cámara et al., 2020).

UAV use cases can be captured with ROSA's knowledge model. This indicates that ROSA's knowledge model can be used to capture the knowledge required for TACA in adaptation scenarios similar to the ones presented. Furthermore, it shows that all entities and relationships contained in the proposed knowledge model had to be used to model the aforementioned adaptation scenarios, supporting their inclusion in the knowledge model.

## 7.5 Development effort

### 7.5.1 Experimental setup

To measure ROSA's development effort for each use case, we count the number of entities and relationships contained in the models created for the use cases presented. To measure the BT managing system development effort for SUAVE, we count the

TABLE 7 Development effort of using ROSA and BTs as managing subsystems.

| ROSA | | | |
|---|---|---|---|
| Use case | Entities | Relations | Total |
| SUAVE | 18 | 12 | 30 |
| SUAVE extended | 22 | 16 | 38 |
| AGV | 18 | 36 | 54 |
| UAV | 24 | 16 | 40 |
| **Behavior tree** | | | |
| Use case | BT | System modes | Total |
| SUAVE | 27 | 30 | 57 |
| SUAVE extended | 34 | 38 | 72 |

number of nodes contained in the BTs created to solve it, and the number of modes and parameters included in the System Modes' configuration file already packaged in SUAVE. In the remainder of this paper, we denote the elements modeled in both approaches as *overhead*.

### 7.5.2 Results

The overhead for both approaches can be seen in Table 7. Although it is not possible to make a straightforward comparison between the development efforts of both approaches using the observed overheads since the difficulty of modeling an element using the different modeling techniques is subjective, analyzing the reason for the observed overheads provides insights for comparing the development efforts of both approaches.

#### 7.5.2.1 ROSA

In TypeDB, each entity and relationship is inserted in the KB with TypeQL queries such as the ones presented in Listing 5. Thus, the total number of queries that the roboticist must define to solve an adaptation scenario is equal to the sum of the number of entities and relationships contained in the model, with a clear separation between the task and adaptation logic.

#### 7.5.2.2 BT managing system

The BT used to model SUAVE's task logic without adaptation contains 10 nodes, and the BT used for the extended use case contains 16 nodes. These values were deducted from the development effort metric for the BT managing system since they are independent of the adaptation problem.

Figure 9 depicts the pattern used to model SUAVE's *search pipeline* action and its related adaptations[12]. As can be seen, there is no separation between the task and adaptation logic, which is the main limitation of using BTs in comparison to using ROSA to model the adaptation logic. The coupling of both logics hinders the reusability of the approach as another system with the same task logic but different adaptation logic, or vice-versa, cannot reuse the existing BTs. In addition, when any changes are made to the task or adaptation problems, it will most likely require changes to parts of the BT that are not necessarily related to the changes introduced. Furthermore, it makes the modeling process more difficult as the roboticist needs to consider both problems simultaneously when modeling the BTs.

12 The full BT can be found in the SUAVE repository.

## 7.6 Development effort scalability

### 7.6.1 Experimental setup

To evaluate how ROSA's development effort scales for more complex use cases, we analyze how the development effort of a base scenario grows with the addition of new actions and adaptations. The growth for adding actions and adaptations depends on the specific application. Thus, we make the following assumptions to generalize and simplify the analysis of adding adaptation.

**Assumption 1:** *Every action requires one function that has only one configuration available consisting of a single component with no parameters.*

**Assumption 2:** *Every* `Component` *is a* `ROSNode` *containing one* `package` *and one* `executable` *attribute; every* `ROSNode` *has one* `component configuration` *with one* `Component Parameter`*; every* `function design` *and* `component configuration` *must be related to a* `constraint` *and contain a* `priority` *attribute; and a single* `Quality Attribute` *is defined for the whole system.*

### 7.6.2 Results

Consider a base scenario where the robot has only one action and complies with Assumption 1 and Assumption 2. The ROSA model to solve it contains 10 elements and is depicted in Figure 10a.

To add structural adaptation to the base scenario (given Assumption 2), it is necessary to add the elements depicted in Figure 10b to the model. This results in a minimum overhead of 6 elements for each structural adaptation. To add parameter adaptation to the base scenario (given Assumption 2), it is necessary to add the elements depicted in Figure 10c to the model. This results in a minimum overhead of 3 elements for each parameter adaptation.

In conclusion, given Assumption 1 and Assumption 2, the total overhead per action can be defined as $10 + 6 * n_{sa} + 3 * n_{pa}$, where $n_{sa}$ and $n_{pa}$ represent the number of structural and parameter adaptations for the action, respectively. This indicates that the ROSA model grows linearly with the number of actions and adaptations, which is made possible by the clear separation of the task and adaptation logic.

## 8 Conclusions and future work

This work proposed ROSA, a knowledge-based solution for task-and-architecture co-adaptation in robotic systems that

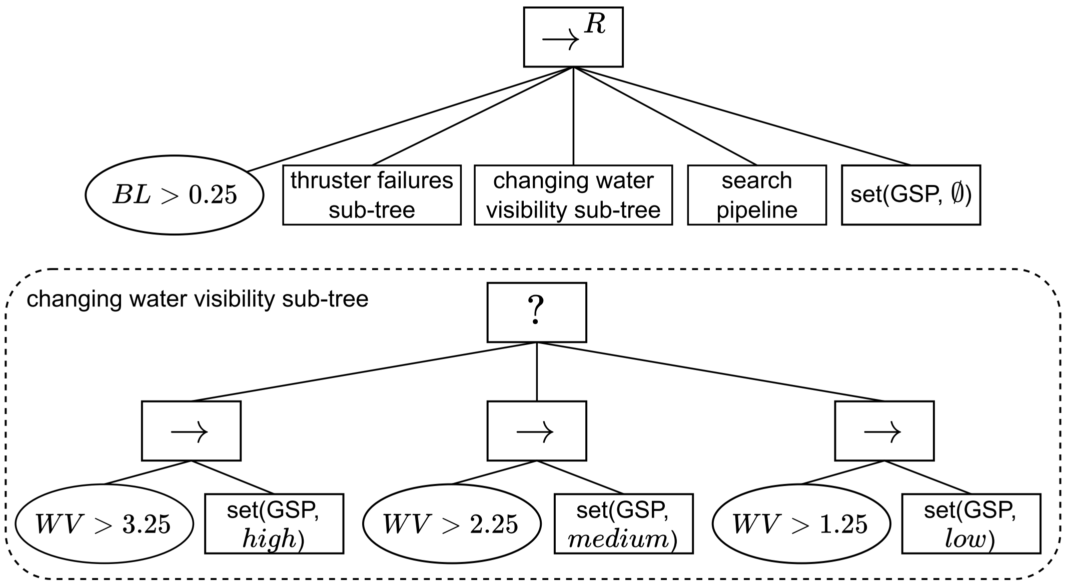


FIGURE 9
Snippet of the BT used to capture SUAVE's task and adaption logic. *WV* and *BL* represent the measured water visibility and battery level, respectively. *GSP*, *MM*, and *FP* represent SUAVE's *generate search path node*, *maintain motion node*, and *follow pipeline node*, respectively. The *set* action changes the system's modes.

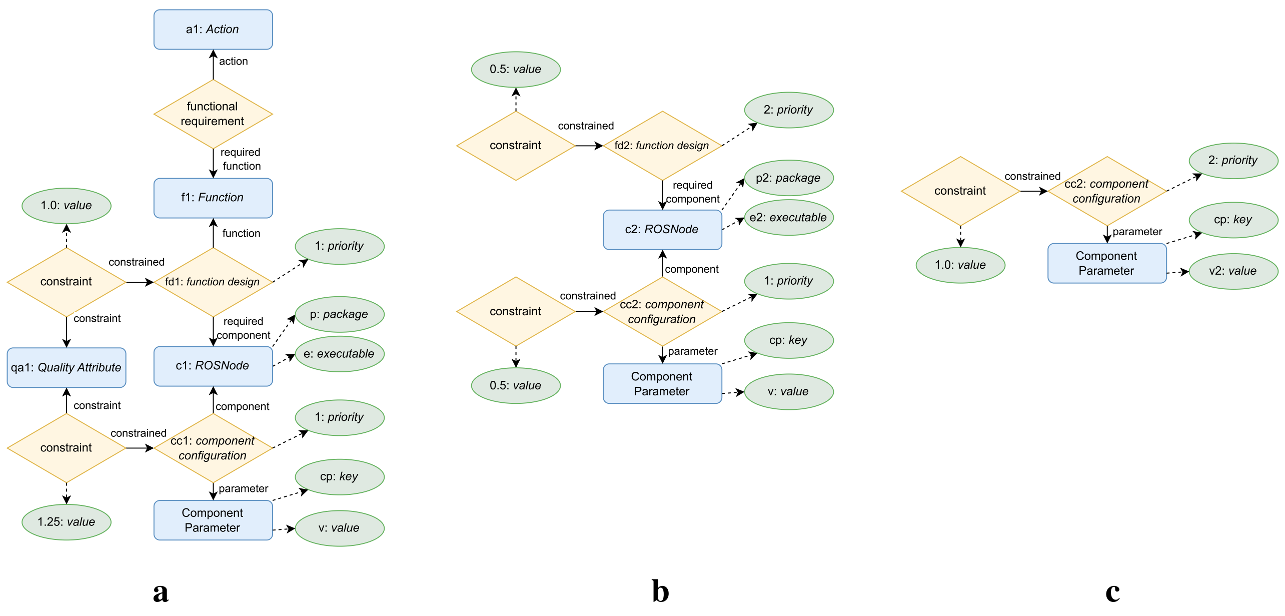


FIGURE 10
Minimum knowledge required to model adaptation given Assumption 1 and Assumption 2. The name attribute is depicted as *name: Entity* or *name: Relationship* inside each diamond and rectangle shape, respectively. **(a)** Base model. **(b)** Structural adaptation. **(c)** Parameter adaptation.

promotes reusability, extensibility, and composability. Reusability was achieved by proposing a knowledge model that can capture the knowledge required for TACA and using it at runtime to reason about adaptation. Extensibility and composability were achieved with an architectural design that allows ROSA's components to be stateless and self-contained. The feasibility of using ROSA in robotic systems at runtime was demonstrated by applying it in simulation to the SUAVE exemplar. ROSA's reusability was demonstrated by using it to model different self-adaptive robotic systems and showing that it can capture all relevant knowledge for adaptation necessary for these use cases. Furthermore, ROSA's development effort and its scalability were demonstrated for the use cases presented in this paper and for a hypothetical scenario.

ROSA modular architecture has been designed to provide reuse and extensibility of the framework by future works applying self-adaptation principles in robotics architectures. For example, the current ROSA implementation supports integration with robotics architectures using planning or behavior tree solutions for the task deliberation layer, but it would be interesting to extend it to support other decision-making methods, such as state machines or Markov decision processes.

As a future work, we intend to integrate learning in ROSA. For example, machine learning methods could be explored to update at runtime the `Quality Attribute` and `Environmental Attribute` estimations and constraints `values`. Another possibility is learning that constraints and estimations exist without prior knowledge, i.e., learning that the relationship itself should be modeled. ROSA's interfaces to manipulate the KB at runtime could be exploited for this end.

## Data availability statement

The original contributions presented in the study are included in the article/supplementary material, further inquiries can be directed to the corresponding author.

## Author contributions

GR: Conceptualization, Investigation, Methodology, Software, Writing – original draft, Writing – review and editing. JP: Conceptualization, Writing – review and editing. ST: Conceptualization, Writing – review and editing. EJ: Conceptualization, Writing – review and editing. CH: Conceptualization, Writing – review and editing.

## Funding

The author(s) declare that financial support was received for the research and/or publication of this article. This work was supported by the European Union's Horizon 2020 Framework Programme through the MSCA network REMARO (Grant Agreement No 956200).

## Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## Generative AI statement

The author(s) declare that no Generative AI was used in the creation of this manuscript.

## Publisher's note

## References

Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (1998). An architecture for autonomy. *Int. J. Robotics Res.* 17, 315–337. doi:10.1177/.027836499801700402

Alberts, E., Gerostathopoulos, I., Malavolta, I., Hernández Corbato, C., and Lago, P. (2025). Software architecture-based self-adaptation in robotics. *J. Syst. Softw.* 219, 112258. doi:10.1016/j.jss.2024.112258

Antoniou, G., and van Harmelen, F. (2004). "Web ontology language: OWL," in *Handbook on ontologies. International handbooks on information systems*. Editors S. Staab, and R. Studer (Springer), 67–92. doi:10.1007/.978-3-540-24750-0_4

Barnett, W., Cavalcanti, A., and Miyazawa, A. (2022). Architectural modelling for robotics: RoboArch and the CorteX example. *Front. Robot. AI* 9, 991637. doi:10.3389/frobt.2022.991637

Beetz, M., Mösenlechner, L., and Tenorth, M. (2010). "CRAM — a Cognitive Robot Abstract Machine for everyday manipulation in human environments," in IEEE/RSJ International Conference on Intelligent Robots and Systems, 1012–1017. doi:10.1109/IROS.2010.5650146

Board, S. E. (2023). "The guide to the systems engineering body of knowledge (SEBoK), v. 2.8," in BKCASE is managed and maintained by the Stevens Institute of Technology Systems Engineering Research Center, the International Council on Systems Engineering, and the Institute of Electrical and Electronics Engineers Systems Council. Editor R. J. Cloutier (Hoboken, NJ: The Trustees of the Stevens Institute of Technology).

Bozhinoski, D., Oviedo, M. G., Garcia, N. H., Deshpande, H., van der Hoorn, G., Tjerngren, J., et al. (2022). MROS: runtime adaptation for robot control architectures. *Adv. Robot.* 36, 502–518. doi:10.1080/01691864.2022.2039761

Bozhinoski, D., and Wijkhuizen, J. (2021). "Context-based navigation for ground mobile robot in semi-structured indoor environment," in *2021 fifth IEEE international conference on robotic computing (IRC)*, 82–86. doi:10.1109/IRC52146.2021.00019

Braberman, V., D'Ippolito, N., Kramer, J., Sykes, D., and Uchitel, S. (2017). "An Extended Description of MORPH: A Reference Architecture for Configuration and Behaviour Self-Adaptation," in *Software engineering for self-adaptive systems III. Assurances*. Editors R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese (Cham: Springer International Publishing, Lecture Notes in Computer Science), 377–408. doi:10.1007/978-3-319-74183-3_13

Cámara, J., Schmerl, B., and Garlan, D. (2020). "Software architecture and task plan co-adaptation for mobile service robots," in Proceedings 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '20) (New York, NY: Association for computing Machinery), 20, 125–136. doi:10.1145/.3387939.3391591

Carreno, Y., Scharff Willners, J., Petillot, Y. R., and Petrick, R. (2021). "Situation-aware task planning for robust AUV exploration in extreme environments," in Proceedings IJCAI Workshop on Robust and Reliable Autonomy in the Wild.

Chen, P. P. (1976). The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.* 1, 9–36. doi:10.1145/320434.320440

Colledanchise, M., and Ögren, P. (2018). *Behavior trees in robotics and AI: an introduction*. (Boca Raton: CRC Press). doi:10.1201/9780429489105

Dorn, C., and Pribadi, H. (2023). "Type theory as a unifying paradigm for modern databases," in Proceedings 32nd International Conference on Information and Knowledge Management (CIKM 2023). Editors I. Frommholz, F. Hopfgartner, M. Lee, M. Oakes, M. Lalmas, M. Zhang, et al. (New York, NY: Association for Computing Machinery), 5238–5239. doi:10.1145/3583780.3615999

Dorn, C., and Pribadi, H. (2024). TypeQL: A Type-Theoretic & Polymorphic Query Language. *Proc. ACM Manag.* 27. doi:10.1145/3651611

Garcia, S., Menghi, C., Pelliccione, P., Berger, T., and Wohlrab, R. (2018). "An architecture for decentralized, collaborative, and autonomous robots," in

2018 IEEE International Conference on Software Architecture (ICSA), 75–7509. doi:10.1109/ICSA.2018.00017

Ghallab, M., Howe, A., Knoblock, C., Mcdermott, D., Ram, A., Veloso, M., et al. (1998). PDDL—the planning domain definition language. [Dataset].

Gherardi, L., and Hochgeschwender, N. (2015). RRA: Models and tools for robotics run-time adaptation. IEEE/RSJ International Conference on Intelligent Robots and Systems IROS, 1777–1784. doi:10.1109/IROS.2015.7353608

Hamilton, J., Stefanakos, I., Calinescu, R., and Cámara, J. (2022). "Towards adaptive planning of assistive-care robot tasks," Proceedings 4th International Workshop on Formal Methods for Autonomous Systems (FMAS 2022), 371 of EPTCS, 175–183. M. Luckcuck and M. Farrell. doi:10.4204/.EPTCS.371.12

Hernández, C., Bermejo-Alonso, J., and Sanz, R. (2018). A self-adaptation framework based on functional knowledge for augmented autonomy in robots. *Integr. Computer-Aided Eng.* 25, 157–172. doi:10.3233/ica-180565

Hochgeschwender, N., Schneider, S., Voos, H., Bruyninckx, H., and Kraetzschmar, G. K. (2016). "Graph-based software knowledge: storage and semantic querying of domain models for run-time adaptation," in Proc. Intl. Conf. On simulation, modeling, and programming for autonomous robots (SIMPAR 2016) (IEEE), 83–90. doi:10.1109/SIMPAR.2016.7862379

IEEE Approved Draft Standard for Robot Task Representation (2024). *P1872.1/D5*. IEEE, 1–35.

Kazhoyan, G., Stelter, S., Kenfack, F. K., Koralewski, S., and Beetz, M. (2021). "The robot household marathon experiment," in 2021 IEEE International Conference on Robotics and Automation (ICRA), 9382–9388. doi:10.1109/.ICRA48506.2021.9560774

Kephart, J. O., and Chess, D. M. (2003). The vision of autonomic computing. *Computer* 36, 41–50. doi:10.1109/mc.2003.1160055

Kortenkamp, D., Simmons, R., and Brugali, D. (2016). *Robotic systems architectures and programming*. Cham: Springer International Publishing, 283–306. doi:10.1007/.978-3-319-32552-1_12

Kotseruba, I., and Tsotsos, K. J. (2018). 40 years of cognitive architectures: core cognitive abilities and practical applications. *Artif. Intell. Rev.* 53, 17–94. doi:10.1007/s10462-018-9646-y

Lotz, A., Inglés-Romero, J. F., Vicente-Chicote, C., and Schlegel, C. (2013). "Managing run-time variability in robotics software by modeling functional and non-functional behavior," in *Enterprise, business-process and information systems modeling*. Editors S. Nurcan, H. A. Proper, P. Soffer, J. Krogstie, R. Schmidt, T. Halpin, et al. (Berlin, Heidelberg: Springer Berlin Heidelberg), 441–455.

Macenski, S., Foote, T., Gerkey, B., Lalancette, C., and Woodall, W. (2022). Robot operating system 2: design, architecture, and uses in the wild. *Sci. Robotics* 7, eabm6074. doi:10.1126/scirobotics.abm6074

Martín, F., Clavero, J. G., Matellán, V., and Rodríguez, F. J. (2021). PlanSys2: A planning system framework for ROS2. IEEE/RSJ International Conference on Intelligent Robots and Systems IROS, 9742–9749. doi:10.1109/IROS51168.2021.9636544

Niemczyk, S., and Geihs, K. (2015). "Adaptive run-time models for groups of autonomous robots," in 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 127–133. doi:10.1109/.SEAMS.2015.21

Niemczyk, S., Opfer, S., Fredivianus, N., and Geihs, K. (2017). "Ice: self-configuration of information processing in heterogeneous agent teams," in Proceedings of the Symposium on Applied Computing (New York, NY, USA: Association for Computing Machinery, SAC '17), 417–423. doi:10.1145/.3019612.3019653

Nordmann, A., Lange, R., and Rico, F. M. (2021). "System modes - digestible system (re-)configuration for robotics," in 2021 IEEE/ACM 3rd international Workshop on Robotics Software Engineering (RoSE), 19–24. doi:10.1109/.RoSE52553.2021.00010

Olivé, A. (2007). *Conceptual modeling of information systems*. Springer. doi:10.1007/978-3-540-39390-0

Park, Y.-S., Koo, H.-M., and Ko, I.-Y. (2012). A task-based and resource-aware approach to dynamically generate optimal software architecture for intelligent service robots. *Softw. Pract. Exp.* 42, 519–541. doi:10.1002/spe.1074

Sanchez-Lopez, J. L., Suárez Fernández, R. A., Bavle, H., Sampedro, C., Molina, M., Pestana, J., et al. (2016). "Aerostack: an architecture and open-source software framework for aerial robotics," in 2016 International Conference on Unmanned Aircraft Systems (ICUAS), 332–341. doi:10.1109/ICUAS.2016.7502591

Silva, G. R., Päßler, J., Zwanepol, J., Alberts, E., Tapia Tarifa, S. L., Gerostathopoulos, I., et al. (2023). "SUAVE: an exemplar for self-adaptive underwater vehicles," in Proc. 18th IEEE/ACM symposium on software engineering for adaptive and self-managing systems SEAMS 2023 (IEEE), 181–187.

Thalheim, B. (1993). Foundations of entity - relationship modeling. *Ann. Math. Artif. Intell.* 7, 197–256. doi:10.1007/BF01556354

Thalheim, B. (2000). *Entity-relationship modeling - foundations of database technology*. Springer.

Valner, R., Vunder, V., Aabloo, A., Pryor, M., and Kruusamäe, K. (2022). TeMoto: a software framework for adaptive and dependable robotic autonomy with dynamic resource management. *IEEE Access* 10, 51889–51907. doi:10.1109/access.2022.3173647

Weyns, D. (2020). *An introduction to self-adaptive systems: a contemporary software engineering perspective*. John Wiley and Sons.

Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., et al. (2013). "On patterns for decentralized control in self-adaptive systems," in *Software engineering for self-adaptive systems II: international seminar, dagstuhl castle, Germany, october 24-29, 2010 revised selected and invited papers*. Editors R. de Lemos, H. Giese, H. A. Müller, and M. Shaw (Berlin, Heidelberg: Springer), 76–107. doi:10.1007/978-3-642-35813-5_4

TYPE Original Research
PUBLISHED 16 July 2025
DOI 10.3389/frobt.2025.1592523

OPEN ACCESS

EDITED BY
Ivano Malavolta,
VU Amsterdam, Netherlands

REVIEWED BY
Calvin Cheung,
University of Michigan–Dearborn,
United States
Marco Stadler,
Johannes Kepler University of Linz, Austria

*CORRESPONDENCE
Lukas Dust,
✉ lukas.dust@mdu.se

RECEIVED 12 March 2025
ACCEPTED 16 June 2025
PUBLISHED 16 July 2025

CITATION
Dust L, Gu R, Mubeen S, Ekström M and Seceleanu C (2025) A model-based approach to automation of formal verification of ROS 2-based systems.
*Front. Robot. AI* 12:1592523.
doi: 10.3389/frobt.2025.1592523

# A model-based approach to automation of formal verification of ROS 2-based systems

Lukas Dust*, Rong Gu, Saad Mubeen, Mikael Ekström and Cristina Seceleanu

School of Innovation, Design, and Technology, Mälardalen University, Västerås, Sweden

Formal verification of robotic applications, particularly those based on ROS 2, is desirable for ensuring correctness and safety. However, the complexity of formal methods and the manual effort required for model creation and parameter extraction often hinder their adoption. This paper addresses these challenges by proposing a model-based methodology that automates the formal verification process using model-driven engineering techniques. We introduce a methodology which can be applied as a toolchain that automates the initialization of formal model templates in UPPAAL using system parameters derived from ROS 2 execution traces generated by the ROS2_tracing tool. The toolchain employs four model representations based on custom Eclipse Ecore metamodels to capture both structural and verification aspects of ROS 2 systems. The methodology supports both implemented and conceptual systems and enables iterative verification of timing and scheduling parameters through model-to-model and model-to-text transformations. A proof-of-concept implementation demonstrates the feasibility of the proposed approach. The designed toolchain supports verification using two types of UPPAAL models: one for individual node verification (e.g., callback latency and buffer overflow) and another for end-to-end latency analysis of ROS 2 processing chains. Experiments conducted on two implemented and one conceptual ROS 2 systems validate the correctness and adaptability of the toolchain. The results show that the toolchain can automate parameter extraction and model generation. The proposed methodology modularizes the verification process, allowing domain experts to focus on their areas of expertise. It targets to enhances traceability and reusability across different verification scenarios and formal models. The approach aims to make formal verification more accessible and practical to robotics developers.

KEYWORDS

**ROS 2, robotic systems, formal verification, model checking, model-based engineering**

## 1 Introduction

Ensuring that a robotic system's design and implementation meet the requirements specification is crucial for guaranteeing the system's desired behavior. Various verification methods are employed to achieve this, with formal methods, e.g., model checking, being particularly effective due to their rigorous mathematical approach to analyzing complex system models during the design phase (Carvalho et al., 2020). Model checking involves an exhaustive exploration of the system's model state space to verify that the system meets its specification, uncovering potential errors that might be missed by traditional trial-and-error methods such as simulation and experimentation (Dust et al., 2023a).

Despite its advantages, the application of formal model-based approaches in distributed and complex systems poses significant challenges. The steep learning curve associated with the mathematical syntax and semantics of formal modeling languages can be a barrier for robotic developers. Consequently, the high initial effort required for model checking often deters its use in industry, leading developers to rely on less rigorous, trial-and-error methods (Rajkumar et al., 2010).

The Robot Operating System (ROS) (OpenRobotics, 2023b; OpenRobotics, 2023a) is an open-source middleware that facilitates rapid prototyping and deployment of robotic systems. ROS-based systems, particularly those with safety-critical applications, have stringent timing requirements that necessitate real-time capabilities in the middleware. These capabilities are influenced by various system components, including communication, task scheduling, and execution. To address the limitations of ROS in real-time applications, ROS 2 was developed, incorporating real-time communication through the Data Distribution Service (DDS) C. S. V. (Gutiéerrez et al., 2018). While DDS provides a robust framework for real-time communication, the task scheduling in ROS 2 still requires extensive analysis to ensure deterministic timing behavior (Casini et al., 2019; Blaß et al., 2021).

Tools like *ROS2_tracing* (Bédard et al., 2022) and *Autoware_perf* (Li et al., 2022) have been developed to trace system execution and analyze performance based on execution traces. However, these tools primarily offer experimental analysis, which may not be exhaustive and could miss potential system errors. In contrast, model checking offers a comprehensive verification approach capable of identifying all potential bugs in the model. Despite this, the manual application of formal methods to ROS 2 systems remains error-prone and time-consuming, requiring significant background knowledge.

In our previous work (Dust et al., 2023a), we utilize the UPPAAL model checker (Alur and Dill, 1994) to create reusable templates for verifying timing behavior and buffer overflow in ROS 2 systems. These templates simplify the modeling process by allowing systems to be instantiated from pre-defined templates rather than constructed from scratch. However, this approach still requires detailed knowledge of static and runtime system parameters, as well as of the modeling language itself, to represent verification properties accurately. The manual nature of this process makes it susceptible to errors, as parameters are often determined through source-code analysis and runtime evaluation.

## 1.1 Problem definition and paper contributions

In our previous work (Dust et al., 2023b; Dust et al., 2024), we identified scheduling-related timing issues in ROS 2 and developed formal model templates to address such issues (Dust et al., 2023a). The proposed template-based verification facilitates formal verification, but initializing the formal model templates requires extended knowledge and analysis of static and runtime parameters. Furthermore, due to the manual process of initializing the formal models, extended knowledge about the modeling language is needed. Additionally, manual source code and runtime analysis make the proposed verification vulnerable to errors. In this paper, we aim to simplify the formal verification process by proposing a model-based methodology that automates parameter determination and model initialization using the existing formal model templates. This methodology is designed for robotics developers, with the goal of making formal verification more accessible and less error-prone. In this article, we extend our work (Dust et al., 2024), automating model-based formal verification using model-driven engineering techniques.

Based on the problem definition, we develop the three research questions. The research questions are presented in the following paragraphs.

As the first goal of this paper, we aim to identify an approach that can be used to automate the application of formal verification. By proposing an approach, we aim to reduce the complexity for practitioners when applying formal methods through the facilitation of automation. To achieve the stated goal, we design a methodology and implement a proof of concept of a toolchain that enables the application of such methodology. Hence, we contribute a bridging approach utilizing ROS 2 traces, modeling, and formal methods that automate formal verification through automated transformations.

As the second goal of this paper, we aim to modularize the process of formal verification. The modularization targets to decouple components of the verification process to enable actors to focus on their domain of expertise in the creation and adaptation of the verification process. In this paper, we modularize the process of formal verification through the design and modeling of a methodology that we apply in a novel toolchain implemented in this paper. Hence, we propose a layered and modular methodology that can be applied through the implemented toolchain.

As the third goal of the paper, we aim to propose a methodology that allows verification of different formal models, focusing on different properties to verify. The proposed methodology aims to separate the concerns in terms of the type of properties to verify. We demonstrate the ability of the methodology to allow verification of different formal models by implementation of a toolchain and experimental evaluation. As a result of the design, implementation, and evaluation, we develop a novel validated UPPAAL model for end-to-end (E2E) latency to enable comparison to formal models proposed in the literature. Furthermore, we develop a proof of concept that covers multiple verification goal-oriented UPPAAL models.

Summarizing, the following research questions are tackled in this paper:

**RQ1:** What approach can be employed to automate the application of formal verification of ROS 2-based applications?

**RQ2:** How can the formal verification process be modularized to enable domain experts to concentrate on their specific areas of expertise without requiring deep formal methods knowledge?

**RQ3:** How can a methodology incorporate verification using different formal models?

The contributions of this paper are summarized as follows:

1. A novel methodology for model-based verification of ROS 2 systems, featuring a toolchain that includes UPPAAL, Eclipse, and *ROS2_tracing*.
2. UPPAAL models for verification of end-to-end latencies in ROS 2 processing chains in a single executor.
3. Ecore metamodels to capture system structure and support verification activities.
4. Automated and conceptual model-to-model transformations from ROS 2 execution traces to Ecore models, and from Ecore models to UPPAAL models.
5. Demonstration of the proposed toolchain's workflow through a proof of concept implementation, covering key aspects of the toolchain architecture.

The remainder of this paper is organized as follows. Section 2 provides an overview of model checking, ROS 2, and model-driven engineering using Eclipse. Section 3 details the proposed methodology and toolchain architecture, along with the potential workflow. Section 4 presents the proof of concept implementation, followed by a discussion of related work in Section 6. The paper concludes with final remarks, and prospective future work in Section 7.

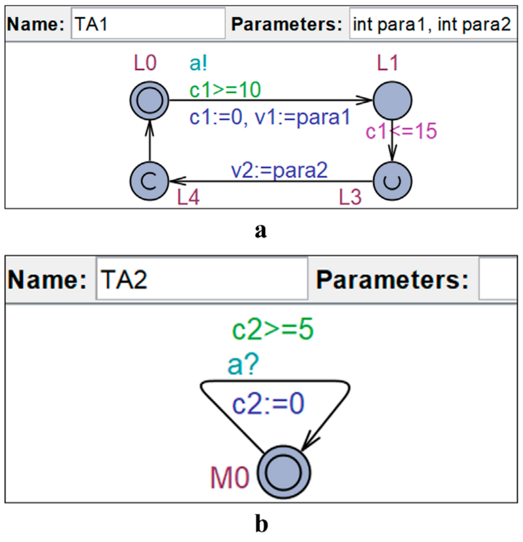


FIGURE 1
Examples of UTA in UPPAAL. **(a)** UTA template *TA1.* **(b)** UTA template *TA2.*

## 2 Background

In this section, we provide an overview of the essential concepts and tools relevant to our work, including model checking, ROS 2, model-driven engineering using Eclipse, and end-to-end timing analysis.

### 2.1 Model checking and UPPAAL

*Model checking* is a formal verification technique that offers a rigorous, mathematical approach to the analysis of complex systems during the design phase (Carvalho et al., 2020). It involves an exhaustive exploration of the system's model state space to ensure that the system meets its specification. UPPAAL is a widely used model checker for the modeling, simulation, and verification of real-time systems described as *timed automata* (Alur and Dill, 1994). It supports the creation of reusable templates to verify timing behavior and buffer overflow, making the application of formal verification more accessible.

Below, we provide a brief, informal overview of timed automata (TA). For detailed and precise definitions of TA and their application in UPPAAL, we refer the reader to the literature (Alur and Dill, 1994; Hendriks et al., 2006).

A *timed automaton* (TA) (Alur and Dill, 1994) consists of a finite set of *locations*, including an initial location, which are connected by *edges*, as well as a finite set of non-negative real-valued variables, known as *clocks*, which measure the elapse of time and progress simultaneously at rate 1. The edges are decorated with a finite set of *actions*, and *guards*, which are conjunctive Boolean formulas of clock constraints that need to evaluate to *true* for the edge to be traversed. Clocks can be reset over the edges, and a partial function assigns *invariants* to locations, which constrain the time allowed to elapse in a particular location. The semantics of TA is defined as a *labeled transition system* with delay and action transitions.

UPPAAL (Hendriks et al., 2006) is a tool used for modeling, simulation, and model checking of an extended version of timed automata called *UPPAAL Timed Automata* (UTA). In UPPAAL, UTA are organized as *templates* (see Figure 1) that can be instantiated. UTA enhances the capabilities of TA by adding features such as data variables, synchronization channels (Boolean variables decorated by "!" for sending, and by "?" for receiving), urgent and committed locations, and more. Furthermore, UPPAAL allows the composition of UTA in parallel as a network of UTA (NUTA), synchronized via *channels*.

Figures 1a,b show two NUTA implemented in UPPAAL. In these figures, blue circles represent *locations* connected by directional *edges*. Double-circled locations are the *initial* locations (e.g., L0). Locations marked with an encircled "u" are *urgent* (e.g., L3), and those with an encircled "c" are *committed* (e.g., L4). UTA imposes constraints that prevent time from progressing in urgent and committed locations. Committed locations have stricter rules: the next edge traversal must start from one of them.

Edges allow for assignments such as resetting clocks (e.g., c1:=0), updating data variables (e.g., v1:=para1), guards (e.g., c1>=10), and synchronization channels (e.g., a! and a?). At location L1, an invariant c1<=15 ensures that clock c1 does not exceed 15 time units in that location. In UPPAAL, UTA templates can include parameters (e.g., para1 in TA1) that are assigned values upon instantiation.

### 2.2 ROS 2

The Robot Operating System 2 (ROS 2) (OpenRobotics, 2023a) is an open-source middleware designed to facilitate the rapid development and prototyping of robotic systems. Unlike what its name suggests, ROS 2 is not a standalone

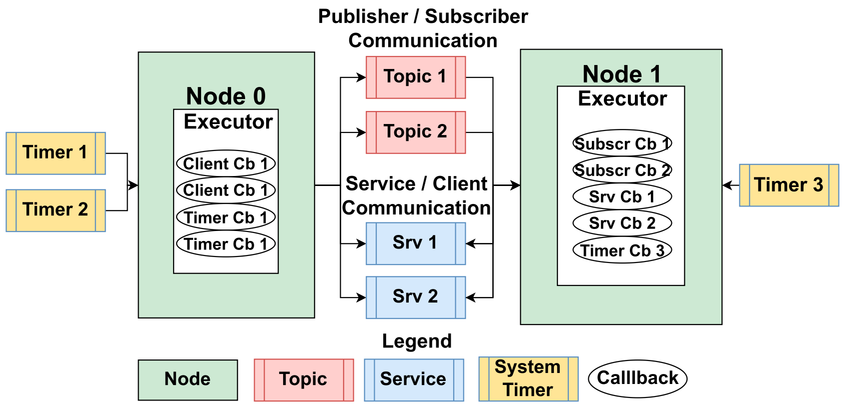


FIGURE 2
An example ROS 2 system showing the concepts of Node, Timer, Topic, Publisher, Subscriber, Service, and Client.

operating system but rather runs on top of an existing host OS, predominantly Linux.

ROS 2 was developed to meet industrial requirements such as fault tolerance and real-time performance. To achieve real-time capabilities, ROS 2 introduced the Data Distribution Service (DDS) Group (2022) as its communication protocol. DDS, created by the Object Management Group (OMG) Group (2022), enables efficient communication between distributed applications. While DDS is the default communication protocol, other protocols such as Zenoh can be utilized.

The fundamental building blocks of ROS 2 systems are *nodes*, which communicate through designated channels using DDS. ROS 2 supports two primary communication paradigms (Birman and Joseph, 1987): *Publisher-Subscriber* and *Service-Client*. In the *Publisher-Subscriber* model, nodes can either *publish* messages to a specific topic or *subscribe* to receive messages from that topic. All nodes subscribed to a topic receive the published messages. Conversely, the *Service-Client* model involves directed communication, where a client node requests a service from a server node, which then processes the request and sends back a response.

Figure 2 illustrates an example of a ROS 2 system with two nodes communicating over four channels. In addition to communication channels, system timers can be used to trigger functions within a node at specified intervals.

Nodes in ROS 2 are executable entities within the host OS and consist of several functions known as *callbacks*, which are the atomic schedulable units in ROS 2. Callbacks are triggered by events such as the arrival of data in an input buffer or a timer event. There are four types of callbacks: timer, subscriber, service, and client.

Each node also includes an executor, which is responsible for scheduling and executing callbacks (Casini et al., 2019; Blaß et al., 2021). The executor can operate with single or multiple threads, depending on the configuration chosen. However, the latest versions of ROS 2 do not provide options to set callback priorities, making the execution susceptible to blocking. This can lead to issues such as buffer overflow and missed callback instances in worst-case scenarios (Dust et al., 2023b; Dust et al., 2023a).

## 2.3 ROS2_tracing

ROS2_tracing (Bédard et al., 2022) is a low-overhead framework based on the Linux Trace Toolkit next-generation (LTTng). It is included in the ROS 2 installation and allows for the generation of execution traces. These traces provide valuable insights into the system's behavior, including callback execution times, message passing instances, and system initialization events. The ROS2_tracing toolbox includes a Python library called tracetools_analysis, which transforms LTTng traces into a defined ROS 2 data model, represented as pandas Python objects (Bédard et al., 2022).

## 2.4 Autoware real Time reference system

The Autoware Reference System (ROS Realtime Working Group, 2025b) is part of the ROS 2 (Robot Operating System) ecosystem, specifically designed to provide a standardized and repeatable benchmarking environment for evaluating the performance of various executors and configurations within the ROS 2 framework (ROS Realtime Working Group, 2025a). This real-time reference system simulates the Autoware. Auto (Autoware Foundation, 2025) LiDAR data pipeline, to measure and compare the performance of different executor implementations. The reference system is defined by a fixed number of nodes, each with specific publishers, subscribers, processing times, and publishing rates. It uses a fixed message type and size for consistency. The system can run on various platforms, including different hardware and operating systems, ensuring that the benchmarks are portable and replicable.

## 2.5 Pattern-based verification of ROS 2 timing

In our previous work (Dust et al., 2023a), we proposed a pattern-based verification approach for analyzing the execution behavior of ROS 2 systems using UPPAAL. This approach focuses on verifying two key properties: callback latency and input buffer sizes.

**Callback Latency**: This is defined as the maximum time between the release of a callback instance and the completion of its

```
// Executor start_exp
// Release Times for WallTimeCallbacks
const int releasesSUBSCRIBER0[MAXX]=
{43,43,43,43,0,0,0,0,0,0};
// Executor
ExV1 = ExecutorExV1(StopTime);
// Callbacks
SUBSCRIBER0 =
WallTimeCallback(0,5,4,releasesSUBSCRIBER0,
SUBSCRIBER,1000);
// System Definition
system ExV1 < SUBSCRIBER0;
```

Listing 1. Example of UPPAAL System Initialization.

execution. Ensuring low callback latency is crucial for maintaining the responsiveness of the system.

**Input Buffer Size**: This property verifies that the input buffer is large enough to handle incoming data for a given system configuration. Adequate buffer sizes prevent data loss and ensure smooth data flow within the system.

To facilitate the verification process, we create three types of UPPAAL templates to represent ROS 2 nodes:

- **Wall-Time-Callbacks**: These are callbacks that are released at specific times.
- **Periodic Callbacks**: These are callbacks that are released periodically.
- **Executors**: These represent different versions of the ROS 2 executor, which schedules and executes callbacks.

These templates can be composed to model a ROS 2 system, allowing for exhaustive verification of timing properties. Listing 1 illustrates an example of UPPAAL system initialization:

The listing above shows an example of how a ROS 2 system can be initialized using UPPAAL templates. In this example:

- An array is created to hold the release times for a Wall-Time-Callback. In this case, the callback is released four times at 43 m intervals.
- An executor is initialized with a defined stop time, which describes the interval for which the verification will be conducted.
- The Wall-Time-Callback template is instantiated with parameters such as the callback ID, execution time, number of releases, release time array, callback type, and buffer size.
- The system is defined as a composition of the executor and the callback.

This initialization process allows for the simulation and verification of the system's timing behavior using UPPAAL. By modeling the system in this way, we can conduct exhaustive verification to ensure that the system meets its timing requirements and identify any potential issues related to callback latency and buffer sizes. The actual verification of such properties happens in the UPPAAL verifier through checking the states of defined variables.

The pattern-based verification approach provides a structured and systematic method for analyzing the timing behavior of ROS 2 systems, making it easier for developers to ensure the correctness and reliability of their systems.

### 2.6 Eclipse modeling framework (EMF)

The Eclipse Modeling Framework (EMF) (Steinberg et al., 2008) is a widely used tool for model-driven engineering (MDE). EMF provides a framework for defining metamodels and generating code from models. It supports model-to-model and model-to-text transformations, enabling the automation of various development tasks. In our work, we utilize EMF to create metamodels that represent different abstractions of ROS 2 systems, facilitating the automation of formal verification. EMF allows the definition of metamodels that are instances of the Ecore metamodel, which can be used to create models representing system components and their interactions.

### 2.7 End-to-end timing analysis

End-to-end timing analysis is crucial for ensuring the correct functionality and safety of autonomous systems, particularly in real-time applications. It involves analyzing the timing behavior of cause-effect chains, which represent sequences of reactions from a cause (e.g., sensing) to an effect (e.g., actuation). Two key metrics in end-to-end timing analysis are the *maximum reaction time* (the maximum time for the system to react to an external input) and the *maximum data age* (the maximum time between sampling and the output being based on that sample) (Teper et al., 2022).

In ROS 2 systems, end-to-end timing analysis can be challenging due to the combination of time-triggered and event-triggered components. Existing methods for periodic and sporadic task systems are not directly applicable to ROS 2. Therefore, we propose new UPPAAL templates that resemble the end-to-end timing analysis presented in (Teper et al., 2022). These templates are used to model and verify the timing behavior of ROS 2 systems, to ensure that they meet the required timing constraints.

## 3 Proposed methodology

In this section, we present a methodology designed to automate the formal verification of ROS 2-based systems. The methodology integrates execution traces generated at runtime, model-driven development, and the composition of formal model declarations and verification. This methodology aims to decouple the process of formal modeling from system development while ensuring traceability, enabling users to apply formal verification without requiring extensive domain expertise.

### 3.1 Definition of users

The primary goal of this methodology is to simplify the verification process for robotic systems. The intended end users are robotics developers who may have limited knowledge of modeling and formal verification. Hence, the aim is to allow robotics engineers to perform formal verification with limited learning effort.

Practically, the methodology is designed to enable the design of an extensible toolchain, where domain experts, such as formal verification specialists and modeling engineers focus on the aspects where they can contribute most. Once such a toolchain following the developed methodology is developed, implemented, and set up, robotics engineers should be able to operate it with limited learning effort. The definition of users and maintainers of the toolchain is essential to analyze potential modularization, as stated in RQ2.

## 3.2 Architectural overview

The toolchain comprises four main architectural components.

1. System Implementation Layer
2. Tracing Layer
3. Modeling Layer
4. Verification Layer

Figure 3 provides an overview of the methodology, showing the included layers and their connections. Each layer is marked in a different color. It can be seen that with Start A and Start B, there are two starting points for the potential application of the methodology. They refer to the two possible application approaches, namely, the verification of already implemented, executable systems, and the verification of conceptual system design. A more detailed overview of the components of each layer can be found in Figure 4. In the figures, MM stands for Meta-Model, M stands for Model, and T stands for Transformation. In Figure 4, boxes stand for artifacts and system components such as models and traces, while ellipses stand for actions such as transformations. The subsequent sections explain the layers in more detail.

### 3.2.1 System Implementation Layer

The ROS 2 Layer 0 (light blue) represents the ROS 2 system implementation, encompassing both static information (Figure 4a) (dark blue) and dynamic information (white). Static information includes system components such as nodes, timers, subscriptions, publishers, services, and clients. Dynamic information involves runtime data such as callback execution times, timer release times, and message passing instances that describe the system behavior during execution.

### 3.2.2 Tracing layer

The Tracing Layer 1 (green) utilizes tracing to generate execution traces from the running system. An overview of the elements in this layer is presented in Figure 4b. Generated traces contain both static and dynamic information, and are then transformed into a human-readable ROS 2 Data Model representation (M0). Additionally, customized analysis can be performed during the initial analysis of the traces, such as message flow analysis, as proposed in (Bédard et al., 2023). Based on the generated traces and analysis, a detailed visualization of system components and their interactions is possible. The traces are an important part to answer RQ1, as they enable automated determination of system parameters.

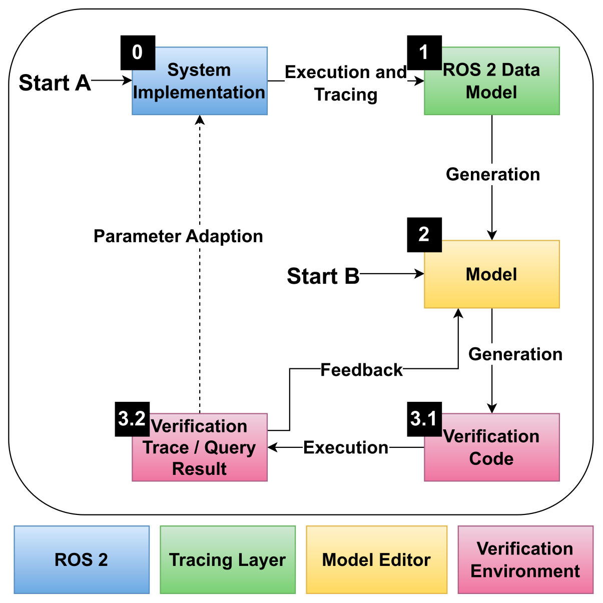


FIGURE 3
Architectural overview of the proposed methodology for automating verification of ROS 2-based systems. The architecture includes layers for system implementation, tracing, modeling, and verification.

### 3.2.3 Modeling layer

The Modeling Layer 2 (light yellow) shown in Figure 4c involves the use of a modeling framework to create and utilize different metamodels and transformations. The metamodels allow the definition of models to model the system from different perspectives. The following exemplary types of metamodels allow representation of various abstractions of the ROS 2 system:

- **MM1 Metamodel**: Input Metamodel: Maps the ROS 2 Data Model to an model for use in a model editor, allowing detailed analysis and traceability.
- **MM2 Metamodel 2**: System Model: Allows modeling of the system from a perspective towards the formal model. Parameters not needed for verification might be excluded, and component links might be made using a different approach. This model is used to decouple the formal modeling further from the tracing and parsing. Hence, if a different formal model is introduced, MM2 has to be exchanged. Nevertheless, there can be multiple different implementations of MM2 to allow the generation of different formal model representations.

While it is possible to generate the verification code from the traces directly, it is beneficial to include different metamodels during the generation steps. This allows traceability throughout the generation process, which might lead to a better understanding of parameters. Furthermore, this allows for the testing of different parameters without the need to adapt and rerun the system. Additionally, the introduction of incremental steps is essential to achieve modularization as stated in RQ2. Note that in a modeling environment, there might be multiple different implementations of MM2 to allow the application of different formal models.

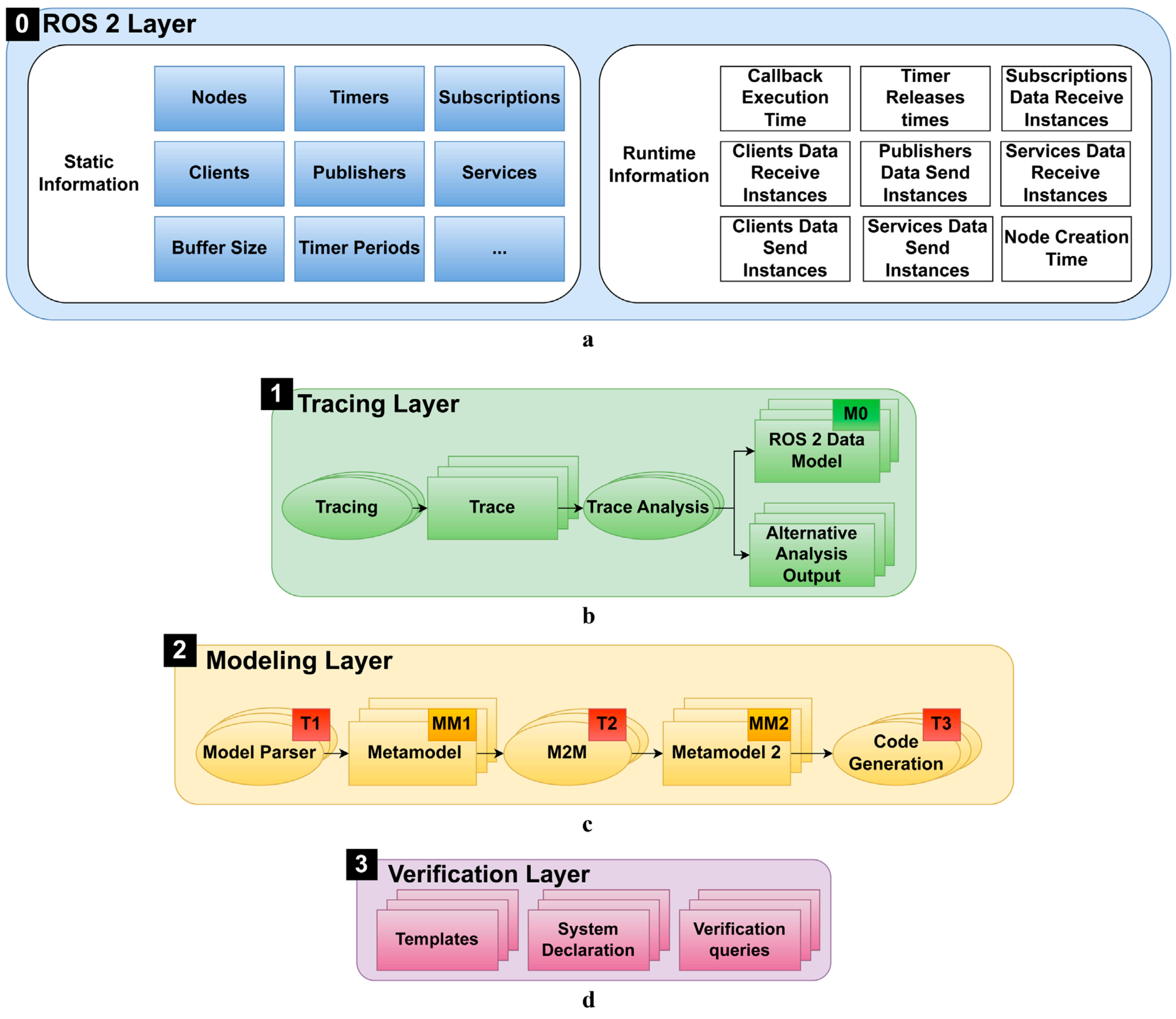


FIGURE 4
Detailed overview of the elements in the component layers. (1) System Implementation Layer. (2) Tracing Layer. (3) EMF Modeling Layer. (4) Verification Layer. **(a)** Overview of the ROS 2 system implementation, highlighting the static and dynamic information captured during system execution. **(b)** Tracing layer overview, showing the process of generating and analyzing execution traces to create the ROS 2 Data Model. **(c)** Modeling Layer overview, illustrating the different metamodels and transformations used to automate the verification process. M2M stands for model-to-model transformation. **(d)** Verification Layer overview, showing the generation of UPPAAL artifacts from EMF models for formal verification.

To simplify the transformation between the different model representations, three types of transformations can be employed:

- **Model Parsing (T1)**: Transforms textual model descriptions into model representations specific to the chosen modeling environment, ensuring compatibility with the metamodel definitions.
- **Model to Model Transformation (T2)**: Converts one model representation to another, facilitating different sets of features and abstractions.
- **Code Generation (T3)**: Translates models into executable code, enabling the generation of formal model declarations and verification queries.

The shown transformations can be automated using designated tools. Such automation potential is important to answer RQ1.

### 3.2.4 Verification layer

The Verification Layer 3 (violet) shown in Figure 4d contains three main elements. First of all, there are the formal model templates, which are created by a verification expert; the templates capture the behavior of the system components, formally. The templates can be composed into a system, in the system declaration. Using the toolchain, the system declaration can be produced through model-to-text transformations. Verification queries can be executed in the verifier. The verifier allows verification of the properties of a system specified by such queries. They can be predefined as templates and adapted to the chosen notation and naming during the model-to-text transformations. When applying the methodology following the architectural overview in Figure 3, upon creation of the formal model and queries in form of a compatible file (3.1), the outcome of the verification is given as potential execution traces and the query results (3.2). In the proposed methodology, multiple different implementations of Templates, Systems Declarations, and Verification Queries can be employed. This is essential to answer RQ3.

### 3.2.5 Relation of components and modularization

In the previous subsections, we divided the methodology into four layers. Each of the layers consists of multiple components.

While the metamodels proposed in the Tracing Layer and the Modeling Layer are not strictly needed to enable automation, in this paper, they are introduced to enable modularization and decoupling of domain knowledge in the verification processes.

Generally, the boxes shown in Figures 4b–d show artifacts that result from specific actions shown as ellipses.

Two consecutive artifacts in the methodology are related by the fact that the set of attributes of the first artifact is an extension of the attributes of the second artifact. Hence, a transformation reduces the set of parameters while changing the structure of the model. As an example, one can transform an instance of the class Trace to an instance of the class ROS 2 Data Model by using Trace_analysis library functions. The classes are substitutable by any other class that obeys the extends mechanism. However, the transformation needs to be adapted, provided that the inherited attributes of the substituted class change. In the Modeling Layer, multiple, different instances of the two metamodels might be created. Any instance of MM1 can be transformed in any instance of MM2, as long as the extends mechanism for the parameters is true. Hence, in case a preceding model is adapted or exchanged, the transformation only needs to be adapted when the inherited attributes change.

The layered approach with the different artifacts allows for reducing complexity and shifting the goal of the models in defined steps through an adaptation of the modeled system architectures incrementally. While the MM1 reflects more the architecture of a ROS 2 system, the MM2 reflects closer the architecture of the proposed formal model. Each parameter reduction and architectural model change is conducted incrementally, reducing the need for complex domain knowledge while allowing traceability. This reflection is essential as an informal proof of modularization to tackle RQ2.

## 3.3 Application of the methodology

To apply the methodology, a toolchain is needed that implements the required components, such as metamodels and transformations. Once the necessary metamodels and transformations are established, the end user (robotics developer) can perform the verification by following the outlined approach. The general flow of applying the methodology is shown in Figure 3, and the workflow can start from two points: verifying legacy systems (Start A) or conceptual systems (Start B).

### 3.3.1 Verification of legacy systems

Starting with a ROS 2 system implementation (Start A), tracing is used to generate runtime execution traces. These traces are transformed into the ROS 2 Data Model using analysis tools. A parser then converts the ROS 2 Data Model into a model. Multiple model-to-model transformations can be applied within the EMF environment to generate a formal model based on predefined templates. The generated formal model definition is used for formal verification, and the results can guide parameter adjustments in the model or the real system. This iterative process allows for thorough testing before implementing changes in the actual system.

### 3.3.2 Verification of new systems

For new system designs (Start B), the system can be modeled directly using the created metamodels. This approach allows for system definition without existing source code. By modeling a system using a modeling environment, only the necessary parameters for verification need to be determined, and the code generation can automatically produce the formal model declaration. Iterative verification of system parameters can then be conducted using the models and formal modeling tool, ensuring a robust design before implementation.

## 3.4 Toolchain setup, development, and extension

As described in the previous section, a toolchain is needed to enable the application of the methodology following Figure 3 to verify ROS 2-based applications. The design and implementation of such toolchain is done in a different order than the actual verification process shown in Figure 3. The modularity of components in the methodology allows domain experts to focus on their area of expertise when implementing the toolchain. Each domain expert is responsible for the implementation of specific components and only interacts with other domain experts to implement the connectors, such as transformations. Below, we give an overview of the implementation process of a toolchain that involves several steps, each requiring specific expertise:

Step 1: Development of Formal Models (Formal Methods Expert): In a first step, the formal model component templates need to be defined, created, and tested in the formal modeling tool.

Step 2: Determination and setting of Parameters (Formal Methods Expert/Robotics Expert): Next, the needed input parameters for the formal verification, and how they can be obtained (e.g., through ROS2_tracing), need to be identified. If they cannot be obtained through the mainline analysis tools, additional analysis methods may be required.

Step 3: Development or Adaptation of Tracing Tools (Robotics Expert): In a following step, the tracing needs to be adapted to capture the parameters as required.

Step 4: Development or Adaptation of Metamodels (Software Engineering/Modeling Expert): The next step incorporates an update of the ROS 2 data model and creation of the metamodels needed to cover the components needed for formal verification.

Step 5: Development or Adaptation of Transformations (Software Engineering/Modeling Expert): Implementation of the parsing, the model-to-model, and model-to-text transformations. Create verification models in the formal verification environment and corresponding metamodels.

When creating a toolchain, we assume the goal is the verification using one specific formal model representation. Nevertheless, in case different formal model approaches are to be used, the toolchain can be extended by extending the second and third layer. While the first metamodel in the layer is for a seamless parsing of a trace output to a model in the model environment, the second metamodel enables the transformation towards the formal model representation. Hence, when adding a formal model representation, a further implementation of the second metamodel has to be introduced to adapt to the new formal model. The first metamodel and the parsing can be reused. The possibility of domain experts focusing on defined steps with defined connectors with inputs and outputs in between different layers, is evidence of modularization and helps answering RQ2.

## 4 Toolchain implementation and application

In this section, we apply and evaluate the proposed methodology through the design, implementation, and application of a toolchain. In the first step, we design and implement a toolchain following the process explained in Section 3.4.

### 4.1 Toolchain design

Following the four layers of the proposed methodology, the designed toolchain comprises four main architectural components: three tools (ROS2_tracing, Eclipse/EMF, and UPPAAL) and the actual system implementation.

An overview of the Tracing Layer, the EMF Modeling Layer and the Verification Layer of the toolchain created in this evaluation is given in Figure 5.

To show the extensibility of the approach, in this paper we implement formal verification based on two approaches of formal modeling. The first approach is verifying callback latency (CBL). The second approach focuses on verification of end to end delays (E2E). Hence, the verification layer comprises of two different UPPAAL models. Each UPPAAL model consists of the UPPAAL timed automata templates, the UPPAAL systems definition and the verification queries.

The parsing layer consists of the tool ROS2_tracing and generated outputs created by the python libraries of tracetools_ analysis, such as custom graph visualization.

The modeling layer contains the EMF Data MM, which allows direct parsing of the ROS 2 data model to a model in Eclipse. The model can be reused for the generation of both formal models. Next, the specialized model representation has to be designed for each of the verification approaches individually. Hence, the EMF Modeling Layer contains one metamodel for the CBL and one metamodel for the E2E. Besides the metamodels, the EMF modeling layer contains model transformations.

We implement the transformations,T1,T2.1 and T31 to an extent to be conducted automatically using the chosen tools. Transformations T2.2and T2.3are conducted by hand. Nevertheless, if a feature is contained in the preceding model, it can be contained in the next model after the transformation. Furthermore, some features that are not directly contained can be calculated during the transformation from the parameters that are contained.

In the following, we explain the implementation of the toolchain and the order in which it is designed. Furthermore, we explain the parameters and features that are contained in each element in Figure 5.

### 4.2 Toolchain implementation

In the next sections, we follow the workflow for creation of a toolchain following the methodology presented in Section 3.4.

#### 4.2.1 Step 1: Formal models in UPPAAL

To demonstrate the methodology and toolchain we utilize two different kinds of UPPAAL models for verification. The first kind of UPPAAL models has been created previously (Dust et al., 2023a), and are explained in Section 2. The model focuses on the verification of latency and callback size of a single callback in a ROS 2 system.

The second kind of UPPAAL models (E2E) used for evaluation in this paper are created during the implementation of the toolchain. The models aim to allow verification of end-to-end (E2E) latency as proposed in the literature (Teper et al., 2022). Generally, our modeling approach is to create a chain that contains all the components of the original chain and adds delays during the execution that are equally long as the maximum latency for each component. Hence, when executing the model, at the end of the simulation of the last component, the system time will be according to the maximum latency of the chain. In related work (Teper et al., 2022), the authors define six types that can describe a callback based on the function in a processing chain: Sensor, Filter, Timer Fusion, Subscription Fusion, Timer Actuator, and Subscription Actuator. For each of the types of callbacks, we propose UPPAAL templates that can be used to model the end-to-end delays and are shown in Figure 6. In what follows, we give an overview of the proposed templates, where the details, such as included parameters, are explained in Section 4.2.2.

**Sensor**: A sensor node is the start of a chain and sends a message periodically. According to (Teper et al., 2022) the maximum latency is defined as the period minus the maximum execution time plus two times the execution time of all callbacks in the executor. To model the callback in UPPAAL, the callback starts in a location where it waits for the time to elapse until the maximum latency to finish its execution. When the execution of the sensor callback has ended, a following callback is triggered. This is modeled through a synchronized channel to the following callback that is initiated by the sensor.

**Filter**: The subscription actuator can be modeled like a filter, as the maximum latency is equal to two times the sum of all callback execution times (Teper et al., 2022). The callback is modeled by triggering the execution through the synchronized channel

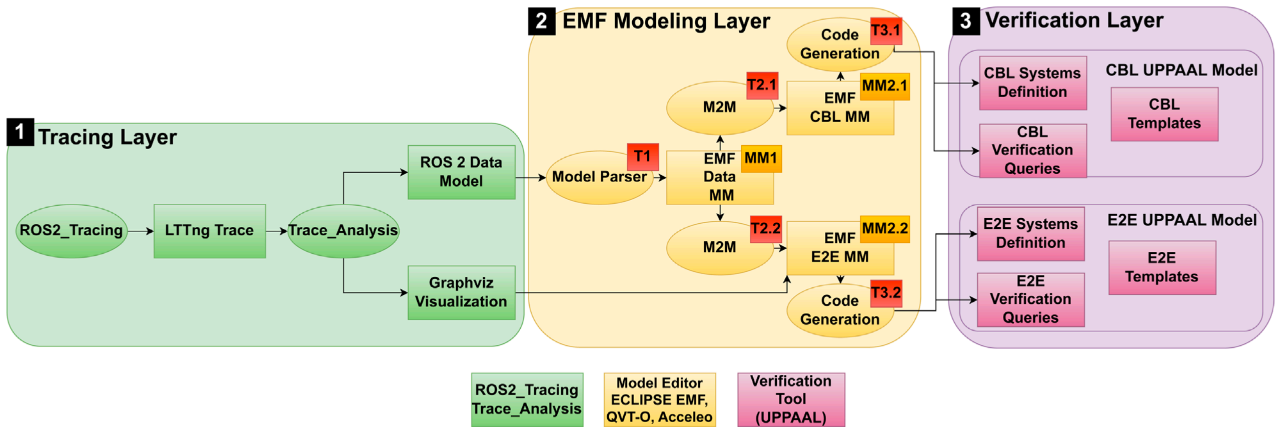


FIGURE 5
Overview of the proposed toolchain and its components. Note that Layer 0 (ROS 2 Layer) is omitted for space reasons.

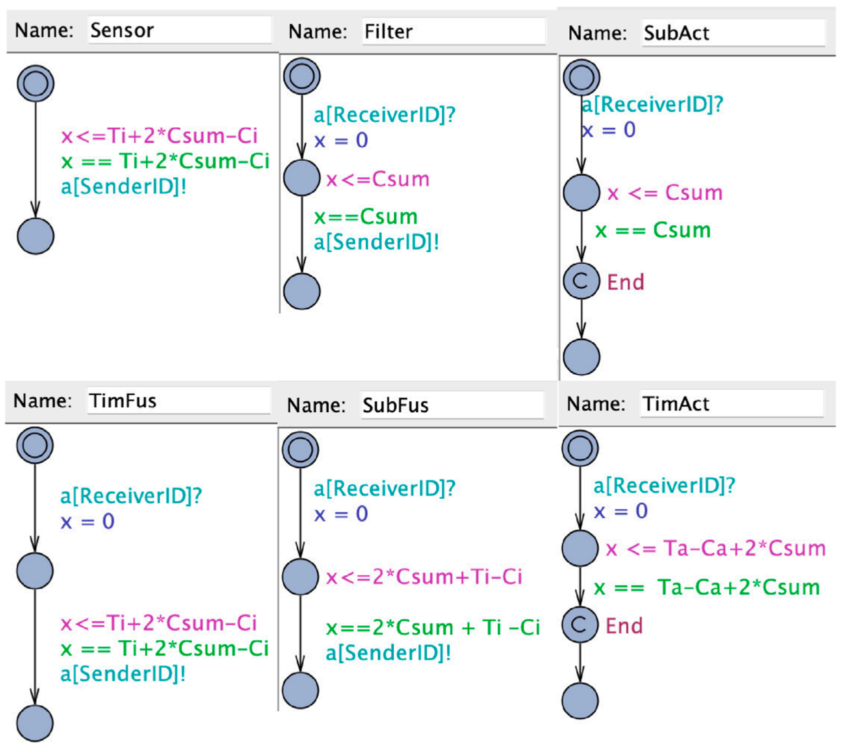


FIGURE 6
UPPAAL templates for the callbacks.

through the preceding callback. Upon triggering of the execution, the callbacks stay in the execution location until the maximum blocking time has elapsed. Upon finishing the execution, the next callback is triggered through a synchronized channel.

**Timer fusion**: Timer fusion in a chain basically consists of two relevant callbacks. The first callback receiving a message from the preceding node can be modeled as a filter. The second callback is a timer callback that has the same latency as a sensor (Teper et al., 2022). Hence, we model the callback as a sensor, but with the difference that the callback is triggered by another callback and upon execution triggers another following callback.

**Subscription Fusion**: When modeling a subscription fusion, there are two different paths possible for a chain. If the subscription that triggers the following node of a fusion lies within the same chain (is triggered by a callback in the chain that is modeled). In this case, the callback can be modeled as a filter node.

If the callback that outputs the fusion result is not contained in the same chain, a second chain has to be introduced to determine the maximum latency. As each chain starts with a sensor node, we model the start of that chain as a sensor that is triggered by another node. Following, each element of the same chain including the fusion callback has to be included separately as a fusion callback.

**Timer actuator**: The timer actuator actually consists of two callbacks that have to be modeled accordingly in our UPPAAL representation. The first callback is triggered by the preceding node in the system and resembles a subscription node. In our example, the callback has to be modeled as a filter. The final callback is a timer callback that executes the actuator. This callback has the same latency as a sensor callback (Teper et al., 2022). Hence, we model the callback as a sensor, but with the difference that the execution is triggered by a preceding callback. Upon finishing the execution, the callback passes through an End location that is used for the verification query.

**Subscription actuator**: The subscription actuator can be modeled like a filter, as the maximum latency is equal to two times the sum of all callback execution times (Teper et al., 2022). The only difference is the actuator being the end of the chain. Hence it does not need to trigger another execution of a following callback.

The shown templates are sufficient to model processing chains of callbacks and determine the upper bound for the latency by checking the global system time while the Actuator callback passes through the End location.

### 4.2.2 Step 2: Determining parameters, features, and formal model declaration

In this step, we identify the parameters required to instantiate the UPPAAL templates for model verification. These parameters are essential for accurately modeling the system's behavior and ensuring the correctness of the verification process.

#### 4.2.2.1 Buffer overflow UPPAAL templates

The legacy UPPAAL model includes three types of templates: BufferOverflow, Executor, and Callbacks. Each template requires specific parameters to be instantiated, as follows.

##### 4.2.2.1.1 Executor template

- **stoptime**: The time at which the executor stops executing. This parameter defines the duration for which the verification is conducted.

##### 4.2.2.1.2 PeriodicCallback template

- **id**: A unique identifier for the callback.
- **execution time (Ci)**: The time required to execute the callback.
- **period (Ti)**: The interval at which the callback is triggered.
- **type**: The type of callback (e.g., timer, subscriber).
- **buffer size**: The size of the buffer associated with the callback.

##### 4.2.2.1.3 SporadicCallback template

- **id**: A unique identifier for the callback.
- **execution time (Ci)**: The time required to execute the callback.
- **amount of releases**: The number of times the callback is released.
- **release array**: An array specifying the release times of the callback.
- **type**: The type of callback (e.g., timer, subscriber).
- **buffer size**: The size of the buffer associated with the callback.

#### 4.2.2.2 End-to-end (E2E) timing analysis templates

For end-to-end timing analysis, we use several templates to model different components of the system. Each template requires specific parameters to capture the timing behavior accurately.

##### 4.2.2.2.1 Sensor template

- **Csum**: The sum of the execution times of all callbacks in the system/executor.
- **Ci**: The execution time of the callback.
- **Ti**: The period of the callback.
- **SenderID**: A unique identifier for the callback that sends information.

##### 4.2.2.2.2 Filter template

- **Csum**: The sum of the execution times of all callbacks in the system/executor.
- **ReceiverID**: The identifier of the callback that receives information.
- **SenderID**: The identifier of the callback that sends information.

##### 4.2.2.2.3 SubFus template

- **Csum**: The sum of the execution times of all callbacks in the system/executor.
- **Ci**: The execution time of the callback.
- **Ti**: The period of the callback.
- **ReceiverID**: The identifier of the callback that receives information.
- **SenderID**: The identifier of the callback that sends information.

##### 4.2.2.2.4 TimFus template

- **Csum**: The sum of the execution times of all callbacks in the system/executor.
- **Ci**: The execution time of the callback.
- **Ti**: The period of the callback.
- **ReceiverID**: The identifier of the callback that receives information.
- **SenderID**: The identifier of the callback that sends information.

##### 4.2.2.2.5 SubAct template

- **Csum**: The sum of the execution times of all callbacks in the system/executor.
- **ReceiverID**: The identifier of the callback that receives information.

##### 4.2.2.2.6 TimAct template

- **Csum**: The sum of the execution times of all callbacks in the system/executor.
- **Ci**: The execution time of the callback.
- **Ti**: The period of the callback.
- **ReceiverID**: The identifier of the callback that receives information.

#### 4.2.2.2.7 Explanation of parameters

- **Ci (execution time)**: The time required to execute a callback.
- **Ti (period)**: The interval at which a callback is triggered.
- **SenderID**: A unique identifier for the callback that sends information, ensuring it can be correctly identified.
- **ReceiverID**: Matches the receiving callback with the sender, ensuring proper communication between callbacks.
- **Csum**: The cumulative execution time of all callbacks within the system or executor, used to assess overall system load.

After determining the described parameters, we can instantiate the UPPAAL templates to create a model that allows for formal verification of buffer overflows and end-to-end latency, via model checking.

### 4.2.3 Step 3: Setup of ROS2_tracing and graph analysis

In a trace generated by ROS2_tracing, the execution of each individual callback is contained. Hence, the callbacks and their executions can be mapped from the tracing output to the model. The timing information contained in the traces allows the calculation of the maximum execution time for each callback. Additionally, for timers, the configured period is recorded in the traces. By aggregating the execution times of all callbacks, the total execution time for the system can be calculated, with the assumption that all callbacks are executed within the same executor.

To model the callback chain, we utilize the tracing information to identify publishers and subscribers, along with their preceding and following nodes. The primary challenge lies in determining the type of each callback for accurate modeling. In the initial automated transition, we categorize all receiving and sending callbacks as filters, all timers as sensors, and all sinks as actuators. A message flow analysis is conducted to visualize the internal connections and relationships, which helps in manually constructing the graphs.

### 4.2.4 Step 4: Development of metamodels

To follow the process shown in Figure 5, we implement three different Metamodels that can be used to create models containing the information needed to automate the verification. MM1 is used to parse the tracing output with all its information into a model of the same architecture to allow traceability.

MM2 is the EMF CBL Metamodel allowing to create models that resemble the system architecture for individual node verification and MM3 is the EMF E2E metamodel allowing modeling of a system to transform it into UPPAAL code for verification of the E2E latency. In the following, we show the implementation of the metamodels.

#### 4.2.4.1 Metamodel 1 - Eclipse Data Metamodel

In the initial step, we develop the EMF Data metamodel, which includes all system components specified by the ROS 2 Data Model, such as subscriptions, callbacks, and timers, represented as classes with parameters as attributes. Figure 7 shows an excerpt from the metamodel implementation. The yellow boxes indicate the classes in the metamodel, with arrows illustrating the dependencies between them. All classes representing system components are child objects of a master class that represents the entire system. Although in a ROS 2 implementation, components like Publishers are contained within Nodes, in this metamodel, the association of Publishers to Nodes is managed by identification handlers modeled as attributes. This approach aligns the representation of the ROS 2 Data Model with the EMF data model.

#### 4.2.4.2 Metamodel 2 - EMF CBL Metamodel

The second metamodel represents the UPPAAL templates within the EMF framework, which are utilized for formal verification. An overview is provided in Figure 8.

In this metamodel, each of the three UPPAAL templates (Executor, WallTimeCallback, and PeriodicCallback) is represented as a distinct class, which are components of a system. As detailed in Section 2, each template includes parameters such as *ids* and buffer sizes. Additionally, the metamodel defines datatypes for attributes like callback type and executor version, which can be specified in a model instance.

Although the toolchain proposal mentions the modeling of requirements such as maximum callback latency, this aspect is reserved for future work.

#### 4.2.4.3 Metamodel 3 - EMF E2E Metamodel

In the created metamodel in Figure 9, the callbacks are modeled as a single class contained in an executor that is part of a system. The callbacks are distinguished by their Type, which is a parameter. Furthermore, the parameters needed to initialize the UPPAAL model such as execution times and the sum of callback execution times. This model allows the representation of the features needed for the end-to-end verification.

### 4.2.5 Step 5: Implementation of model-to-model transformations

#### 4.2.5.1 T1: Model parsing - ROS 2 to EMF Data Metamodel

We implement the parsing T1 as a python function in Trace_Analysis. The function takes the ROS 2 Data Model and creates an XML file that can be imported as a model (using the EMF Data Metamodel) in the Eclipse workspace. The parsing is done by reading the data of the ROS 2 Data Model and printing them in an XML document with the desired formatting of the EMF Data Model.

#### 4.2.5.2 T2.1: Model-to-Model - EMF Data Metamodel to EMF CBL Metamodel

In this model-to-model transformation, T2.1 the components of the EMF Data Model are mapped to the components in the verification model using QVT-O. QVT-O is an operational mapping language (Eclipse Foundation, 2025b). In the first step, the periodic timers are mapped to periodic callbacks with the type attribute set to TIMER. The period and execution time is extracted from the attributes in the EMF Data Model and passed to the verification model. Furthermore, for timers, the buffer size is set to one. The second mapping is between the subscription callback and the wall-time-callback of the type SUBSCRIBER. A subscription callback is released on the reception of data. As neither the release time of the callback, nor the reception of the data is initially contained in the system traces, we map the publishing time of the data in the topic the callback is subscribed to as the release time. Furthermore, we pass the execution time and buffer size as further parameters.

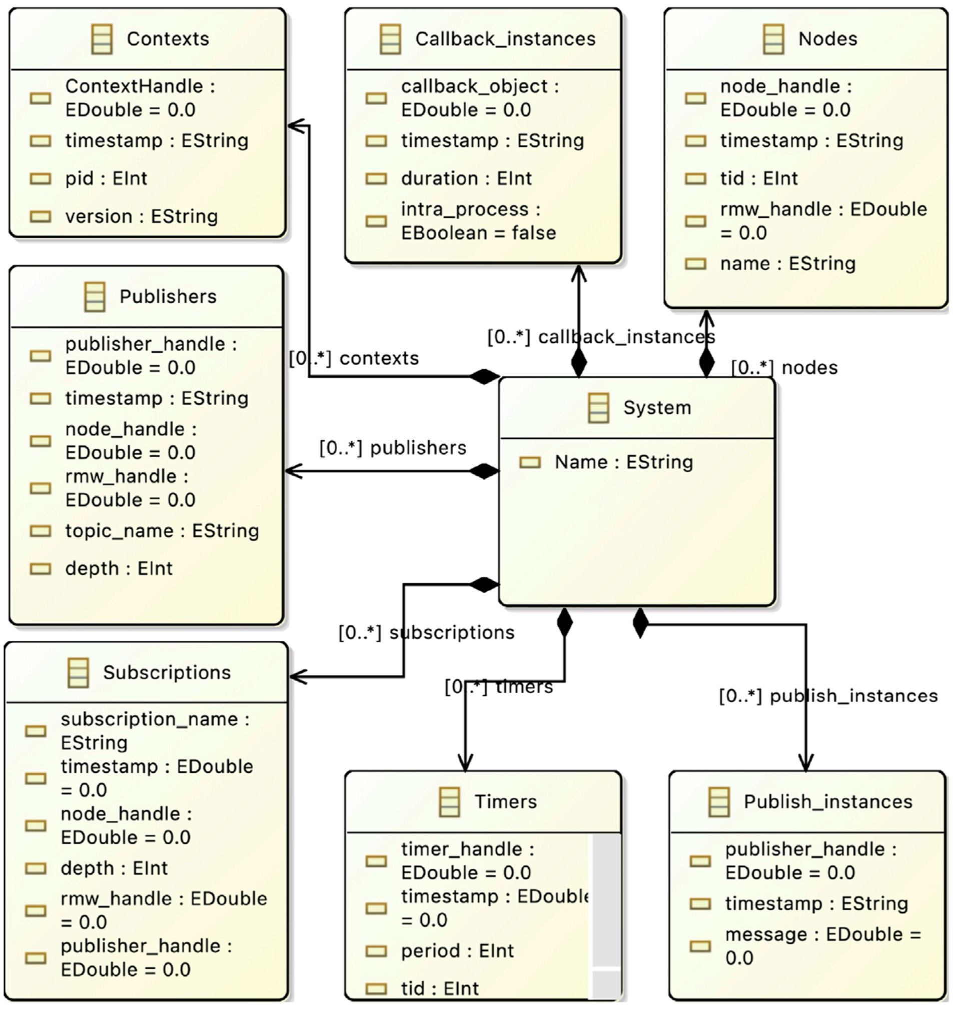

Contexts
ContextHandle : EDouble = 0.0
timestamp : EString
pid : EInt
version : EString
Callback_instances
callback_object : EDouble = 0.0
timestamp : EString
duration : EInt
intra_process : EBoolean = false
Nodes
node_handle : EDouble = 0.0
timestamp : EString
tid : EInt
rmw_handle : EDouble = 0.0
name : EString
Publishers
publisher_handle : EDouble = 0.0
timestamp : EString
node_handle : EDouble = 0.0
rmw_handle : EDouble = 0.0
topic_name : EString
depth : EInt
System
Name : EString
[0..*] contexts
[0..*] callback_instances
[0..*] nodes
[0..*] publishers
[0..*] subscriptions
[0..*] timers
[0..*] publish_instances
Subscriptions
subscription_name : EString
timestamp : EDouble = 0.0
node_handle : EDouble = 0.0
depth : EInt
rmw_handle : EDouble = 0.0
publisher_handle : EDouble = 0.0
Timers
timer_handle : EDouble = 0.0
timestamp : EDouble = 0.0
period : EInt
tid : EInt
Publish_instances
publisher_handle : EDouble = 0.0
timestamp : EString
message : EDouble = 0.0

FIGURE 7
Data Metamodel in Eclipse.

#### 4.2.5.3 T3.1: Model-to-Text - EMF CBL Metamodel to UPPAAL code

With the model-to-text transformation T3.1 using the tool Acceleo (Eclipse Foundation, 2025a), the EMF verification model is translated into the UPPAAL code. Therefore, the classes contained in the EMF verification model are mapped to specific code snippets, e.g., representing the UPPAAL template instantiation. Then, the code is dynamically filled with the needed parameters based on the class attributes.

#### 4.2.5.4 T2.2, T3.2: EMF Data Metamodel to EMF E2E Metamodel to E2E UPPAAL code

The transformations T2.2 and T3.2 are needed to fully automate the process of verifying ROS 2 applications using the E2E UPPAAL models. As we evaluate the automation of such transformations on T2.1 and T3.1, and the repetitive implementation of the transformation as being rather engineering than research, for the sake of simplicity, T2.2 and T3.2 are not implemented by a tool but, in the context of this paper, are conducted manually.

## 4.3 Toolchain application

The application and evaluation of the toolchain is carried out using three ROS 2 systems (Use-Case 1, 2, 3). Use-Case 1 and Use-Case 3 are implemented in source code and executable, while Use-Case 2 is evaluated from a conceptual perspective without actual source code implementation.

We show the automation of formal verification on two examples of formal models in UPPAAL, each focusing on a different set of properties to verify. The implementation shows how to set up the toolchain and its main components in the context of verification of *buffer overflow*, *callback latency* and *end-to-end delays* for ROS 2 processing chains.

While for the verification of the buffer overflow and callback latency, we reuse UPPAAL templates that have been proposed in related work, for the verification of end-to-end timing analysis, we propose new UPPAAL UTA templates. We implement exemplar metamodels to demonstrate the application of the toolchain. For the verification of the buffer overflow, we implement prototypes of the

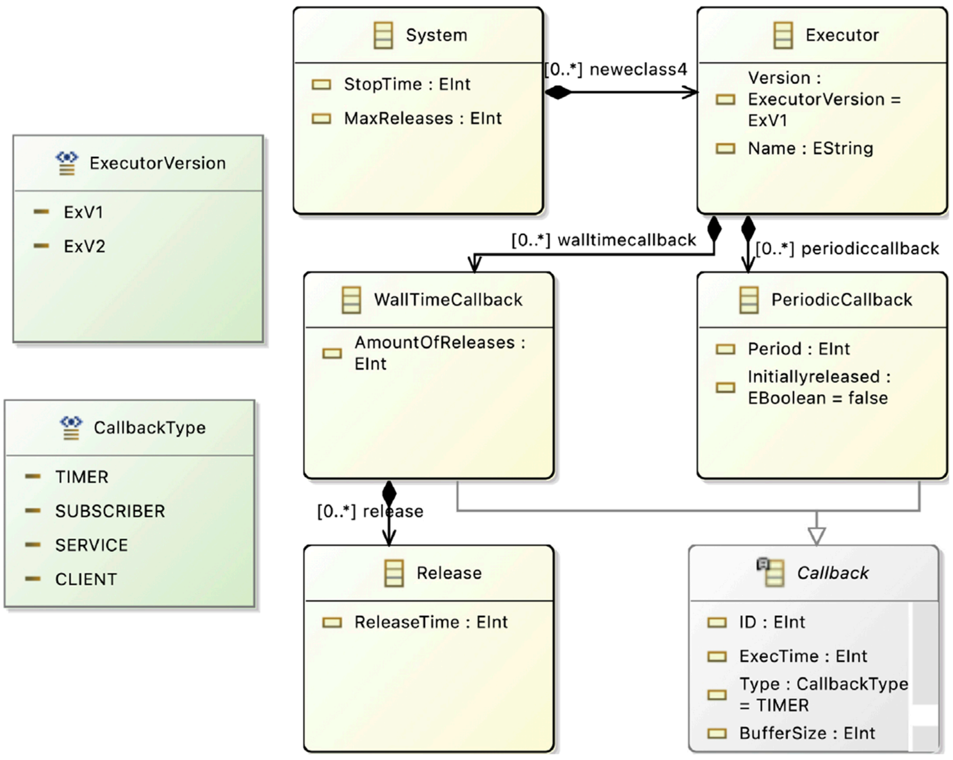


FIGURE 8
Verification Metamodel in Eclipse.

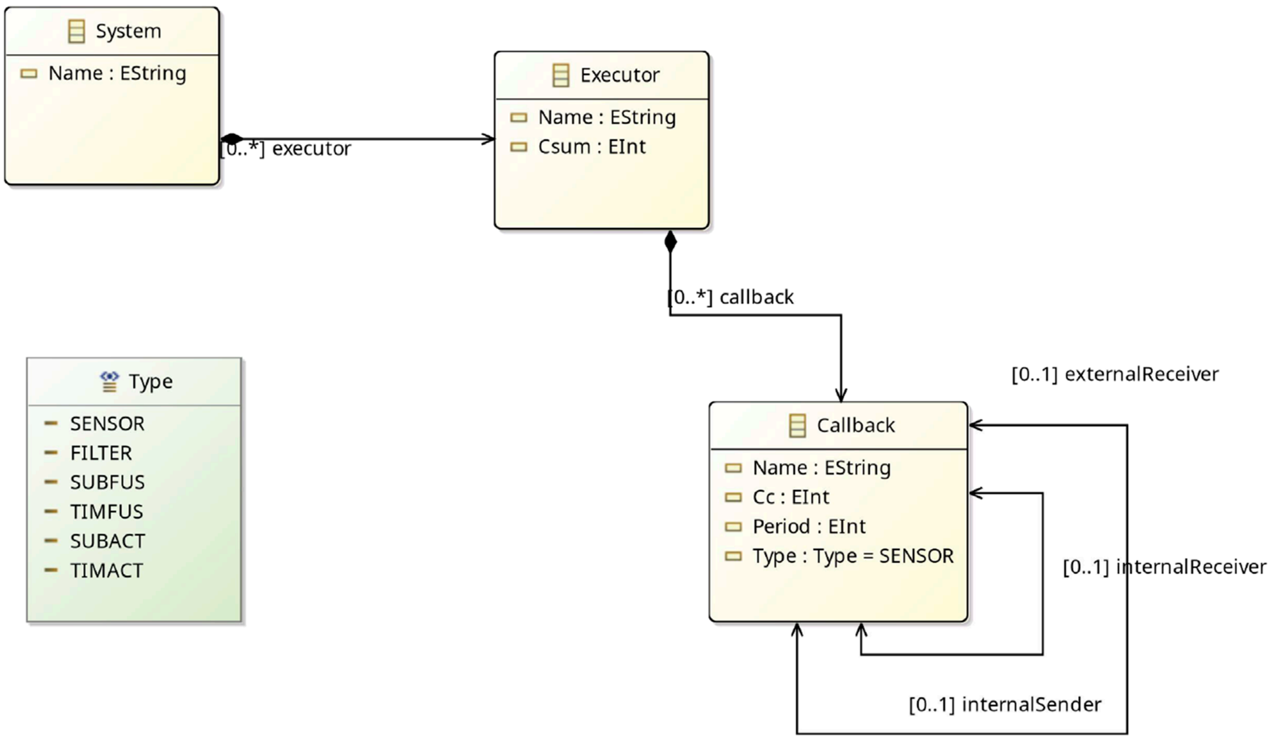


FIGURE 9
E2E Metamodel in Eclipse.

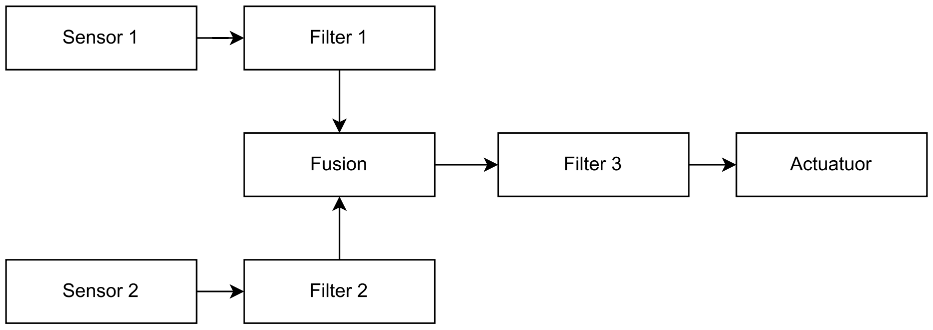


FIGURE 10
System overview.

model parsing, the model-to-model transformation, and the model-to-text transformation. In the second part of verifying end-to-end latencies, we perform the transformations by hand. The created artifacts, such as metamodels, templates, source code, and graphs, are published in (Dust et al., 2025).

### 4.3.1 Use-cases

The first system (Use-Case 1) is a lightweight ROS two implementation of two nodes similar to the setup depicted in Figure 2. The system has been proposed by Casini et al. (2019) and used for real-time evaluation and demonstration of different scheduling approaches of ROS two in Blaß et al. (2021), Dust et al. (2023b), and Dust et al. (2023a). The system offers traceability through controlled execution times and controlled trigger events of the included callbacks. The limited complexity simplifies manual analysis, and hence enables simpler comparison and evaluation of verification and modeling approaches.

The second system (Use-Case 2) is a conceptual ROS two system as given in the evaluation of Teper et al. (2022). The system is used to evaluate the correctness of the created UPPAAL templates for E2E verification. As a part of RQ3 we aim to provide validated UPPAAL models. The correctness of the proposed formal models is demonstrated by repeating the calculations from the case study in Teper et al. (2022). Furthermore, as the system is a conceptual design, as we have no access to the original ROS two code, the approach of verification of conceptual designs (START B) is demonstrated. An overview of the nodes in the system is given in Figure 10. The system consists of two sensors, which contain a ROS two timer each publishing a message at a given interval. The message is received by a filter callback that forwards the message on reception. The two filter messages are fused into one message in the fusion node. A third filter node receives and forwards the fused message. The final message is received by an actuator node. In an actual system, the fusion and the actuator can be implemented in two different ways. The subscription fusion and actuation, and the timer subscription and actuation. Both are shown in Figures 11, 12. In the subscription configuration, the messages are forwarded using the same callback triggered by the subscriptions. In the timer configuration, the messages are received by a subscription callback, and then the final message is published by a different timer callback.

The third system (Use-Case 3) is an ROS two real-time benchmark system (ROS Realtime Working Group, 2025b) and resembles parts of an autonomous driving stack. As a controlled real-world system is used to demonstrate applicability of the proposed methodology. In Figure 13, we show an excerpt of the system, visualized through implemented message flow analysis. For simplicity, we focus the verification of the end-to-end latency on the chains shown in the figure.

### 4.3.2 Application of legacy UPPAAL models and automated transformations on use-case 1

As a first step, we utilize Use-Case 1 to assess and validate our toolchain prototype regarding automation as stated in RQ1, and verification as stated in RQ3. We evaluate the proof of concept by generating UPPAAL code through our proposed workflow, with an example excerpt shown in Listing 1. Instead of manually analyzing the system and calculating potential release times, our approach automatically generates traces and performs model transformations to produce executable UPPAAL code. We then ran this code in UPPAAL and verified the accuracy of the results at each stage.

By focusing on subscription and timer callbacks, the generated traces provided the necessary information to either directly determine or infer the required parameters, such as callback periods, release times, buffer sizes, and execution times. However, some parameters for services and clients, as well as certain subscription callback release times, are currently not captured by ROS2_tracing.

Despite these limitations, the presence of parameters for subscribers and periodic timers demonstrates the feasibility of our toolchain. Additionally, it is possible to add custom trace points, although this requires expert knowledge. We are working on incorporating the necessary trace points into the mainline releases of ROS 2.

Our observations indicate that the model-based verification of ROS two applications can be automated by using system execution traces and model-driven engineering to automatically populate model-based verification templates.

### 4.3.3 Verification of E2E latency of conceptual system design on use-case 2

To implement the system in the modeling layer, we create models in Eclipse containing the different configurations following the grammar of the defined E2E metamodel. An example of such a model for a subscription fusion and the subscription actuation is shown in Figure 14. It can be seen that all callbacks are included in

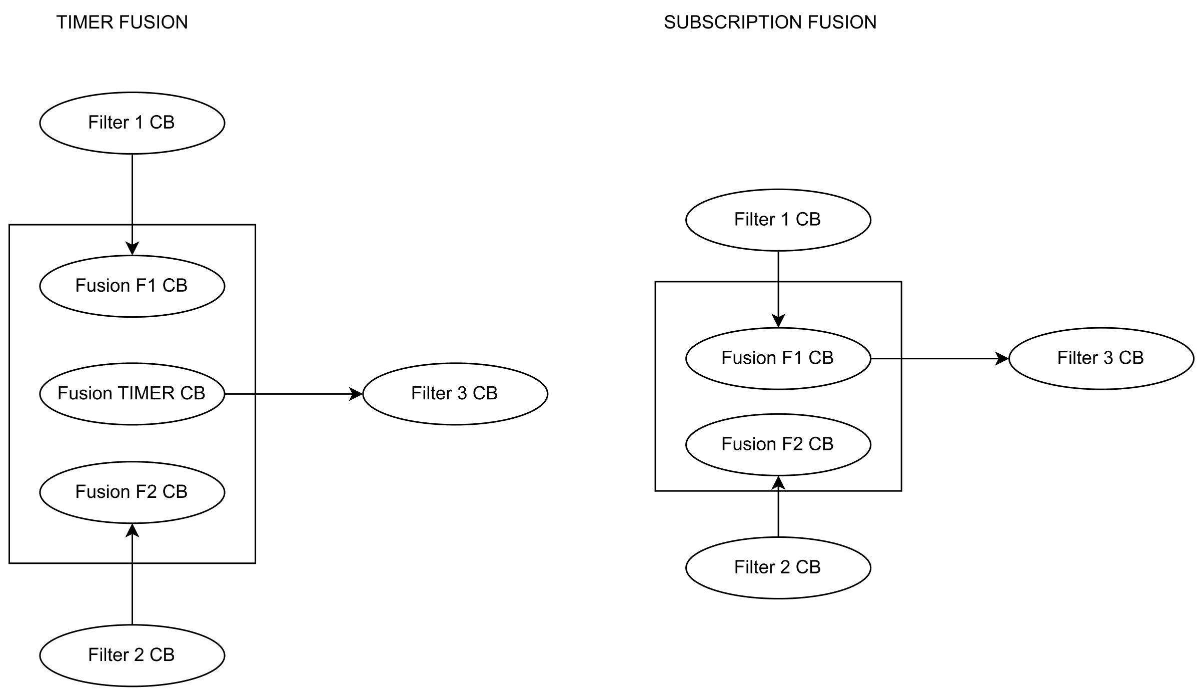


FIGURE 11
Two versions of the fusion node.

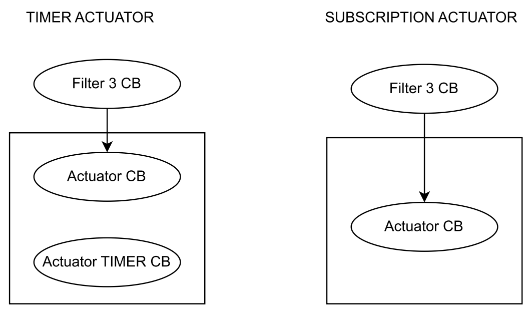


FIGURE 12
Two versions of the actuator node.

the same executor. Furthermore, the callbacks contain information such as the period and type of timers. The callback, whose parameters are shown in the example, is the Fusion F1 CB that receives the data from Filter one and forwards the fused message to Filter 3. The links are done through the connection of the parameters External Receiver and External Sender. The fusion is indicated through the internal sender Sub Fusion 2.

Next, we transform the model into the system instantiation for UPPAAL as shown in Listing 2 for an example with a subscription fusion and subscription actuation. As described, the actual transformation in this example is done by hand, but can be automated through Acceleo model-to-text transformation.

After performing the transformations and modeling in UPPAAL, we run the verification and compare the obtained results for the E2E latency with the results from (Teper et al., 2022). Table 1 shows the system parameters and the results of the verification, which match the results from Teper et al. (2022). Hence, we validate the correctness of our models to achieve parts of our goal in RQ3 and the approach to the transformations towards RQ1 with the verification of conceptual system designs.

### 4.3.4 Verification of E2E latency of legacy system implementation on use-case 3

In the following, we use the implementation of the ROS 2 Autoware Real-Time benchmark (ROS Realtime Working Group, 2025b) (Use-Case 3) to show the verification process on an actual ROS two implementation. In this experiment, we demonstrate formal verification on a real-world use-case to answer RQ3. Furthermore, we demonstrate the modularization of the toolchain and reusability of components compared to the application of Use-Case 1 to answer RQ2. We run the system in a development container and create a ROS2_tracing trace. The trace is transformed into the ROS 2 Data Model. In Figure 13, we show an excerpt of the system, visualized through implemented message flow analysis. For simplicity, we focus the evaluation on the chains shown in the figure. Furthermore, we assume all the callbacks to be in the same executor.

We perform verification using the E2E metamodels. We transform the system into four different versions of the verification metamodel. Each representing a different chain that can be verified from the given example. The first chain is going from the Euclidean Cluster Settings to the Intersection Output. The second chain goes from the Front Lidar Driver to the Object Collision Estimator. The third chain goes from the Front Lidar Driver to the Intersection Output via subscription fusion on Euclidean Cluster Detector. The last chain goes from the Euclidean Cluster Settings to the Object

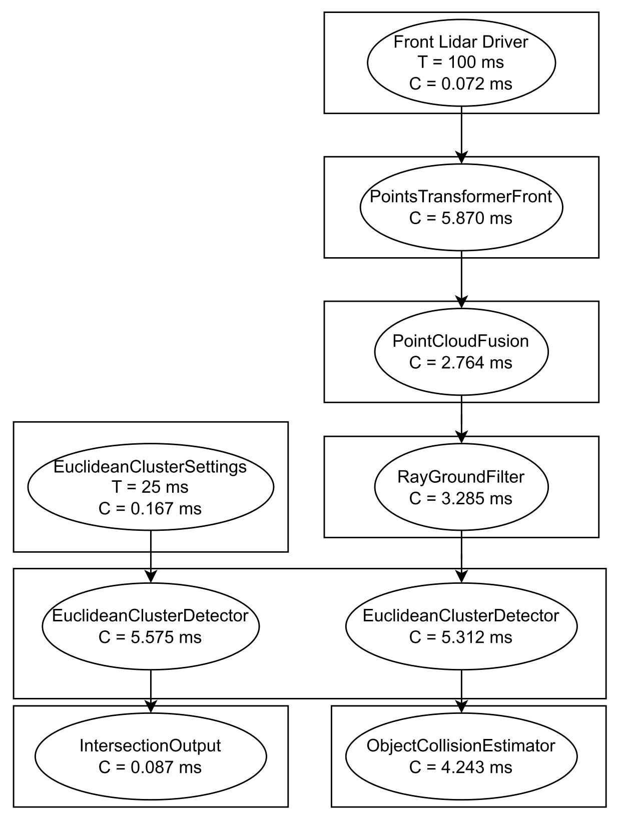


FIGURE 13
Extract from the lidar pipeline in the autoware reference system with the obtained worst case execution times C and the periods T for timers.

Collision Estimator via subscription fusion on the Euclidean Cluster Detector. Hence, following the methodology from Start A, the following results in Table 2 are obtained for the E2E latency for the individual chains.

## 5 Evaluation and discussion

In this section, we first compare the proposed methodology and implemented toolchain with a manual application of formal verification. Next, we discuss the threats to validity before answering the research questions.

### 5.1 Comparison of manual and automated verification

After implementation and application of the toolchain, in this section, we compare the verification steps following the methodology to a manual approach to provide more evidence towards automation in RQ1 and modularization for RQ2.

#### 5.1.1 Verification of legacy systems

The following steps are needed to perform formal verification of legacy systems using a manual approach:

1. System Implementation
2. System Component Determination
3. Real-Time Parameter Determination
4. Formal model System composition
5. Formal Verification

The following steps are needed to perform formal verification of legacy systems using the automated approach:

1. System Implementation
2. Systems Execution and Tracing
3. Model Parsing
4. Model Transformation
5. Verification Code Generation
6. Formal Verification

At first glance, the automated verification approach contains more steps. Nevertheless, the given steps can be performed using pre-defined transformations. In the manual approach, analysis and extraction of system components and runtime parameters have to be conducted by hand, which is error-prone and requires application and domain knowledge. Furthermore, additional domain expert knowledge in formal verification is needed to apply formal modeling and verification.

#### 5.1.2 Verification of conceptual systems

The following steps are needed to perform formal verification of conceptual systems using a manual approach:

1. Formal Model System Definition
2. Real-Time Parameter Determination
3. Formal Verification

The following steps are needed to perform formal verification of conceptual systems using the automated approach:

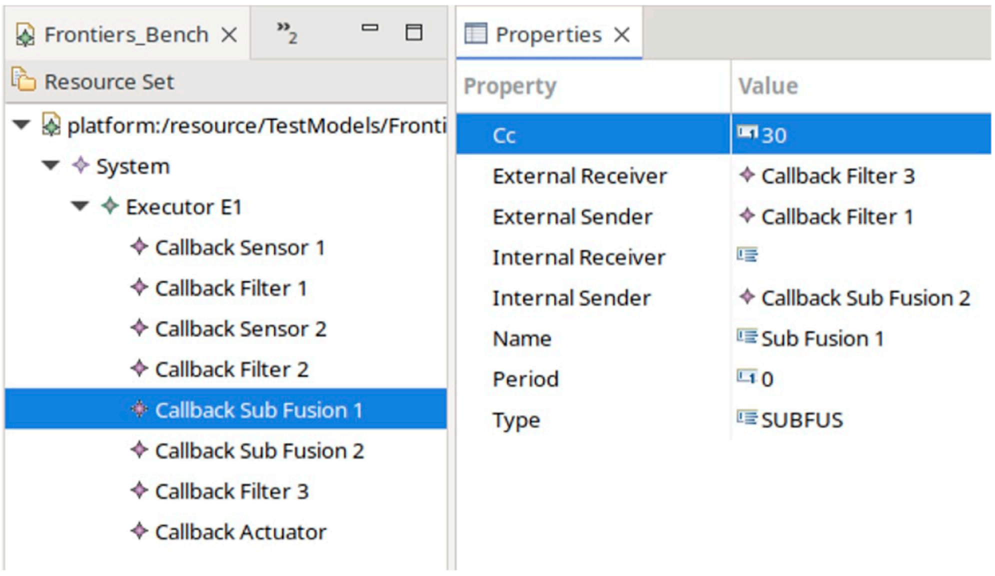
1. Architectural Modeling
2. Real-Time Parameter Determination
3. Generation of Formal Models
4. Formal Verification

In a manual approach to verifying conceptual systems, the practitioner directly works in the formal modeling environment. This reduces the complexity of the toolchain. However, domain knowledge is required to create formal models and formal verification. In the automated approach, a practitioner models the system in a modeling environment before automatically generating the formal models using pre-defined transformations.

### 5.2 Limitations and threats to validity

While the proposed methodology has strong potential for generalization due to its use of MDE techniques, the need for customization and the reliance on specific tools during the implementation present challenges.

First of all, there might be scenarios where transformations from ROS two execution traces to formal models (via EMF metamodels) may not fully capture all relevant system behaviors and parameters. Additionally, ROS2_tracing may not capture all necessary parameters (e.g., service/client interactions), which could lead to incomplete or incorrect models.



- platform:/resource/TestModels/Fronti
  - System
    - Executor E1
      - Callback Sensor 1
      - Callback Filter 1
      - Callback Sensor 2
      - Callback Filter 2
      - Callback Sub Fusion 1
      - Callback Sub Fusion 2
      - Callback Filter 3
      - Callback Actuator

| Property | Value |
|---|---|
| Cc | 30 |
| External Receiver | Callback Filter 3 |
| External Sender | Callback Filter 1 |
| Internal Receiver | |
| Internal Sender | Callback Sub Fusion 2 |
| Name | Sub Fusion 1 |
| Period | 0 |
| Type | SUBFUS |

FIGURE 14
E2E Model instance in Eclipse.

```
// Under Utilized System: Sub Fus, Sub Act
// Csum: 180ms, Cch1: 110ms, Cch2 160ms, Ts 360ms
// Ct1: 10ms Tt1: 360ms
// Ct2: 20ms Ct2: 360ms
// CHAIN 1:
/*
S1 = Sensor(180, 10, 360, 0);
F1 = Filter(180, 0, 1);
Fus1 = Filter(180, 1, 2);
F3 = Filter(180, 2, 3);
A = SubAct(180, 3);
system S1, F1, Fus1, F3, A;
*/
// CHAIN 2:
/*
S2 = Sensor(180, 20, 360, 0);
F2 = Filter(180, 0, 1);
Fus2Cb = Filter(180, 1, 2);
S1 = SubFus(180, 10, 360, 2, 3);
F1 = Filter(180, 3, 4);
Fus1 = Filter(180, 4, 5);
F3 = Filter(180, 5, 6);
A = SubAct(180, 6);
system S2, F2, S1, Fus2Cb, F1, Fus1, F3, A;
*/
```

**Listing 2. Created UPPAAL model system declaration.**

Next, the automated classification of callbacks and system components (e.g., as filters, sensors, actuators) based on trace data may lead to misclassifications, especially in complex chains.

In the implementation of the methodology, there is a toolchain dependency, where the correctness of the verification heavily depends on the accurate functioning of multiple tools (ROS2_tracing, Eclipse EMF, QVT-O, Acceleo, UPPAAL). Bugs or misconfigurations in any of these could compromise results.

In this paper, a proof of concept is demonstrated on a conceptual system and a specific benchmark (Autoware). The scaling of the approach to large, heterogeneous, or multi-executor ROS two systems has to be evaluated further. As an example, the evaluation in this paper assumes all callbacks are in the same executor, which may not reflect real-world deployments with multiple executors or distributed systems.

Additionally, the toolchain and formal models rely on specific versions or configurations of ROS two and the tracing tools used, limiting applicability across different setups. Nevertheless, different templates of formal models can be introduced to model different versions of ROS two systems that can be matched through manually set parameters by the developer.

While the toolchain is shown to work in a controlled setting, there is a need for extended statistical or empirical analysis to support claims of improved efficiency, accuracy, or usability. Furthermore, user studies or usability evaluations are needed to evaluate the simplifications for robotics developers when applying formal methods. Furthermore, more evaluation is needed on how verification results are interpreted or used to guide system design decisions.

## 5.3 Answers to the research questions

After utilizing the methodology to implement and apply formal verification, in this section, we answer the posed research goals.

TABLE 1 Experiment Results with the formal models (all results are in ms).

| | | Csum | Cch | Ts | Tf | Ta | Teper et al. | Formal models |
|---|---|---|---|---|---|---|---|---|
| sub fus + sub act | Chain 1 | 180 | 110 | 360 | — | — | 1,430 | 1,430 |
| | Chain 2 | 180 | 160 | 360 | — | — | 2,490 | 2,490 |
| sub fus + tim act | Chain 1 | 210 | 140 | 420 | — | 840 | 2,900 | 2,900 |
| | Chain 2 | 210 | 190 | 420 | — | 840 | 4,140 | 4,140 |
| tim fus + sub act | Chain 1 | 210 | 140 | 420 | 840 | — | 2,900 | 2,900 |
| | Chain 2 | 210 | 160 | 420 | 840 | — | 2,890 | 2,890 |
| tim fus + tim act | Chain 1 | 240 | 170 | 480 | 960 | 960 | 4,730 | 4,730 |
| | Chain 2 | 240 | 190 | 480 | 960 | 960 | 4,720 | 4,720 |

TABLE 2 Obtained Upper bounds for the End-To-End Latency.

| Chain | E2E Latency |
|---|---|
| C1 | 134.333 m |
| C2 | 291.553 m |
| C3 | 398.511 m |
| C4 | 398.511 m |

**RQ1:** What approach can be employed to automate the application of formal verification of ROS 2-based applications?

Answering this question, we first propose a methodology that incorporates four layers. The first layer is the implementation of ROS two application. In case an implementation is given, the system can be run, and runtime information can be recorded during execution in an execution trace. Such a generated trace can be utilized to parse the given system and run-time parameters into a model in a chosen modeling environment. In the modeling environment, a second metamodel allows the focus on needed parameters and components for formal verification. To allow automation in the modeling process, model-to-model transformations can be utilized to automatically create a model instance of the second model based on an instance of the first model. The second model can then be automatically transformed to the formal model representation utilizing formal model templates and a model-to-text transformation.

Generally, while the traces could be parsed into the second model representation or even the formal model directly, the introduction of the second layer of modeling allows extendability and decouples the process of tracing from the modeling and verification.

Following the methodology, verification can not only be automated with a given ROS two system implementation, but the model-to-text transformation can be used to generate a formal model from the EMF model automatically. This allows users to start with conceptual systems design in the modeling environment before implementing a system.

To demonstrate and assess the automation, we implement a toolchain with the model parsing, the model-to-model, and a model-to-text transformation for verification of callback latency.

We use the tool ROS2_tracing to generate system traces, which are converted into a ROS two data model using trace_analysis. We extend trace_analysis by a function that allows automated parsing of the data model to an EMF instance of the same data model in Eclipse.

We implement model-to-model transformation utilizing QVT-O and test the generation of the second EMF model representation. Next, we implement and test the model-to-text transformation using Acceleo, where we generate runnable UPPAAL code that is used for formal verification.

Hence, we show that utilizing model-driven engineering techniques with model parsing, model-to-model, and model-to-text transformation can automate the process of determining the parameters and, secondly, the instantiation of formal models.

Furthermore, in the same toolchain, we implement models to verify end-to-end latency in ROS two processing chains. In this implementation, the model parsing and the EMF data model can be reused. We implement the second model and the UPPAAL templates. The model-to-model and model-to-text transformations have not been implemented and have only been conducted by hand. Nevertheless, we demonstrate the automation of such transformations on the first verification example. Hence, in future work, the transformations can be automated as well.

**RQ2:** How can the formal verification process be modularized to enable domain experts to concentrate on their specific areas of expertise without requiring deep formal methods knowledge?

In this methodology, we apply model-driven engineering to decouple the process of formal modeling from systems tracing and automate significant steps throughout the process. We implement two different metamodels for each formal verification approach. The first metamodel allows the import of parameters obtained by tracing and does not need to be adapted until the tracing approach changes. The second metamodel focuses on the parameters needed in a specific verification approach. Hence, such a metamodel is added or changed when formal models are adapted or added. Two consecutive models are connected by the fact that the first model is an extension of the second model. The models are substitutable by any other model that obeys the extension mechanism. However, the transformation needs to be adapted, provided that the attributes of the substituted class change. This acts as an informal proof of modularization. In the implementation of the toolchain, we demonstrate such ability of replacement by implementing verification of ROS two systems using two different UPPAAL models. Each of the UPPAAL models has its own implementation of MM2, but stems from the same implementation of MM1.

Following the proposed methodology, we identify three main expert domains that are needed in the creation and maintenance of a toolchain.

1. Robotics Expert: Following the given methodology, the robotics expert is responsible for the tracing of systems and runtime parameters, such as the application of the final toolchain.
2. Software Engineering/Modeling Expert: The software engineering/modeling expert implements the model-to-model transformations, such as the definition of the metamodels. The implementation of the parsing and the model-to-text transformations needs to be in collaboration with the robotics expert (parsing) and the formal methods expert (model-to-text).
3. Formal Methods Expert: The formal methods expert is responsible for the creation of the formal model templates that can be reused for verification. Furthermore, the expert needs to compose the formal verification queries.

As the transformations between the toolchain components can be automated, the robotics developer as a practitioner only needs to learn the execution of such transformations to apply a toolchain following the proposed methodology.

**RQ3:** How can a methodology incorporate verification using different formal models?

With the last research question, we focus on the modularity and extensibility of the toolchain. The proposed methodology incorporates multiple steps in a modeling environment. Firstly, this allows the decoupling of domain expertise needed to design such a component, but it also allows for the extendability of the toolchain. The model parsing and implementation of the ROS two data model, such as the EMF data model, is reusable for different verification approaches. As long as the parameters are contained in the trace, multiple formal model representations can be built upon such parameters. To introduce a different formal model representation to the toolchain, we add a different EMF model in the second part of the toolchain. This EMF model represents the parameters and components needed for the second formal model representation. To automate the verification process, new model-to-model and model-to-text transformations have to be created, incorporating the new EMF model representation and the UPPAAL templates. When applying the methodology to multiple formal model representations in the same toolchain, such a toolchain consists of one tracing and parsing and data model implementation that can be reused for all formal model representations, as long as all needed parameters are contained in the trace. Next, there will be their formal models and individual EMF metamodels for each of the individual approaches with their specific model-to-model and model-to-text transformations. Hence, when extending the toolchain with a new formal model, the methodology can be applied to extend an existing toolchain with the needed components, while reusing the model parsing and the EMF data model.

## 6 Related work

The analysis of ROS two execution behavior has been a subject of interest in recent research. Casini et al. (2019) and Blaß et al. (2021) conduct response time analysis, which is crucial for the formal verification of ROS two timing behavior. Their work has laid the foundation for creating formal model templates and modeling timing requirements.

Halder et al. (2017) propose formal verification of ROS two communication between nodes using UPPAAL. Their approach models low-level parameters such as queue sizes and timeouts to verify queue overflow. While their focus is on modeling and verification, our toolchain emphasizes the automation of verification processes.

Carvalho et al. (2020) employ an Alloy extension called Electrum to implement a model-checking technique that automatically creates models from configurations extracted in continuous integration and specifications. Their approach targets high-level architectural verification, whereas our toolchain aims to verify low-level behavior such as system execution.

Webster et al. (2016) and Liu et al. (2018) work on formal verification of requirements for robotic systems and ROS two message passing in DDS using different model checkers. Although their approaches are manual and use different model checkers than UPPAAL, our focus is on automating the verification process. Extending our toolchain to support other model checkers could be a valuable future direction.

Kim and Kim (2022) introduce the Robo Fuzz Framework, which is used for fuzz testing robotic systems to find bugs in system implementations. Their framework focuses on data type mutation and violation of physical laws and hardware specifications. In contrast, our framework focuses on timing and execution verification of ROS two applications. Additionally, fuzz testing is not exhaustive.

Anand and Knepper (2015) present ROSCoq, a “correct-by-construction” approach for developing certified ROS two systems. While their approach is not applicable to legacy systems, it complements the verification conducted in our work.

Beckmann and Thoss (2010) explore model-based development of DDS-based systems such as ROS 2. The work highlights how the DDS architecture supports model-based development, whereas our focus is on verification.

Parra et al. (2021) develop Ecore models to specify QoS requirements for ROS 2. This work is complementary to ours, as we focus on architectural components and verification related to task scheduling.

Dal Zilio et al. (2023) propose a toolchain for runtime and offline verification of general robotic systems beyond ROS. While the authors focus on general robotic systems and application code with internal logic, our toolchain targets timing issues induced by using ROS 2 as middleware. Additionally, our toolchain leverages model-driven engineering in the Eclipse environment to support iterative verification and model-based development.

Teper et al. (2022) provide an end-to-end timing analysis for ROS two systems, focusing on cause-effect chains and their timing behavior. The work is significant for understanding the maximum reaction time and maximum data age in ROS two systems, which are critical for ensuring real-time performance.

Bédard et al. (2023) utilize ROS2_tracing to allow message flow analysis of ROS two systems. The work can be used as a ground for allowing additional analysis in the tracing layer of our toolchain, and is used as a foundation for the structural analysis of ROS two systems in our evaluation.

In our previous work (Dust et al., 2023a), we develop reusable UPPAAL templates to verify timing behavior and buffer overflow in ROS two systems. Building on this foundation, our current work aims to further simplify the formal verification process by automating parameter determination and model initialization, making formal verification more accessible and less error-prone for robotics developers.

## 7 Conclusion and future work

In this article, we introduce a novel approach to automating model-based verification for ROS 2-based applications using model-driven engineering techniques. This work extends our paper (Dust et al., 2024) and builds on our previous work (Dust et al., 2023b), which identified potential timing issues, and utilizing the formal model templates proposed in (Dust et al., 2023a). In this article, we develop a methodology that leverages ROS two system traces to automate the verification process. Our toolchain uses ROS two execution traces to initialize pre-defined formal model templates through models and model transformations.

The toolchain supports the verification of both implemented and conceptual systems by providing four different model representations, enhancing traceability throughout the process. Additionally, it allows for parameter refinement and iterative verification of system parameters without repeated source code adaptation.

A key feature of our approach is its flexibility in supporting different types of formal modeling analyses. We demonstrated this by comparing two formal modeling approaches: one at the individual node level and one at the system level (end-to-end analysis). The individual node level analysis focuses on verifying the timing behavior of specific nodes, while the end-to-end analysis examines the timing behavior of cause-effect chains across the entire system. This comparison showcases the toolchain's versatility in accommodating various verification needs.

Our evaluation demonstrates the feasibility of using ROS2_tracing to capture the necessary trace points for verification. However, customization may be required to include all needed parameters. Further evaluation is needed to determine the extent to which execution times from a single system execution are sufficient for verification. Nonetheless, our work shows the potential for automatic parameter determination using system traces and model-based development for formal verification.

The toolchain also opens up possibilities for automated model-based generation of ROS two application code and modeling of requirements, which are areas for future research. While these features would enhance automation, they are not essential to demonstrate the feasibility and novelty of our methodology.

Despite being demonstrated with specific tools (ROS2_tracing, Eclipse EMF, Acceleo, QVT-O, and UPPAAL), our approach can be implemented using different tools, such as other tracing tools, model editors, and verification tools. This flexibility makes our methodology adaptable to various development environments. As an example, when choosing a different formal modeling environment, only an additional metamodel with the corresponding model-to-model and model-to-text transformations is needed. In contrast, the tracing and parsing components do not need to be changed.

Given the preliminary state of the toolchain implementation, this paper serves as a first proposal and proof of feasibility, making it suitable for researchers and tool developers. More evaluation and implementation are needed to make the toolchain usable on real robotics systems. To enable more extensive verification, the proposed metamodels and transformations need to be refined and extended. Additional formal model templates should be developed and integrated into the toolchain. Future work will also involve modeling requirements and verification properties, which were not included in this implementation.

In conclusion, our work demonstrates the potential of using system traces and model-driven engineering to automate the formal verification of ROS two systems. By refining the data model and output of ROS2_tracing, and providing formal proof of correctness for the toolchain implementation, we can further enhance the robustness and usability of our approach. Future research will focus on automating verification and simulation feedback, modeling requirements, and generating ROS two application code. Additionally, investigating the ease of use of the proposed toolchain will be essential to ensure its practical applicability in real-world scenarios.

## Data availability statement

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found below: https://sites.google.com/view/mbfvros2.

## Author contributions

LD: Writing – original draft, Writing – review and editing. RG: Writing – review and editing. SM: Writing – review and editing. ME: Writing – review and editing. CS: Writing – review and editing.

## Funding

The author(s) declare that financial support was received for the research and/or publication of this article. We acknowledge the support of the Swedish Knowledge Foundation via the synergy project ACICS–Assured Cloud Platforms for Industrial Cyber-Physical Systems, grant nr. 20190038, the Hög project SEINE–Automatic Self-configuration of Industrial Networks, grant nr. 20220230, the research profile DPAC - Dependable Platform for Autonomous Systems and Control project, grant number: 20150022, and the Swedish Governmental Agency for Innovation Systems (VINNOVA) via the PROVIDENT, INTERCONNECT, and FLEXATION projects.

## Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

## Generative AI statement

The author(s) declare that Generative AI was used in the creation of this manuscript. Generative AI was used to correct the text from spelling and grammar mistakes, such as rephrasing specific sentences.

## Publisher's note

## References

Alur, R., and Dill, D. L. (1994). A theory of timed automata. *Theor. Comput. Sci.* 126, 183–235. doi:10.1016/0304-3975(94)90010-8

Anand, A., and Knepper, R. (2015). "Roscoq: robots powered by constructive reals," in *Interactive theorem proving*. Editors C. Urban, and X. Zhang (Cham: Springer International Publishing), 34–50.

Autoware Foundation (2025). Autoware: open-source software stack for autonomous driving. Available online at: https://github.com/autowarefoundation/autoware (Accessed May 26, 2025).

Beckmann, K., and Thoss, M. (2010). "A model-driven software development approach using omg dds for wireless sensor networks," in *IFIP international workshop on software technolgies for embedded and ubiquitous systems* (Springer), 95–106.

Bédard, C., Lajoie, P.-Y., Beltrame, G., and Dagenais, M. (2023). Message flow analysis with complex causal links for distributed ros 2 systems. *Robot. Aut. Syst.* 161, 104361. doi:10.1016/j.robot.2022.104361

Bédard, C., Lütkebohle, I., and Dagenais, M. (2022). ros2_tracing: multipurpose low-overhead framework for real-time tracing of ros 2. *IEEE Robot. Autom. Lett.* 7, 6511–6518. doi:10.1109/lra.2022.3174346

Birman, K., and Joseph, T. (1987). "Exploiting virtual synchrony in distributed systems," in *Proceedings of the eleventh ACM Symposium on Operating systems principles*. doi:10.1145/41457.37515

Blaß, T., Casini, D., Bozhko, S., and Brandenburg, B. B. (2021). "A ros 2 response-time analysis exploiting starvation freedom and execution-time variance," in *IEEE real-time systems symposium RTSS* (IEEE), 41–53.

Carvalho, R., Cunha, A., Macedo, N., and Santos, A. (2020). *Verification of system-wide safety properties of ros applications*. IEEE/RSJ International Conference on Intelligent Robots and Systems IROS.

Casini, D., Blaß, T., Lütkebohle, I., and Brandenburg, B. (2019). "Response-time analysis of ros 2 processing chains under reservation-based scheduling," in *31st euromicro conference on real-time systems*, 1–23.

Dal Zilio, S., Hladik, P.-E., Ingrand, F., and Mallet, A. (2023). A formal toolchain for offline and run-time verification of robotic systems. *Robot. Aut. Syst.* 159, 104301. doi:10.1016/j.robot.2022.104301

Dust, L., Ekström, M., Gu, R., Mubeen, S., and Seceleanu, C. (2024). "A model-based methodology for automated verification of ros 2 systems," in *Proceedings of the 2024 ACM/IEEE 6th international workshop on robotics software engineering*, 35–42.

Dust, L., Gu, R., Seceleanu, C., Ekström, M., and Mubeen, S. (2023a). "Pattern-based verification of ros 2 nodes using uppaal," in *International conference on formal methods for industrial critical systems* (Springer), 57–75.

Dust, L., Gu, R., Seceleanu, C., Ekström, M., and Mubeen, S. (2025). *A model-based approach to automation of formal verification of ROS 2-based systems — sites.google.com* Sweden. Available online at: https://sites.google.com/view/mbfvros2 (Accessed March 08, 2025).

Dust, L., Persson, E., Ekström, M., Mubeen, S., Seceleanu, C., and Gu, R. (2023b). "Experimental evaluation of callback behavior in ros 2 executors," in *28th international conf. On emerging technologies and factory automation*.

Eclipse Foundation (2025a). Acceleo: code generator for eclipse. Available online at: https://eclipse.dev/acceleo/ (Accessed May 26, 2025).

Eclipse Foundation (2025b). Qvt operational mappings (qvto). Available online at: https://wiki.eclipse.org/QVTo/ (Accessed May 26, 2025).

Group, O. M. (2022). Object management group. Available online at: https://www.omg.org/ (Accessed November 7, 2022).

Gutiéerez, C. S. V., Juan, L. U. S., Ugarte, I. Z., and Vilches, V. M. (2018). Towards a distributed and real-time framework for robots: Evaluation of ROS 2.0 communications for real-time robotic applications. doi:10.48550/arXiv.1809.02595

Halder, R., Proença, J., Macedo, N., and Santos, A. (2017). "Formal verification of ros-based robotic applications using timed-automata," in *2017 IEEE/ACM 5th international FME workshop on formal methods in software engineering (FormaliSE)*, 44–50.

Hendriks, M., Yi, W., Petterson, P., Hakansson, J., Larsen, K., David, A., et al. (2006). "Uppaal 4.0," in *Third international conference on the quantitative evaluation of systems - (QEST'06)*.

Kim, S., and Kim, T. (2022). "Robofuzz: fuzzing robotic systems over robot operating system (ros) for finding correctness bugs," in *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*, 447–458.

Li, Z., Hasegawa, A., and Azumi, T. (2022). Autoware_perf: a tracing and performance analysis framework for ros 2 applications. *J. Syst. Archit.* 123, 102341. doi:10.1016/j.sysarc.2021.102341

Liu, Y., Guan, Y., Li, X., Wang, R., and Zhang, J. (2018). "Formal analysis and verification of dds in ros2," in *2018 16th ACM/IEEE international conference on formal methods and models for system design (MEMOCODE)* (IEEE).

OpenRobotics (2023a). Ros 2: documentation. Available online at: https://docs.ros.org/en/humble (Accessed May 22, 2023).

OpenRobotics (2023b). Ros: distributions. Available online at: http://wiki.ros.org/Distributions (Accessed July 28, 2023).

Parra, S., Schneider, S., and Hochgeschwender, N. (2021). "Specifying qos requirements and capabilities for component-based robot software," in *2021 IEEE/ACM 3rd international workshop on robotics software engineering (RoSE)*.

Rajkumar, R., Lee, I., Sha, L., and Stankovic, J. (2010). "Cyber-physical systems: the next computing revolution," in *Proceedings of the 47th design automation conference*, 731–736.

ROS Realtime Working Group (2025a). Ros 2 realtime reference system. Available online at: https://github.com/ros-realtime/reference-system (Accessed May 26, 2025).

ROS Realtime Working Group (2025b). Autoware reference system. Available online at: https://github.com/ros (Accessed May 26, 2025).

Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: eclipse modeling framework*. 2nd edn. Addison-Wesley Professional.

Teper, H., Günzel, M., Ueter, N., von der Brüggen, G., and Chen, J.-J. (2022). "End-to-end timing analysis in ros2," in *2022 IEEE real-time systems symposium (RTSS)*, 53–65. doi:10.1109/RTSS55097.2022.00015

Webster, M., Dixon, C., Fisher, M., Salem, M., Saunders, J., Koay, K. L., et al. (2016). Toward reliable autonomous robotic assistants through formal verification: a case study. *IEEE Trans. Human-Machine Syst.* 46, 186–196. doi:10.1109/thms.2015.2425139