frontiers | Frontiers in Computer Science

TYPE Original Research
PUBLISHED 31 October 2025
DOI 10.3389/fcomp.2025.1659785

OPEN ACCESS

EDITED BY
Novarun Deb,
University of Calgary, Canada

REVIEWED BY
Wei Li,
City University of Hong Kong, Hong Kong SAR, China
Nafees Akhter Farooqui,
Integral University, India

*CORRESPONDENCE
Paola Giannini
paola.giannini@uniupo.it

RECEIVED 04 July 2025
ACCEPTED 23 September 2025
PUBLISHED 31 October 2025

CITATION
Bergenti F, Egidi L, Galliera L, Giannini P and Monica S (2025) Correct implementation of agent interaction protocols.
Front. Comput. Sci. 7:1659785.
doi: 10.3389/fcomp.2025.1659785

Correct implementation of agent interaction protocols

Federico Bergenti[1], Lavinia Egidi[2], Leonardo Galliera[2], Paola Giannini[3]* and Stefania Monica[4]

[1]Dipartimento di Ingegneria dei Sistemi e delle Tecnologie Industriali, Università di Parma, Parma, Italy, [2]Dipartimento di Scienze e Innovazione Tecnologica, Università del Piemonte Orientale, Alessandria, Italy, [3]Dipartimento per lo Sviluppo Sostenibile e la Transizione Ecologica, Università del Piemonte Orientale, Vercelli, Italy, [4]Dipartimento di Scienze e Metodi dell'Ingegneria, Università di Modena e Reggio Emilia, Reggio Emilia, Italy

Coordinating agents that communicate through asynchronous message exchanges to execute interaction protocols presents a complex and pressing challenge. In this article, we address this issue by introducing Multiparty Session Types (MPST) for the formal specification of agent interaction protocols, from which we derive implementations of the corresponding agent systems. Correctness is ensured on one side by the MPST methodology, which derives the local protocols of participants from a global specification by projection, and on the other by translating local types into agents, providing a proof that these agents behave as prescribed by the local protocols of participants. Our agent language is Jadescript, an agent programming language that targets the widely used JADE agent platform. In addition to the theoretical framework, we describe a prototype implementation of the related tools.

KEYWORDS

multiparty sessions, global types, agent interaction protocols, agent programming languages, soundness of translation

1 Introduction

1.1 Agent programming languages

The interest in agent programming languages, as discussed in Bădică et al. (2011), dates back to the early proposals of agent technologies, as witnessed by Shoham (1993), and since then, it has grown significantly. Agent programming languages represent an important research topic because they are recognized, e.g., by Bordini et al. (2006), as important tools to support *Agent-Oriented Software Engineering* (*AOSE*), as witnessed by Bergenti et al. (2004). Agent programming languages are based on specific agent models and provide specific language constructs to work with these models at a high level of abstraction. The diverse agent programming languages share some characteristics, such as simplicity and ease of use, to simplify the work of programmers and increase the quality of produced agents. Despite these similarities, the specific features that these languages provide differ significantly, concerning, e.g., the selected agent mental attitudes (if any), the integration with an agent platform (if any), and the reference programming paradigm (e.g., declarative or imperative).

The literature already provides some classifications of relevant agent programming languages in terms of structured surveys of the current state of the art. For example, Bădică et al. (2011) classify agent programming languages according to their adoption of agent mental attitudes. According to Bădică et al. (2011), agent programming languages can be classified into: *Agent-Oriented Programming* (*AOP*) languages, as studied by Shoham (1991), *Belief-Desire-Intentions* (*BDI*) languages, as studied by Bordini et al. (2007); hybrid

languages, which combine the two previous approaches; and other languages, which fall outside of the previous classes. Note that this classification recognizes that BDI languages follow the AOP paradigm, but it treats them specifically for their notable relevance in the literature. Bordini et al. (2006) propose a different classification, in which agent programming languages are grouped into three classes: declarative, imperative, and hybrid. Most of the agent programming languages discussed in Bordini et al. (2006) are declarative because declarative languages natively focus on automated reasoning, which is an important tool for developing effective agents. However, Bordini et al. (2006) also discusses some relevant imperative languages. Some of them have been defined by adding language constructs to host languages. The presence (or absence) of a host language is an important characteristic of most imperative languages.

Following the mentioned classifications, the language discussed in Bergenti et al. (2018, 2020), namely Jadescript, is an imperative AOP language designed from scratch around the abstractions that the *Java Agent DEvelopment framework* (*JADE*) promoted to support the construction of effective agents, as discussed in Bergenti et al. (2020). Jadescript shares relevant similarities to popular scripting languages, e.g., Python and JavaScript, to provide a clear and simple way to implement agents and related abstractions, e.g., (communication) ontologies, as discussed in Tomaiuolo et al. (2006), and behaviors, as discussed in Bergenti and Petrosino (2018). The source codes of Jadescript agents bear relevant similarities with the pseudocodes used to describe agents in research articles, e.g., Shoham and Leyton-Brown (2008); Yokoo (2001). Jadescript is meant to help programmers adopt the best development practices to enhance the overall quality of produced software. For example, agents should not busy wait for events, and Jadescript natively provides cyclic behaviors, as described in Petrosino and Bergenti (2018), and related data types, discussed in Petrosino et al. (2022), to allow agents to suspend when no events can be processed, as exemplified in Bergenti et al. (2018). The Jadescript type system, which is briefly summarized in Petrosino et al. (2022), provides very high-level abstractions for effective agent programming and promotes the construction of robust and maintainable code. Jadescript is designed to be integrated with mainstream development tools, and therefore it comes with a comprehensive set of programmer-friendly development tools. In particular, the dedicated Jadescript plugin for Eclipse discussed in Petrosino et al. (2021) is available (github.com/aiagents/jadescript) as the official tool to use Jadescript.

1.2 Agent interaction protocols

The near-future development plans for Jadescript include dedicated support for (agent) interaction protocols, as discussed by Poslad (2007). In particular, the plans target a subset of the interaction protocols standardized by the *Foundation for Intelligent Physical Agents* (*FIPA*, https://www.fipa.org), which is an IEEE Standards Committee established to promote interoperability among agents. FIPA specifies some general-purpose interaction protocols, and FIPA-compliant agents are requested to support at least some of them. FIPA encourages designers and programmers to adopt these interaction protocols, which motivates the need for dedicated support for them in Jadescript.

An interaction protocol is a way to govern the message-based communication among two or more agents. Interaction protocols are described in terms of reference scenarios in which agents can play, at least, two roles: *initiator* and *participant*. Interaction protocols mandate the types of messages exchanged among the agents playing the various roles, and they normally assume the presence of one initiator and more than one participant. The initiator sends the first message to start the enactment of the interaction protocol. A subset of the participants receives this first message, and each of them independently decides what to do, which normally involves sending other messages to other agents. The specification of an interaction protocol states the correctness constraints for the messages involved in the reference scenario. In FIPA specifications, messages are labeled via their performatives, and these correctness constraints are expressed in terms of their performatives, as discussed in Bergenti and Ricci (2002).

1.3 Multiparty session types

A *multiparty session* (MPS for short) is an interaction among *participants* communicating by exchanging messages, (Honda et al., 2008, 2016). The interaction is specified by a *global type* of the session. *Local* or *session* types may be retrieved as *projections* from the global type. Session types give a decoupled (i.e., distributed) view of a protocol from the perspective of each participant. Typical safety properties ensured by session types are *communication safety* (absence of communication errors), *session fidelity* (agreement with the protocol) and *deadlock-freedom*, as discussed in Honda et al. (2016). When dealing with more than two participants, the type system also guarantees the liveness property known as *progress*, which entails *livelock freedom* and *orphan message freedom*, Hüttel et al. (2016). Livelock freedom means that a participant waiting for a message will eventually get it, while orphan message freedom means that a message sent by a participant will eventually be read by the one it was sent to. These properties are difficult to prove in a distributed environment.

The global types of Honda et al. (2016) have several limitations that make them suitable for specifying the ordinary interaction protocols among agents. In particular, a session involves a fixed set of participants, whose behavior is individually specified when the session is first initiated: there is no notion of specifying the *behavior for a class of participants* that share the same behavior, and no participant can *dynamically* (i.e., during an ongoing session) *leave* the interactions. These are, however, basic requests for the FIPA interaction protocols.

A very expressive enhancement of global types was proposed in Deniélou and Yoshida (2011); Deniélou et al. (2012). Roles are defined as classes of local behaviors that an arbitrary number of participants can dynamically join and leave. A locking mechanism is introduced to enforce communication safety and progress. This extension is very expressive; however, it is unrealistically implementable with the communication pattern of agent languages, since it requires some sort of centralized register handling the association between participants and roles. Other extensions

tailored to specific application domains were proposed. Most of these extensions target specific programming languages (Castro-Perez et al., 2019; Cledou et al., 2022; Lagaillardie et al., 2022; Dagnino et al., 2023). Of the language-independent ones, we mention the two that are closer to our proposal. The first is Pabble (Ng and Yoshida, 2014, 2015), an extension of the language Scribble (Honda et al., 2011), for describing the multiparty session types in a Java-like syntax. In Pabble, multiple participants can be grouped in the same role and indexed, and there is the possibility of changing the participants in a role by *parameterisation*. The examples reported in Ng and Yoshida (2014, 2015) show how different interconnection typologies of parallel processes can be modeled. The second extension was proposed to ensure good properties of the interactions in MPSs in spite of failures. Viering et al. (2021) introduces the notions of *failure aware sub-sessions* and *role set*, which are similar to the roles of Deniélou and Yoshida (2011); Deniélou et al. (2012). A global type is specified by a number of sub-protocols that may be spawned in parallel. Sub-protocols may have role sets with different participants, and starting a sub-protocol requires that all the participants in its role sets are at the same point of the execution of their protocol.

1.4 Types for agent interaction protocols

Our proposal, inspired by the two mentioned extensions, is tailored to the goal of specifying FIPA interaction protocols, in which groups of agents are often addressed by an agent mediating their interactions with the rest of the world. In the following, participant is a synonym for agent. We call these groups of participants *role sets*, and their *coordinator* is the only participant interacting with them. The coordinator can broadcast a message to all the participants of the role set, and there is a construct to execute a sub-protocol on all the participants in a role set. In addition to projectability, we give some well-formedness restrictions on the global types that are meant to enforce their realisability by Jadescript agents. The aim of our work is to generate from local types the skeleton of the communications of the Jadescript agents implementing the interaction protocol, so that we can prove that the system has the progress property by construction. The translation is non-trivial due to the different models of execution of multiparty sessions (channel-based) and agent systems (event-based). Even though we are not there yet, in the article, we show the road we intend to pursue through an example. The discussed example is a hand-made translation from the local types projections of a global type describing the FIPA brokering interaction protocol to Jadescript agents.

1.5 Contributions

We make the following contributions:

- We generalize session types to capture protocols in which roles may comprise any number of participants interacting with a coordinator, while also permitting participants within a role to leave the interaction. We define the projection of global types onto local types and a reduction semantics for the latter.
- We provide a novel reduction semantics for the Jadescript agent programming language.
- We define a translation from session types to Jadescript agents and prove that this translation satisfies the session fidelity property, that is, the system exchanges messages according to the local protocols. This also implies that every message that is sent can be eventually read and that any agent awaiting a message will eventually receive it.
- We prototype a toolchain that automatically generates agent implementations from session types.

1.6 Outline of the paper

In Section 2, we first introduce our global and local session types in Sections 2.1 and 2.2. Then, the projection of global types onto local types and well-formedness are defined in Section 2.3. A semantics for session types, outlining the sequence of message exchanges among participants, is presented in Section 2.4. In Section 2.5, we analyse some of the limitations of the introduced types and motivate their rationale. Section 3 introduces the core of the Jadescriptagent language, Featherweight Jadescript (FJS for short), with its operational semantics. The translation from local types to FJS along with its correctness, i.e., the statement of the session fidelity result, is given in Section 4. A description of its implementation as an Xtext plugin for Eclipse can be found in Section 5. Finally, Section 6 describes recent applications of MPST in the area of actor and agent languages. The Supplementary material contains: the formal definition of projection (Section 1), the proof of Theorem 4.2 with the definitions and lemmas it relies on (Section 2) and the English auction interaction protocol with its sessions types (Section 3.1), the behaviors generated by the translation (Section 3.2) and a trace of an execution of the generated agent system (Section 3.3).

2 Global and local types with role sets

The typical FIPA specification of an interaction protocol is shown in Figure 1, where `sub-protocol` refers to agents that may execute the request of the initiator. The `broker` is the agent receiving the request from the `initiator` and trying to get it done by some `agent`. Of the m `agents` that the `broker` contacts, only a subset may be willing to fulfill the request. The UML diagrams, even though more rigorous than a textual description, are not a formal specification that can be used to prove properties of a given implementation.

2.1 Global types

We define an extension of the formalism of global types of Yoshida and Gheri (2020) in which we can define *role sets*, which are sets of participants executing the same sub-protocols. *Role sets*

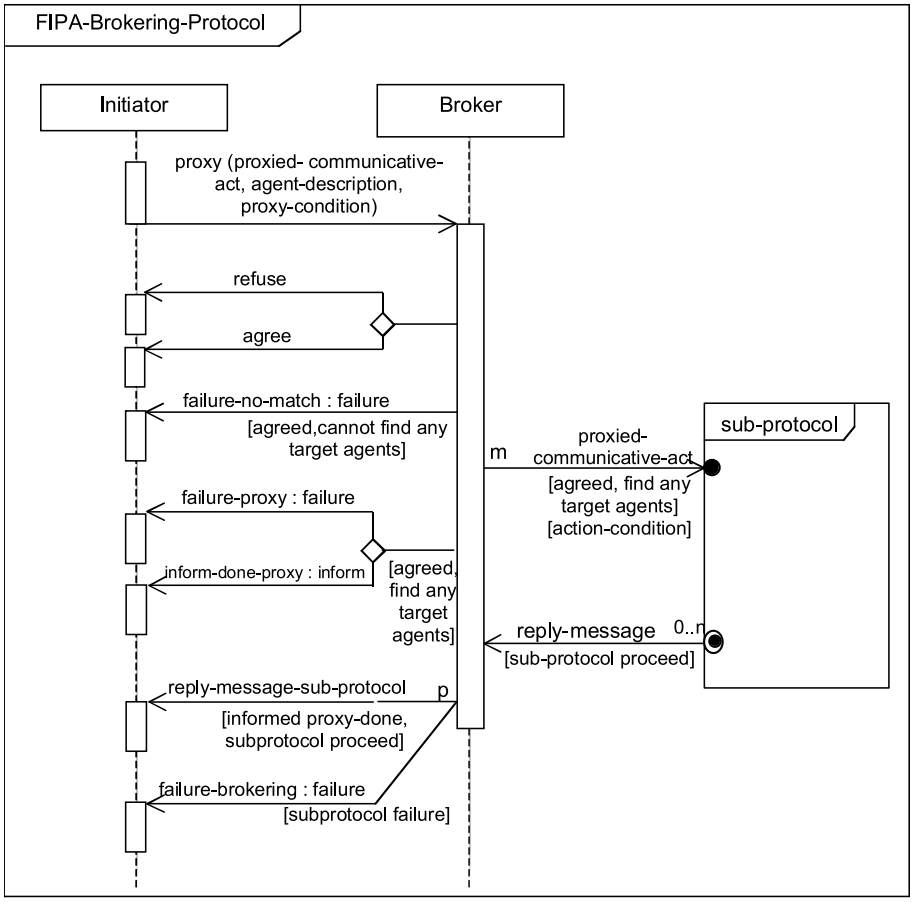

FIGURE 1
FIPA specification of the brokering interaction protocol.

specify their *coordinator*, which should be one of the participants in the MPS. In the rest of the section, we use the following meta-variables: $\mathsf{p}, \mathsf{q}, \mathsf{r}$ for single *participants*; x, y, z for *participant variables* (in the scope of a `map`); p, q, r for either *participants or participant variables*; R for *role sets*; Z for either single *participants or role sets*; Q for either *participants or participant variables or role sets*; m for *labels of messages* and S for *basic types* (`int`, `bool`,...).

Definition 2.1 (Global types). A *global protocol* declaration specifies the *participants* and the *role sets* involved in the protocol and the associated *global type*. Each *role set* is coupled with a participant (from the declared ones) which is its *coordinator*.

$$\texttt{global protocol}\ \mathit{name}(\overline{\mathsf{p}};\ \overline{\langle \mathsf{R}, \mathsf{p}\rangle}) = \mathsf{G}$$

where G is defined by:

$\mathsf{G} ::= p \to Q\ \{m_i\langle \overline{S}^i\rangle.\mathsf{G}_i\}_{i\in I}$		Choice of communications
$\mid \mu X.\mathsf{G} \mid X$		Recursion and Variable
$\mid \mathsf{End}$		End of protocol
$\mid \texttt{map}\ x : \langle \mathsf{R}, \mathsf{q}\rangle\ \mathsf{G}_1; \mathsf{G}_2$		Sub-protocol for participants in a role set
$\mid \langle x, \mathsf{q}, \mathsf{Quit}\rangle$		Exit from role set

where I is a finite non-empty index set and $m_h \neq m_k$ for $h \neq k$.

When $|I| = 1$, we drop the brackets and write $p \to Q\, m\langle \overline{S}\rangle.\mathsf{G}$ for short.

In the syntax, we highlighted the non-standard constructs. The first construct specifies a *choice of communications*, i.e., one of the messages with a label in $\{m_i\}_{i\in I}$ is sent from a participant to another. In addition to single participants, the receiver may specify all the participants in a role. In this case, we enforce the restriction that the sender be the coordinator of the role set. After one of the communications, the protocol continues as prescribed by the global type corresponding to the selected label. Recursion introduces a recursion variable X that can be used in its body to return to the beginning of G. As usual, we assume recursion to be guarded. End stands for the end of the protocol, but also the end of a sub-protocol, as we will see shortly. The `map` construct prescribes that the same protocol G_1 be executed by all the participants in the role set R. In G_1, the variable x denotes any participant in R. The semicolon preceding G_2 means that the coordinator q of R must have completed the protocol G_1 on all the participants in R before continuing as specified by G_2. So End occurring in G_1 does not mean the end of the whole interaction, but just of the sub-protocol G_1. The semicolon ; can only occur after the body of a `map`. In G_1, there cannot occur free recursion variables. We also impose the restriction that `map` cannot be nested. The FIPA

interaction protocols analyzed so far can be formalized without `map` nesting. However, eventually we would like to remove this restriction. Finally, the last clause of the definition is used by a participant in a role set, denoted by x, to exit from the protocol. Here `q` is the coordinator of the role set. This means that subsequent messages sent from `q` to the participants of its role set will not be sent to this participant.

In Listing 1 we formalize, using our global types, the FIPA brokering interaction protocol. The participants of the protocol

```
1  global protocol myBrokering(role initiator,
       role broker,roleset Subagents:broker)
2  {
3    proxy(string) from initiator to broker.
4    choice at broker{
5      refuse() from broker to initiator.stop()
           from broker to Subagents. End
6    } or {
7      agree() from broker to initiator.
           findAgent(string) from broker to
           Subagents.
8      map role agent:<Subagents,broker>{
9        choice at agent {
10         notPossible() from agent to broker.
               QUIT() from agent to~broker.End
11       } or {
12         canDo() from agent to broker.End
13       }
14     } ;
15     choice at broker {
16       failNoMatch() from broker to initiator.
             stop() from broker to Subagents.End
17     } or {
18       foundMatches() from broker to initiator
             .
19       continue() from broker to Subagents.
20         map role agent:<Subagents,broker>{
21           rec COLLDATI: {
22             sendMore() from broker to agent.
23             choice at agent {
24               addData(string) from agent to
                     broker.loop COLLDATI
25             } or {
26               noMoreData() from agent to
                     broker. End
27             } or {
28               someError() from agent to
                     broker.QUIT() from agent to
                      broker.End
29             }
30           }
31         } ;
32         choice at broker {
33           replyFromSubagents(string) from
                 broker to initiator.End
34         } or {
35           failBrokering() from broker to
                 initiator.End
36         }
37 }}}
```

Listing 1. Global type for the brokering interaction protocol.

are specified in line 1, and they are the `initiator`, the `broker` and a number of `agents` in the role set, which are `Subagents` with the broker as their coordinator. We use the Java-like syntax, coming from Scribble, Honda et al. (2011), which differs from our formal syntax mainly in the definition of the choice construct. The choice construct, e.g., lines 4, 9, 11, and 31 of Listing 1, specifies the leader of the choice, i.e., the sender of the communication. Each branch should start with a message from the leader to the same participant, and the labels of the messages in different branches must be different. This is the same requirement we had for the formal syntax. For the choice starting at line 4, the leader is the `broker`. The first message of the branch at line 5 has the label `refuse`, and the one at line 7 has the label `agree`, both sent to the `initiator`. The `choice` construct always has more than one branch; choices with just one branch are just communications, e.g., line 3 or the second message both in lines 5 and 7. For the syntax of the recursion construct, lines 21–30, the occurrence of the recursion variable at line 24 is preceded by the key word `loop`.

The interaction starts with the `initiator` sending a message labeled `proxy` to the `broker`. We specified a simple `string` as the payload, but more complex data structures may be exchanged. After this, there is a choice made by the `broker` that may decide to fulfill the request or to refuse it. In line 5, after sending the message labeled `refuse` to the `initiator`, the `broker` sends a message also to the participants in the role set `Subagents`, communicating that there will not be any subsequent interaction. As we will see when describing the projection of global to local types, this message is important since the `Subagents` (which were not involved in the initial communication of the choice) have to know whether they will be engaged in the protocol or not. In the branch starting at line 7 and ending at line 35, after the message sent from the `broker` to the `initiator`, the broker sends a message to all the participants in the role set `Subagents` asking whether they are available to do the task (second message in line 7). After this, we find the `map` construct, which means that every `agent` in the role set `Subagents` has to complete the sub-protocol specified at lines 9–13. That is, the `broker` waits for a message sent by the `agent` whose label can either be `notPossible`, saying that it is not available, followed by the message that communicates that it exits the protocol, or `canDo`, saying that it is willing to continue the interaction. From now on, the subsequent communications between the `broker` and the role set `Subagents` will not involve the `agents` that quit the protocol. After execution of the subprotocol, the `choice` starting at line 15 begins by informing the `initiator` whether there are agents willing to execute the task or not. In the former case, the `initiator` is first informed that some agents are willing to do the task, and then for each one of these agents, `map` at lines 19–30, there is a recursive interaction: it starts with the `broker` sending to each `agent` a message labeled `sendMore` and containing the data that need to be processed; then the `broker` waits from each `agent` either the processed data or a message that says that all data have been sent (label `noMoreData`), or an error message followed by a message informing that the `agent` leaves the role set. After all the agents have communicated either that they have transferred all data or that they quit, the protocol ends with the broker either sending the results or communicating failure to the `initiator`.

2.2 Local/session types

The local/session types are the view of a protocol from the perspective of each participant.

Definition 2.2 (Local/Session types). *Local protocol* declarations specify the *participants* and the *role sets* involved in the protocol from the point of view of a participant or role set and the associated *local/session* type.

$$\texttt{local protocol}\ \textit{name}\ \texttt{at}\ Z(\overline{\mathsf{p}};\ \overline{\langle \mathsf{R}, \mathsf{q}\rangle}) = \mathsf{T}$$

where T is defined by:

$\mathsf{T} ::=$	$Q\ !\{m_i\langle\overline{S}^i\rangle.\mathsf{T}_i\}_{i\in I}$	Choice of outputs
	$\mid p?\{m_i\langle\overline{S}^i\rangle.\mathsf{T}_i\}_{i\in I}$	Choice of inputs
	$\mid \mu X.\mathsf{T} \mid X$	Recursion and Variable
	$\mid \mathsf{End}$	End of protocol
	$\mid \texttt{map}\ x:\langle \mathsf{R}, \mathsf{q}\rangle\ \mathsf{T}_1; \mathsf{T}_2$	Sub-protocol on role sets
	$\mid \mathsf{q!Quit.End} \mid x?\mathsf{Quit.End}$	Request/Accept exit from the role set

where I is a finite non-empty index set and $m_h \neq m_k$ for $h \neq k$.

When $|I| = 1$ we drop the brackets and write $Q!m\langle\overline{S}\rangle.\mathsf{T}$ and $r?m\langle\overline{S}\rangle.\mathsf{T}$ for short.

For local types, we have the standard constructs: *choice of outputs* (sending a message with label within a finite set of labels) also called *internal choices*, *choice of inputs* (receiving a message with a label as before) also called *external choices*, and guarded recursion. As for global types the receiver of a send can be either a single participant or a role set. Then we have the `map` construct and the request for and acceptance of the exit from the interaction. The `map` construct can only occur in the local type of the coordinator of a role set and similar restrictions apply for the request and acceptance of the `Quit` message. The first can only occur in a participant in a role set and the second in a coordinator. These restrictions are enforced by the projection. Note that, as for global types, only the body of a `map` is followed by `;`.

Listings 2–4 contain the local types of the `broker`, the `initiator`, and the role set participants, corresponding to the global type of Listing 1. Again, we use the Java-like syntax with the `choice` construct specifying, as for the global types, the leader of the choice. Branches of internal choices will start with a message `to` another participant (the labels of the messages should be different in different branches), and external choices will start with a message `from` the leader. These types are obtained as *projections* from the global type of Listing 1.

```
1  local protocol myBrokering at role broker(
       role initiator, roleset Subagents:broker)
2  {
3    proxy(string) from initiator.
4    choice at broker{
5      refuse() to initiator.stop() to Subagents
         .End
6    } or {
7      agree() to initiator.findAgent(string) to
           Subagents.
8      map role agent:<Subagents,broker>{
9        choice at Subagents{
10         notPossible() from agent.QUIT() from
             agent.End
11       } or {
12         canDo() from agent.End
13       }
14     } ;
15     choice at broker{
16       failNoMatch() to initiator.stop() to
           Subagents.End
17     } or {
18       foundMatches() to initiator.continue()
           to Subagents.
19       map role agent:<Subagents,broker>{
20         rec COLLDATI: {
21           sendMore() to agent.
22           choice at Subagents{
23             addData(string) from agent.loop
                 COLLDATI
24           } or {
25             noMoreData() from agent.End
26           } or {
27             someError() from agent.QUIT()
                 from agent.End
28           }
29         }
30       } ;
31       choice at broker{
32         replyFromSubagents(string) to
             initiator.End
33       } or {
34         failBrokering() to initiator.End
35 }}}}
```

Listing 2. Local type for `broker`.

2.3 Projection of global types onto session/local types

Projection is the main tool to ensure that the protocol described by the global type can be implemented by a set of participants with their associated processes.

Assuming Z to be either a participant or a role set, the *projection of a global type* G *on* Z, *dubbed* $\mathsf{G} \upharpoonright Z$, is roughly obtained by erasing Z from the communications of G. The formal definition of projection is given in Formal Definition of Projection. Here we show it on our example.

The projection of the communication $p \to Q\, m\langle\overline{S}\rangle.\mathsf{G}$ on the sender p is $Q!m\langle\overline{S}\rangle.\mathsf{G} \upharpoonright p$ and on the receiver Q is $p?m\langle\overline{S}\rangle.\mathsf{G} \upharpoonright Q$. Moreover, the projection on a participant Z which is neither p nor Q is just $\mathsf{G} \upharpoonright Z$. E.g., line 3 of Listing 1 is projected on the `broker` and the `initiator`, producing line 3 of Listings 2, 3 respectively. Looking instead at the `Subagents` in Listing 4, the local type starts with the projection of line 4 of the global type.

The projection of a choice $p \to Q\{m_i\langle\overline{S}^i\rangle.\mathsf{G}_i\}_{i\in I}$ on a participant is a combination of the projections of its branches on the participant. So if we project on the sender, we combine $\{Q!m_i\langle\overline{S}^i\rangle.\mathsf{G}_i \upharpoonright p\}_{i\in I}$ to form an internal choice; on the receiver, we combine $\{p?m_i\langle\overline{S}^i\rangle.\mathsf{G}_i \upharpoonright Q\}_{i\in I}$ to form an external choice. In

```
1  local protocol myBrokering at role initiator(
       role broker)
2  {
3    proxy(string) to broker.
4    choice at broker{
5      refuse() from broker.End
6    } or {
7      agree() from broker.
8      choice at broker{
9        failNoMatch() from broker.End
10     } or {
11     foundMatches() from broker.
12     choice at broker{
13       replyFromSubagents(string) from broker.
             End
14     } or {
15       failBrokering() from broker.End
16 }}}}
```

Listing 3. Local type for `initiator`.

```
1  local protocol myBrokering at roleset
       Subagents:broker(role broker) {
2    choice at broker{
3      stop() from broker.End
4    } or {
5      findAgent(string) from broker.
6      choice at Subagents{
7        notPossible() to broker.QUIT() to
             broker.End
8      } or {
9        canDo() to broker.
10       choice at broker{
11         stop() from broker.End
12       } or {
13         continue() from broker.
14         rec COLLDATI: {
15           sendMore() from broker.
16           choice at Subagents{
17             addData(string) to broker.loop
                   COLLDATI
18           } or {
19             noMoreData() to broker.
20           } or {
21             someError() to broker.QUIT() to
                   broker.End
22           }
23 }}}}}
```

Listing 4. Local type for `Subagents`.

our Java-like syntax, the branches of a choice have to start with a communication from the leader of the choice, say q, to a Z. So projecting on the leader of the choice, we obtain a correct internal choice (branches starting with a send to Z), and projecting on the receiver, we obtain an external choice (branches starting with a receive from q). For example, the initial choice, line 4, of Listing 1, is projected on the `broker`, producing the choice starting at line 4 of Listing 2, where the branches start at lines 5 and 7 with a message sent to the `initiator`. The projection on the `initiator` starts at line 4 of Listing 3, with branches at lines 5 and 7, with an initial receive from `broker`. But what about the participants in the role set of the `Subagents`, and in general, on participants different from the sender and the receiver of the communication? Assume the participant is Z; we should combine the set $\{\mathsf{G}_i \upharpoonright Z\}_{i\in I}$. What we do is we combine the projections of the branches with a *merge* operation, which is successful if either all the local types in the set are equal or they all start with the receive of a message with different labels from the same sender so that we can combine them and produce an external choice. Intuitively, this means that either a participant which is not involved in the first communication of a choice does the same actions independently from the branch chosen, or it needs to be informed first on which branch was selected. Looking again at lines 5 and 7 of Listing 1, we can see that the second communication (in both lines) is a communication from `broker` to `Subagents`. The merge produces the external choice starting at line 3 of Listing 4.

Projection of recursion, $\mu X.\mathsf{G}$, on a participant Z involved in some communication in G is $\mu X.(\mathsf{G} \upharpoonright Z)$, where the projection of the recursion variable is simply X, so that the local type is in turn a recursive type. When Z is not involved in any communication in G, the projection is End. Let us look at the recursion construct at lines 20–29 of Listing 1, which is iterating the request from the `broker` to the `Subagents` for more data until the `agent` does not have any more data to send or decides to quit. The projection on the `broker` is at lines 20–29 of Listing 2, and the one on `Subagents` is at lines 15–22 of Listing 4, whereas there is no recursive construct in the projection on the `initiator`. Projection of a $\mathtt{map}\ x{:}\langle \mathsf{R}, \mathsf{q}\rangle\ \mathsf{G}_1; \mathsf{G}_2$ produces $\mathtt{map}\ x{:}\langle \mathsf{R}, \mathsf{q}\rangle\ (\mathsf{G}_1 \upharpoonright \mathsf{q}); (\mathsf{G}_2 \upharpoonright \mathsf{q})$ when projecting on the coordinator of the role set. That is, $\mathsf{G}_2 \upharpoonright \mathsf{q}$ will be executed when the execution of sub-protocol $\mathsf{G}_1 \upharpoonright \mathsf{q}$ with all the members of `Subagents` ends, with either Quit or End. Consider the $\mathtt{map}$ constructs, lines 8–14 and 19–30, of Listing 1. The sub-protocols end with either `QUIT`, the second communications of lines 10 and 27, or `End` at lines 12 and 25. The projection of these $\mathtt{map}$ on the `broker` is at the same lines in Listing 2, where the projection of `QUIT` is a receive of a message labeled `QUIT` from a participant of the role set `Subagents`, followed by `End`. The projection of $\mathtt{map}$ on all the other participants Z produces the projection of $\mathsf{G}_1 \upharpoonright Z$ in which End is substituted with $\mathsf{G}_2 \upharpoonright Z$, and if $Z = x$, i.e. a member of r, the projection of Quit is $\mathsf{q}!\mathsf{Quit}.\mathsf{End}$. That is, it is the responsibility of the coordinator to wait for the completion, denoted by the use of semicolon ";", of the sub-protocol on all the participants of the role set R. The projection of the first $\mathtt{map}$, lines 8–14, of Listing 1 on the `Subagents` starts at line 7 of Listing 4 with the projection of the `choice` (body of the $\mathtt{map}$). We can see that the branch starting with sending to `broker` the message labeled `notPossible()` after sending the message labeled `QUIT` ends the whole protocol. Instead the branch starting with sending the message labeled `canDo()` follows with the projection of the global type after ";" on `Subagents`. Similarly for the second $\mathtt{map}$. Here the projection on `Subagents` starts at line 15 with the `rec` construct. In this case, also the branch starting with sending `noMoreData()` to the `broker` (line 20 of Listing 4) ends the protocol, since the last choice of the global protocol (lines 31–35) does not involve the `Subagents`.

Definition 2.3. A global type is *well-formed* if the projection is defined for all the participants and role sets.

Consider removing the communication `stop()from broker to Subagents` from line 5 of Listing 1. The projection of this branch of the choice on `Subagents` would be `End`. Indeed, the protocol should end with the `broker` sending to the `initiator` the message labeled `failNoMatch()`. On the contrary, in the other branch, i.e., if the `broker` had sent the message labeled `agree` to the `initiator`, the `Subagents` should do the interaction prescribed by the protocol. But how could `Subagents` know whether they have to leave the protocol or wait for the message labeled `findAgent(...)`? The previously described merge in this case would fail, since on one of the branches the projection is `End` and on the other is a receiver of the message labeled `findAgent(...)` from the broker. So the projection would not be defined. Consider now the other choice at lines 22–29 of Listing 1. Here the `initiator` is neither the leader of the choice nor does it receive the first message. In this case, however, the `initiator` is not involved in any interaction in the branches of the choice, so its projection would be End in all branches. If we look at the projection on the `initiator`, this choice, correctly, does not appear.

2.4 Session types semantics

The semantics of session types is given as a *labeled reduction* on *configurations* consisting of the participants associated with the session type that describes their protocol. Since communication is asynchronous, there can be messages that were sent from a participant to another that have not been read yet. To record these messages, each participant, p, has a *queue* containing the messages received but not yet read. A triple $\langle \mathsf{q}, m, \overline{S} \rangle$ denotes a *message* sent by participant q, labeled m and carrying data of type $\overline{S}$. *Queues* μ are defined by:

$$\mu ::= \emptyset \mid \langle \mathsf{q}, m, \overline{S} \rangle \cdot \mu$$

The order of messages in the queue is the order in which they will be read. Order matters only between messages with the same sender, so we consider message queues modulo the following structural equivalence:

$$\langle \mathsf{q}, m, \overline{S} \rangle \cdot \langle \mathsf{r}, m', \overline{S}' \rangle \cdot \mathcal{M}' \equiv \langle \mathsf{r}, m', \overline{S}' \rangle \cdot \langle \mathsf{q}, m, \overline{S} \rangle \cdot \mathcal{M}' \text{if } \mathsf{p} \neq \mathsf{r}$$

The two equivalent queues $\langle \mathsf{q}, m, \overline{S} \rangle \cdot \langle \mathsf{r}, m', \overline{S}' \rangle \equiv \langle \mathsf{r}, m', \overline{S}' \rangle \cdot \langle \mathsf{q}, m, \overline{S} \rangle$ represent a situation in which both participants q and r sent a message to p, and p has not yet read these messages. This situation may arise in a multiparty session with asynchronous communication.

The *label of the reduction*, α, records the interaction that happens during the reduction:

$$\alpha ::= \mathsf{p}!\mathsf{q}.m \mid \mathsf{q}?\mathsf{p}.m$$

We have the asynchronous send of a message with label m from participant p to participant q and the actual reading by participant q of the message labeled m sent by participant p.

A *configuration*, $\mathbb{N}$, is the parallel composition of *participants*, which may be simple participants or role coordinators.

$$\begin{array}{lll} \mathbb{N} ::= \mathsf{p}[\![\, \mu, \mathsf{T} \,]\!] \mid \mathsf{p}[\![\, \mu, \mathcal{R}, \mathsf{T} \,]\!] \mid \mathbb{N} \parallel \mathbb{N} & & \text{configuration} \\ \mathsf{T} ::= \ldots \mid \mathsf{map}(\mathsf{T}_1 \ldots \mathsf{T}_n); \mathsf{T} & & \text{(extended) types} \end{array}$$

In case the participant is a coordinator, in addition to the queue of messages, it also contains the list of the names of the participants in the role set, $\mathcal{R}$. Moreover, to trace the execution of the sub-protocols in a `map`, we extend the syntax of types by adding a `map` *context* that records the progress in the execution of the sub-protocols of the participants of a role set. We assume the standard laws for parallel composition, stating that $\parallel$ is associative, commutative, and has neutral elements $\mathsf{p}[\![\, \emptyset, \mathsf{End} \,]\!]$ or $\mathsf{p}[\![\, \emptyset, \mathsf{R}, \mathsf{End} \,]\!]$. These laws give rise to the structural congruence on configurations, which we assume when writing the reduction rules.

In Figure 2 we present the rules of the semantics of session types. In the top part of the figure, we define a precongruence introducing and removing the `map` context, while the reduction rules describe the communications taking place. In writing the rules, we use _ for elements of the configuration whose value does not affect the reduction. For example, consider Rule (SND), $\mathsf{p}[\![\, \_, \mathsf{q}!\{m_i\langle \overline{S}^i \rangle.\mathsf{T}_i\}_{i\in I} \,]\!]$ means that the rule can be applied to a participant p with any queue and also that p could be a coordinator or any other participant. Similarly, $\mathsf{q}[\![\, \mu, \_\mathsf{T} \,]\!]$ means that the participant could be a coordinator or any other participant. Here, the queue is referred to since on the right-hand side of the reduction we add a message to it.

The precongruence Rule (MAP-INIT) transforms the `map` construct in a `map` context by instantiating the type of the sub-protocol for all the participants in the role set. Note that, since participant variables may only occur in a `map` construct, this rule ensures that there will be no variables as senders or receivers of communications.

Rule (MAP-END), removes the `map` context when all the participants in the role set have terminated the execution of the sub-protocol.

Finally, we can unroll a recursive type, Rule (REC), by substituting the occurrences of the recursion variable with the type itself.

With Rule (SND), the participant p, after sending one of the messages with a label in $\{m_i\}_{i\in I}$ to q, continues with the type corresponding to the selected label. Sending means inserting at the end of the queue of q the message specifying that its sender is p and the sort of the payload.

In Rule (SND-RS), a coordinator sends a message to all the participants of the role set it coordinates. Note that here, the sender must be a coordinator, i.e., must have the field with the list of the participants in the role set, and the receivers are the simple participants in its list $\mathcal{R}$. A participant q removes a message from its queue, provided that its associated process is an external choice having a message with a label and sorts matching the one in the queue. The message is then removed from the queue, and the process continues with the selected type. The congruence on the queue may be needed to bring to the front the first message from p.

The Rules (SND-QUIT) and (RCV-QUIT) are similar to send and receive except for the fact that send necessarily goes from a participant to a coordinator and receive can only be found in a coordinator. The existence of the projection of the global type ensures these restrictions.

Rule (MAP) executes the reduction in one of the role set participants, and Rule (PRE-CONG) closes the reduction by the precongruence.

$$(\text{MAP-INIT})\ \mathsf{q}[\![\ \_, \mathcal{R}, \mathsf{map}\ x{:}\langle \mathsf{R}, \mathsf{q}\rangle\ \mathsf{T};\ \mathsf{T}'\ ]\!] \triangleright \mathsf{q}[\![\ \_, \mathcal{R}, \mathsf{map}(\mathsf{T}_1 \ldots \mathsf{T}_n);\ \mathsf{T}'\ ]\!] \qquad \mathcal{R} = \mathsf{p}_1 \ldots \mathsf{p}_n \quad n \geq 1,\ \mathsf{T}_i = \mathsf{T}[\mathsf{p}_i/x]$$

$$(\text{MAP-END})\ \mathsf{q}[\![\ \_, \mathcal{R}, \mathsf{map}(\mathsf{End} \ldots \mathsf{End});\ \mathsf{T}'\ ]\!] \triangleright \mathsf{q}[\![\ \_, \mathcal{R}, \mathsf{T}'\ ]\!]$$

$$(\text{REC})\ \mathsf{p}[\![\ \_, \mu X.\mathsf{T}\ ]\!] \triangleright \mathsf{p}[\![\ \_, \mathsf{T}[\mu X.\mathsf{T}/X]\ ]\!]$$

$$(\text{SND})\ \mathsf{p}[\![\ \_, \mathsf{q}!\{m_i\langle \vec{S}^i\rangle.\mathsf{T}_i\}_{i\in I}\ ]\!] \parallel \mathsf{q}[\![\ \mu, \_\ \mathsf{T}\ ]\!] \parallel \mathbb{N} \xrightarrow{\mathsf{p!q}.m_h} \mathsf{p}[\![\ \_, \mathsf{T}_h\ ]\!] \parallel \mathsf{q}[\![\ \mu \cdot \langle \mathsf{p}, m_h, \vec{S}^h\rangle, \_\ \mathsf{T}\ ]\!] \parallel \mathbb{N} \quad h \in I$$

$$(\text{SND-RS})\ \mathsf{q}[\![\ \_, \mathcal{R}, \mathsf{R}!\{m_i\langle \vec{S}^i\rangle.\mathsf{T}_i\}_{i\in I}\ ]\!] \parallel_{\mathsf{p}\in\mathcal{R}} \mathsf{p}[\![\ \mu_\mathsf{p}, \mathsf{T}_\mathsf{p}\ ]\!] \parallel \mathbb{N} \xrightarrow{\mathsf{p!R}.m_h} \mathsf{q}[\![\ \_, \mathcal{R}, \mathsf{T}_h\ ]\!] \parallel_{\mathsf{p}\in\mathcal{R}} \mathsf{p}[\![\ \mu_\mathsf{p} \cdot \langle \mathsf{p}, m_h, \vec{S}^h\rangle, \mathsf{T}_\mathsf{p}\ ]\!] \parallel \mathbb{N} \quad h \in I$$

$$(\text{RCV})\ \mathsf{q}[\![\ \langle \mathsf{p}, m_h, \vec{S}^h\rangle \cdot \mu, \_\ \mathsf{p}?\{m_i\langle \vec{S}^i\rangle.\mathsf{T}_i\}_{i\in I}\ ]\!] \parallel \mathbb{N} \xrightarrow{\mathsf{q?p}.m_h} \mathsf{q}[\![\ \mu, \_\ \mathsf{T}_h\ ]\!] \parallel \mathbb{N} \quad h \in I$$

$$(\text{SND-QUIT})\ \mathsf{p}[\![\ \_, \mathsf{q}!\mathsf{Quit}.\mathsf{End}\ ]\!] \parallel \mathsf{q}[\![\ \mu, m, \mathsf{T}\ ]\!] \parallel \mathbb{N} \xrightarrow{\mathsf{p!q.Quit}} \mathsf{p}[\![\ \_, \mathsf{End}\ ]\!] \parallel \mathsf{q}[\![\ \mu \cdot \langle \mathsf{p}, \mathsf{Quit}, \epsilon\rangle, \mathcal{R}, \mathsf{T}\ ]\!] \parallel \mathbb{N}$$

$$(\text{RCV-QUIT})\ \mathsf{q}[\![\ \langle \mathsf{p}, \mathsf{Quit}, \epsilon\rangle \cdot \mu, \mathcal{R}, \mathsf{T}\ ]\!] \parallel \mathbb{N} \xrightarrow{\mathsf{q?p.Quit}} \mathsf{q}[\![\ \mu, \mathcal{R} \setminus \{\mathsf{p}\}, \mathsf{T}\ ]\!] \parallel \mathbb{N}$$

$$(\text{MAP})\ \frac{\mathsf{q}[\![\ \mu, \mathcal{R}, \mathsf{T}_i\ ]\!] \parallel \mathbb{N} \xrightarrow{\alpha} \mathsf{q}[\![\ \mu', \mathcal{R}', \mathsf{T}_i'\ ]\!] \parallel \mathbb{N}'}{\mathsf{q}[\![\ \mu, \mathcal{R}, \mathsf{map}(\mathsf{T}_1 \ldots \mathsf{T}_i \ldots \mathsf{T}_n);\ \mathsf{T}'\ ]\!] \parallel \mathbb{N} \xrightarrow{\alpha} \mathsf{q}[\![\ \mu', \mathcal{R}', \mathsf{map}(\mathsf{T}_1 \ldots \mathsf{T}_i' \ldots \mathsf{T}_n);\ \mathsf{T}'\ ]\!] \parallel \mathbb{N}'} \quad 1 \leq i \leq n$$

$$(\text{PRE-CONG})\ \frac{\mathsf{q}[\![\ \_, \mathsf{T}'\ ]\!] \parallel \mathbb{N} \xrightarrow{\alpha} \mathbb{N}'}{\mathsf{q}[\![\ \_, \mathsf{T}\ ]\!] \parallel \mathbb{N} \xrightarrow{\alpha} \mathbb{N}'} \quad \mathsf{q}[\![\ \_, \mathsf{T}\ ]\!] \triangleright \mathsf{q}[\![\ \_, \mathsf{T}'\ ]\!]$$

FIGURE 2
Precongruence and reduction rules for asynchronous sessions with role sets.

2.5 Limitations of our modeling

Our modeling provides a faithful representation of the principal FIPA agent interaction protocols described at http://www.fipa.org/repository, where the specifications are given in a semi-formal manner through UML sequence diagrams complemented by textual annotations. Nevertheless, when considered in the broader setting of interaction protocols, our approach does not fully account for their dynamic evolution. Specifically, we impose the following restrictions:

- each coordinator is associated with exactly one role,
- agents are statically bound to their roles and may only leave them, in which case their execution terminates.

In principle, these restrictions can be relaxed following the approach of Deniélou and Yoshida (2011), which allows participants to dynamically join and leave roles. However, in order to guarantee input-lock freedom and to prevent orphan messages, all communications must be routed through a centralized registry responsible for maintaining the association between participants and roles. We contend that this solution is not fully aligned with the agent-oriented paradigm and is likely to introduce a performance bottleneck. This perspective is supported by the fact that the line of work initiated in Deniélou and Yoshida (2011) has not been subsequently pursued, whereas later extensions of global types accommodating role sets—reviewed in the introduction—tend to converge toward solutions closer to ours.

By contrast, the restriction on the map construct, which disallows nesting, is of a purely technical nature. The definitions of global and local types, their projections, and the dynamic semantics presented in Figure 2 remain unaffected. The restriction becomes relevant only in the implementation of the agents corresponding to local types. As discussed in Section 4, the coordinator agent maintains only minimal information about the role set participants: their addresses and, in the case of a map construct, the number of participants that have not yet completed their interaction. The support of nested maps would require maintaining a tree of pending interactions. While feasible, such an extension would also necessitate a substantial generalization of the correctness proof. Note that an alternative approach to alleviate this limitation, as proposed in Viering et al. (2021), consists in structuring global types into hierarchical parent-child subtypes, whose maintenance is delegated to the running system. Such an approach, however, presupposes the existence of auxiliary agents—external to the protocol participants—entrusted with tracking the refinement and evolution of subtypes. This reliance on meta-level supervisory entities introduces an architectural overhead that is at odds with the autonomy and locality principles underpinning agent-based systems.

3 Featherweight Jadescript

We introduce the subset of Jadescript, FJS, that is relevant to our translation. In defining FJS we adopt the standard agent terminology, calling agents what in the session type world are participants.

An FJS *program* is a sequence of *agent and behavior declarations*, *Ag* and *Bh*, defined by

$$Ag ::= \texttt{agent}\ a\ [\texttt{property rs,cnt}]\ \texttt{on create}\ P \qquad \text{agent declaration}$$

$$Bd ::= id(\overline{x}) = Bh \qquad \text{behaviour declaration}$$

$$Bh ::= \langle P, a_i?\{m_i(\overline{x}^i).P_i\}_{i\in I}\rangle \qquad \text{behaviour handlers}$$

$$
\begin{array}{lll}
P & ::= & \mathtt{0} \mid a!m(\overline{e}).P \mid \mathtt{rs}!m(\overline{e}).P \mid id(\overline{e}).P \mid st.P & \text{process} \\
 & \mid & \mathtt{if}\ e_1{:}P_1\ \mathtt{elif}\ e_2{:}P_2 \cdots \mathtt{else}\ P_n\ (n \geq 2) & \\
v & ::= & \mathtt{true} \mid \mathtt{false} \mid \mathsf{n} \mid \cdots & \text{value} \\
e & ::= & v \mid x \mid e\ \mathsf{op}\ e \mid \cdots & \text{general expression} \\
 & \mid & \mathtt{rs}[i] & \text{access to role set} \\
st & ::= & \mathtt{rmv}(\mathtt{rs}, a) \mid \mathtt{init}(\mathtt{cnt}) \mid \mathtt{cnt}\text{--} & \text{statement}
\end{array}
$$

where I is non-empty, and for all $h, k \in I$, $h \neq k$, either $m_h \neq m_k$ or $a_h \neq a_k$. An *agent* has a name and a process P which is executed when the agent is created. For agents who are coordinators, we also have the fields, `rs` and `cnt`, denoting the set of the agents of the role set it coordinates and a counter used to iterate on them. We use a, b, c for agent names.

Behaviors specify how an agent reacts to events. We consider the *internal event* raised by the scheduling of the behavior for *execution*, and the *external events* raised by the *reception of messages* from other agents. In particular, P is the process executed when the behavior is selected for execution and $a_i?\{m_i(\overline{x}^i).P_i\}_{i\in I}$, where I is non-empty, and for all $h, k \in I$, $h \neq k$, either $m_h \neq m_k$ or $a_h \neq a_k$, says that if m is received from a_i, then the process P_i is executed by substituting the variables $\overline{x}^i$ with the values specified by the received message.

The process `0` terminates a behavior, $\mathtt{rs}!m(\overline{e}).P$ (or $a!m(\overline{e}).P$) is a send of the message m to all the agents of the role set `rs` (or to the agent a) followed by the execution of a P, and $id(\overline{e}).P$ is the *activation of the behavior id* followed by the execution of P. Activation of a behavior does not start the execution of the behavior but only adds it to the list of active behaviors of an agent. The process $st.P$ executes the statement st followed by P. The statement $\mathtt{rmv}(\mathtt{rs}, a)$ removes the agent a from the role set list (when a sends the message Quit to its coordinator). The other two statements are used in the translation of the `map` construct to wait for the end of the execution of the sub-protocol of all the agents in the role set. The conditional construct `if-elif-else` is an internal choice among the processes P_i, $1 \leq i \leq n$. The process executed is P_j, where j is such that the guard e_j evaluates to `true` and all e_i, $i < j$, evaluate to `false`. We assume an expression language including booleans and integers and some basic operations.

In FJS we do not include the *ontology*, i.e., the classification of the content of messages in the considered domain, present in Jadescript and, as customary, in agent languages. The ontology is, however, considered in our implementation, which assumes that it is provided with the input global type.

3.1 Operational semantics of FJS

The operational semantics of FJS is specified, as for session types, by a *labeled reduction* on configurations in Figure 3. *Configurations*, χ, are defined by

$$
\begin{array}{lll}
\chi & ::= & \mathsf{A}_1 \parallel \cdots \parallel \mathsf{A}_n \quad \text{configuration} \\
\mathsf{A} & ::= & a[\![\, \mathsf{mb}, [(Rs, cnt),]P, \overline{Bh} \,]\!] \quad \text{runtime agent} \\
\mathsf{mb} & ::= & \overline{\langle a, m, \overline{v} \rangle} \quad \text{mailbox}
\end{array}
$$

A configuration is the parallel composition of *runtime agents*. An agent is identified by a name, e.g. a, and to it the following are associated: a *mailbox* mb, the running *process* P, and a multiset of *active behaviors*, $\overline{Bh}$. Mailboxes, like queues for participants of Section 2.4, contain the messages specifying the sender of the message, its label, and, in this case, the values of the payload. Also here, messages can be rearranged according to the equivalence that allows one to swap messages coming from different agents, so that only the order of messages coming from the same agent is relevant. As for session types, we assume the standard laws for parallel composition, stating that $\parallel$ is associative, commutative, and has a neutral element $a[\![\, \emptyset, \mathtt{0}, \emptyset \,]\!]$.

The *labels of the reduction*, ℓ, as for session types, record the interaction that happens during the reduction:

$$\ell ::= a!b.m \mid a?b.m \mid \tau$$

and in addition, we have the label τ, which is used for a reduction that does not involve an interaction between agents.

The labels of the reductions and their meanings are the ones of Section 2.4.

The rules of the labeled reductions are given in Figure 3, where with $e \Downarrow v$ we indicate that the evaluation of e produces the value v. Similarly for sequences of expressions, $\overline{e} \Downarrow \overline{v}$. We use the same convention about the use of _ as in Section 2.4.

Rule (SND-MSG) sends one of the messages to an agent, by adding it at the end of the receiver's mailbox and then continues the execution with the following process.

In Rule (SND-MSG-RS) the message is put in the mailbox of all the agents in the list *Rs*.

Rule (ACTV) adds a behavior to the set of active behaviors of the agent. The behavior must be one of the defined ones. The formal parameters are substituted by the values resulting from the evaluation of the argument of the behavior.

Rule (STAT) executes a statement before continuing with the following process. The statement modifies either the list of agents in the role set, if the statement asks to remove an agent from it, or the counter. The counter is used in the translation of the `map` construct present in the type of a coordinator to count the number of agents in the role set that have completed the sub-protocol body of the `map`. Its initialization is done with the statement `init(cnt)` that sets the field to the current length of the list *Rs*.

The conditional construct executes the first process whose guard is `true`, Rules (IF-THEN) and (IF-CONT) , or the one in the `else` branch if none of them was `true`, Rule (IF-ELSE). The two rules that follow are executed when the agent is idle.

Rule (EXEC) schedules for execution a behavior from the set of active behaviors and starts executing the process in its handler for execution.

Rule (RCV) specifies how the event of receiving a message is handled. If the agent is not running any process, and has already executed its execution handler, in case there is a message in the mailbox with sender, message label and number of parameters matching one of the handlers, it removes the message from the mailbox and executes the process associated with its handler after substituting the variables with the payload of the message.

We assume that behaviors with `0` as the handler for execution and empty message handlers are removed from the list of active behaviors.

$$\text{(SND-MSG)}\quad a[\![\ \_, b!m(\tilde{e}).P, \overline{Bh}\ ]\!] \parallel b[\![\ \mathsf{mb}, \_\,P', \overline{Bh}'\ ]\!] \parallel \chi \xrightarrow{a!b.m} a[\![\ \_, P, \overline{Bh}\ ]\!] \parallel b[\![\ \mathsf{mb}\ \langle a, m, \tilde{v}\rangle, \_\,P', \overline{Bh}'\ ]\!] \parallel \chi \qquad \tilde{e} \Downarrow \tilde{v}$$

$$\text{(SND-MSG-RS)}\quad a[\![\ \_, (Rs, cnt), \mathtt{rs}!m(\tilde{e}).P, \overline{Bh}\ ]\!] \parallel_{b\in Rs} b[\![\ \mathsf{mb}_b, P_b, \overline{Bh}_b\ ]\!] \parallel \chi \xrightarrow{a!\mathtt{rs}.m} a[\![\ \_, (Rs, cnt), P, \overline{Bh}\ ]\!] \parallel_{b\in Rs} b[\![\ \mathsf{mb}_b\ \langle a, m, \tilde{v}\rangle, P_b, \overline{Bh}_b\ ]\!] \parallel \chi \qquad \tilde{e} \Downarrow \tilde{v}$$

$$\text{(ACTV)}\quad a[\![\ \_, id(\tilde{e}).P, \overline{Bh}\ ]\!] \parallel \chi \xrightarrow{\tau} a[\![\ \_, P, \overline{Bh}\ Bh'[\tilde{v}/\tilde{x}]\ ]\!] \parallel \chi \qquad id(\tilde{x}) = Bh' \ \wedge\ \tilde{e} \Downarrow \tilde{v}$$

$$\text{(STAT)}\quad a[\![\ \_, (Rs, cnt), st.P, \overline{Bh}\ ]\!] \parallel \chi \xrightarrow{\tau} a[\![\ \_, (Rs', cnt'), P, \overline{Bh}\ Bh'[\tilde{v}/\tilde{x}]\ ]\!] \parallel \chi$$
$$(Rs', cnt') = \begin{cases} (Rs, len(Rs)) & st = \mathtt{init(cnt)} \\ (Rs, cnt-1) & st = \mathtt{cnt\text{--}} \\ (Rs \setminus b, cnt) & st = \mathtt{rmv(rs}, b\mathtt{)} \end{cases}$$

$$\text{(IF-THEN)}\quad a[\![\ \_, \mathtt{if}\ e_1{:}P_1\ \cdots, \overline{Bh}\ ]\!] \parallel \chi \xrightarrow{\tau} a[\![\ \_, P_1, \overline{Bh}\ ]\!] \parallel \chi \qquad e_1 \Downarrow \mathtt{true}$$

$$\text{(IF-CNT)}\quad a[\![\ \_, \mathtt{if}\ e_1{:}P_1\ \mathtt{elif}\ e_2{:}P_2 \cdots, \overline{Bh}\ ]\!] \parallel \chi \xrightarrow{\tau} a[\![\ \_, \mathtt{if}\ e_2{:}P_2\ \cdots, \overline{Bh}\ ]\!] \parallel \chi \qquad e_1 \Downarrow \mathtt{false}$$

$$\text{(IF-ELSE)}\quad a[\![\ \_, \mathtt{if}\ e{:}P\ \mathtt{else}\ P', \overline{Bh}\ ]\!] \parallel \chi \xrightarrow{\tau} a[\![\ \_, P', \overline{Bh}\ ]\!] \parallel \chi \qquad e \Downarrow \mathtt{false}$$

$$\text{(EXC)}\quad a[\![\ \_, \mathtt{0}, \overline{Bh}\ \langle P, a_i?\{m_i(\tilde{x}^i).P_i\}_{i\in I}\rangle\ ]\!] \parallel \chi \xrightarrow{\tau} a[\![\ \_, P, \overline{Bh}\ \langle \mathtt{0}, a_i?\{m_i(\tilde{x}^i).P_i\}_{i\in I}\rangle\ ]\!] \parallel \chi \qquad P \neq \mathtt{0}$$

$$\text{(RCV)}\quad a[\![\ \langle a_j, m_j, \tilde{v}\rangle\ \mathsf{mb}, \_\,\mathtt{0}, \overline{Bh}\ \langle \mathtt{0}, a_i?\{m_i(\tilde{x}^i).P_i\}_{i\in I}\rangle\ ]\!] \parallel \chi \xrightarrow{a?a_j.m_j} a[\![\ \mathsf{mb}, \_\,P_j[\tilde{v}/\tilde{x}^j], \overline{Bh}\ ]\!] \parallel \chi \qquad j \in I$$

FIGURE 3
Operational semantics of FJS.

4 Translation from local types to FJS

In this section, we present the formalization of the translation between session types and FJS agents and its correctness.

The syntax-directed translation, presented in Figure 4, takes as input a session type and an association between recursion variables and names of behaviors, and returns an FJS agent and a list of definitions of behaviors. Moreover, the translation is indexed by a process whose use will be made clear when looking at the translation of the `map` construct.

The syntax-directed rules of the translation are given in Figure 4. We assume a correspondence between the identifiers used as senders and receivers of messages in the session types and the ones in FJS given by

$$(\!|\mathsf{p}|\!) = a_{\mathsf{p}} \qquad (\!|\mathsf{R}|\!) = \mathtt{rs} \qquad (\!|x|\!) = x$$

As expected, a participant and a role set denote the corresponding agent and role set. The variable of the `map` construct is left unchanged, since it will be substituted by the translation of `map` with the reference to an agent in the role set.

An internal choice, Rule (TR-INT-CH), is translated into the activation of a behavior, *id*(), and then stops. The behavior *id*() contains only the process executed when the behavior is scheduled for execution. This process is an FJS conditional construct that, under some conditions, $\mathbf{c}_i$, selects the sending of the message with label m_i to $(\!|Q|\!)$, followed by the process which is the translation of the type in the corresponding branch. Here, **e** and **c** are placeholders for expressions and conditions, respectively, that have to be filled when instantiating the translation to produce a Jadescript program. The behaviors generated by the translation of the branches of the choice are collected together, and the newly defined behavior is added. Note that its name is fresh so that it does not collide with the ones in $\overline{Bd}^i$ for $i \in I$.

Rule (TR-EXT-CH) produces a process that activates a behavior, *id*(), and then ends its execution. The behavior *id*() has only incoming message handlers. In particular, the handlers on reception of one of the messages with labels m_i from $(\!|p|\!)$ will execute the process corresponding to the selected branch. So, once scheduled for execution, *id*() will wait until one of these messages is present in its mailbox. The behaviors generated are as in the previous case.

The following two rules translate a recursive interaction. In Rule (TR-REC), we define the behavior *id*(), which contains the process P executed when the behavior is scheduled for execution. The code of the process P is obtained by the translation of the body of the type `T`. The process associated with the recursion variable activates *id*() and then stops. So it does exactly what the translation of $\mu X.\mathsf{T}$ does. Rule (TR-VAR) returns the process associated with the recursion variable that as we can see from Rule (TR-REC), coincides with the translation of $\mu X.\mathsf{T}$.

The translation of the `map` construct is the most complex one because we have to produce a process that activates the execution of the sub-protocol corresponding to the body of the `map` on all the agents in the role set, and then controls the end of their executions before continuing with the execution of the process P' translating the type after the semicolon. For each agent in the role set we define a behavior $id_i()$ by replacing the variable x with the i-th agent name in the role set in the process P obtained from the translation of the sub-protocol `T`. Also, in the behaviors obtained from this translation, the variable x is

$$\text{(TR-INT-CH)}\ \frac{([\bar{X} \mapsto \bar{P},\ T_i])^P = \langle P_i', \overline{Bd}^i \rangle \quad P_i'' = ([Q])!m_i(\bar{\mathbf{e}}^i) \cdot P_i' \quad i \in I \quad id \text{ fresh}}{([\bar{X} \mapsto \bar{P},\ Q!\{m_i\langle \bar{S}^i \rangle \cdot \top_i\}_{i\in I}])^P = \langle id() \cdot 0, \bigcup_{i\in I} \overline{Bd}^i \cup \{id() = \langle P', \emptyset \rangle\}\rangle}$$

where

$$P' = \mathit{if}\ c_1 : P_1''\ \mathit{elif}\ c_2 : P_2'' \cdots \mathit{else}\ P_n''$$

$$\text{(TR-EXT-CH)}\ \frac{(X \mapsto P,\ T_i)^P = \langle P_i', Bd^e \rangle \quad i \in I \quad id \text{ fresh}}{(\bar{X} \mapsto \bar{P}, p?\{m_i\langle \bar{S}^i \rangle \cdot T_i\}_{i\in I})^{P'} = \langle id() \cdot 0, \bigcup_{i\in I} \overline{Bd}^i \cup \{id() = \langle 0, ([p])?\{m_i\langle \bar{x}^i \rangle \cdot P_i'\}_{i\in I}\rangle\}\rangle}$$

$$\text{(TR-REC)}\ \frac{(\bar{X} \mapsto \bar{P} X \mapsto P'', T)^P = \langle P', \overline{Bd}\rangle \quad P'' = id().0 \quad id \text{ fresh}}{(\bar{X} \mapsto \bar{P}, \mu X.T)^P = \langle id().0, \overline{Bd} \cup \{id() = \langle P', \emptyset\rangle\}\rangle}$$

$$\text{(TR-VAR)}\ \frac{X \mapsto P \in \bar{X} \mapsto \bar{P}}{(\bar{X} \mapsto \bar{P}, X)^P = \langle P, \emptyset\rangle}$$

$$\text{(TR-MAP)}\ \frac{(\emptyset, T)^{P_e} = \langle P, \overline{Bd}\rangle \quad (\bar{X} \mapsto \bar{P},\ T')^0 = \langle P', \overline{Bd}'\rangle}{(\bar{X} \mapsto \bar{P}, \mathrm{map}\, x : \langle R, q\rangle T;\ T')^0 = \langle id().0, \bigcup_{1\le i\le n} \overline{Bd}^i \cup \overline{Bd}' \cup \{id'() = \langle P', \emptyset\rangle, id() = Bh_b\}\rangle}$$

where

$$P_e = \ \text{cnt}-.\text{if cnt==0} : id'().0 \text{ else } 0 \quad Bh_b = \langle \text{init}(cnt). \textstyle\sum_{1<i<n} id_i().0, \emptyset\rangle$$

$$\overline{Bd}^i = \overline{Bd}[rs[i]/x] \cup \{id_i() = \langle P[rs[i]/x], \emptyset\rangle\}$$

$$\text{(TR-END)}(\bar{X} \mapsto \bar{P},\ \text{End}\ )^P = \langle P, \emptyset\rangle$$

$$\text{(TR-QUIT)}(\bar{X} \mapsto \bar{P}, p\ ?\ \text{Quit .End})^{\ P} = \langle id().0, \{id() = \langle 0, (p)\ ?\ \text{Quit .rmv(rs, ([p])).}\ P\rangle\}\rangle.$$

FIGURE 4
Translation of a session type into a Jadescript agent.

replaced. The process returned from the translation activates all the defined behaviors. We use the summation just to indicate the sequence of activations. To control the end of execution, we also initialize the coordinator variable `cnt` to the number of agents in the role set. The process that will be executed at the end of the sub-protocol, P_e, decrements `cnt` and, if the counter is zero, then activates the execution of the behavior whose handler executes the process P' translating the type after the semicolon. To trigger the execution of the process P_e at the end of the execution of the sub-protocol `T`, we decorate its translation with it. Note that we expect the translation of `map` and the one of the type following the semicolon to be decorated with `0`, since we do not allow nested `map`. As we can see from Rule (TR-END), occurrences of `End` inside a `map` will be translated with P_e, producing the required result.

Finally, the translation of the acceptance of the message `Quit`, Rule (TR-QUIT), activates a behavior with a handler for the incoming message `Quit` that removes the agent from the role set and then executes the process associated with the end of the `map` (as for Rule (TR-END)).

Let *Lp* be the local protocol

`local protocol` *name* `at` $Z(\overline{\mathsf{p}}; \overline{\langle \mathsf{R}, \mathsf{q}\rangle}) = \mathsf{T}$

and $(\emptyset, \mathsf{T})^0 = \langle P, \overline{Bd}\rangle$. The *translation of Lp*, dubbed $(\!|Lp|\!)$, is the agent declaration

`agent` $(\!|Z|\!)$ `[property rs,cnt] on create` $P \qquad \overline{Bd}$

The process of the agent that will be executed at the creation of the agent (in the initial configuration) is the translation of the type `T`. The behaviors of the agent are the ones produced by the translation, and the fields `rs` and `cnt` are present if the participant/agent is a coordinator.

Definition 4.1. Let $\mathcal{R}_i$ be the set of initial participants of the role set R_i. The *initial configurations* of local protocols, $\mathcal{I}_t(\_)$, and the ones of their (agent) translations, $\mathcal{I}_a(\_)$, are defined by:

$$\mathcal{I}_t(\texttt{local protocol}\ \mathit{name}\ \texttt{at}\ Z(\overline{\mathsf{p}}; \overline{\langle \mathsf{R}, \mathsf{q}\rangle}) = \mathsf{T}) = \begin{cases} \mathsf{p}[\![\ \emptyset, \mathsf{T}\ ]\!] & \text{if } Z = \mathsf{p} \in \overline{\mathsf{p}} \\ \mathsf{q}_i[\![\ \emptyset, \mathcal{R}_i, \mathsf{T}\ ]\!] & \text{if } Z = \mathsf{q}_i \text{ and } \langle \mathsf{q}_i, \mathsf{R}_i\rangle \in \overline{\langle \mathsf{q}, \mathsf{R}\rangle} \\ \|_{\mathsf{r}\in\mathcal{R}_i}\ \mathsf{r}[\![\ \emptyset, \mathsf{T}\ ]\!] & \text{if } Z = \mathsf{R}_i \text{ and } \langle \mathsf{q}_i, \mathsf{R}_i\rangle \in \overline{\langle \mathsf{q}, \mathsf{R}\rangle} \end{cases}$$

and

$$\mathcal{I}_a(\texttt{agent}\ (\!|Z|\!)\ [\texttt{property rs,cnt}]\ \texttt{on create}\ P) = \begin{cases} (\!|\mathsf{p}|\!)\ [\![\ \emptyset, P, \emptyset\ ]\!] & \text{if } Z = \mathsf{p} \in \overline{\mathsf{p}} \\ (\!|\mathsf{q}_i|\!)\ [\![\ \emptyset, (Rs_i, 0), P, \emptyset\ ]\!] & \text{if } Z = \mathsf{q}_i \in \overline{\mathsf{q}} \\ \|_{\mathsf{r}\in\mathcal{R}_i}\ (\!|\mathsf{r}|\!)\ [\![\ \emptyset, P, \emptyset\ ]\!] & \text{if } Z = \mathsf{R}_i \end{cases}$$

where $Rs_i = \{(\!|\mathsf{p}|\!)\ |\ \mathsf{p} \in \mathcal{R}_i\}$

We now prove that the agent system obtained by translating the session types that are projections of a global type correctly executes the global protocol. We show *session fidelity*, i.e., the

communications of the agents match the ones of the session types projection of the global type. For agents, the labels of the reductions show, in addition to the interactions between agents, also the internal actions of agents. So we define a reduction that filters these "silent" actions and only shows the interactions between agents. We then use this reduction to match the one of the session types.

The reduction on agent configurations $\chi \overset{\ell}{\Rightarrow} \chi'$, where $\ell = a\dagger b.m$ and $\dagger =!$ or $\dagger =?$, is defined by

$$\chi \overset{\ell}{\Rightarrow} \chi' \quad \text{iff} \quad \chi \xrightarrow{\tau} \chi_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} \chi_n \xrightarrow{\ell} \chi$$

That is, any number of silent reductions followed by one of the actions (input or output).

Theorem 4.2 (Session Fidelity). Let $Lp_1,\ldots,Lp_n$ be the projections of

$$\texttt{global protocol}\ name(\overline{\mathsf{p}};\ \overline{\langle \mathsf{R}, \mathsf{p}\rangle}) = \mathsf{G}$$

on all its participants. Consider the initial configurations $\mathbb{N}_0 = \|_{1\le i\le n}\, \mathcal{I}_t(Lp_i)$ and $\chi_0 = \|_{1\le i\le n}\, \mathcal{I}_a(\llbracket Lp_i \rrbracket)$. For all $n \in \mathbb{N}$,

1. $\chi_0 \overset{\ell_1}{\Rightarrow} \chi_1 \overset{\ell_2}{\Rightarrow} \cdots \overset{\ell_n}{\Rightarrow} \chi_n$ implies $\mathbb{N}_0 \xrightarrow{\alpha_1} \mathbb{N}_2 \xrightarrow{\alpha_2} \mathbb{N}_2 \cdots \xrightarrow{\alpha_n} \mathbb{N}_n$
   where if $\ell_i = a\dagger b.m$, then $\alpha_i = \mathsf{p}\dagger\mathsf{q}.m$ and $a = (\!|\mathsf{p}|\!)$ and $b = (\!|\mathsf{q}|\!)$
2. $\mathbb{N}_n \xrightarrow{\mathsf{p}?\mathsf{q}.m} \mathbb{N}_{n+1}$ implies $\chi_n \xrightarrow{(\!|\mathsf{p}|\!)?(\!|\mathsf{q}|\!).m} \chi_{n+1}$ and
3. $\mathbb{N}_n \xrightarrow{\mathsf{p}!\mathsf{q}.m_h} \mathbb{N}_{n+1}$ implies $\chi_n \xrightarrow{(\!|\mathsf{p}|\!)!(\!|\mathsf{q}|\!).m_k} \chi_{n+1}$ where $m_h, m_k \in \{m_i\}_{i\in I}$ for some I.

Proof. The proof is by induction on n. For the proof, we consider for the agents an *extended agent configuration*, in which we add to each agent the set of behaviors defined by the translations. As in Giannini et al. (2006) we define a relation, $Rel(\mathbb{N}, \mathbb{A})$, between a type configuration and an extended agent configuration that, loosely speaking, expresses the fact that the agent configuration $\mathbb{A}$ implements the type configuration $\mathbb{N}$. We show that the initial configurations, $\mathbb{N}_0$ and $\mathbb{A}_0$, stand in the relation, i.e., $Rel(\mathbb{N}_0, \mathbb{A}_0)$ and that the relation is an invariant maintained by input and output reductions. This yields the result.

The theorem says that the interactions made by the agents follow the protocol specified by the session types, Item 1, and vice versa the interaction required by the protocol can be done by the agents, Items 2 and 3. When a session type requires to do a send chosen in a set $m_h \in \{m_i\}_{i\in I}$, Item 3, then the corresponding agent, $(\!|\mathsf{p}|\!)$, may send a different message, m_k, to $(\!|\mathsf{q}|\!)$. However, this message will be chosen in the same set. So from $\chi_n \xrightarrow{(\!|\mathsf{p}|\!)!(\!|\mathsf{q}|\!).m_k} \chi_{n+1}$ and Item 1, by Rule (SND), we also get $\mathbb{N}_n \xrightarrow{\mathsf{p}!\mathsf{q}.m_k} \mathbb{N}_{n+1}$. This is correct since the type $\mathsf{q}!\{m_i\langle\vec{S}^i\rangle.\mathsf{T}_i\}_{i\in I}$ of participant p only requires that the participant start by sending a message to q within the set $\{m_i\}_{i\in I}$.

The Supplemental material contains the definition of the relation $Rel(\mathbb{N}, \mathbb{A})$ and the proofs of the needed results.

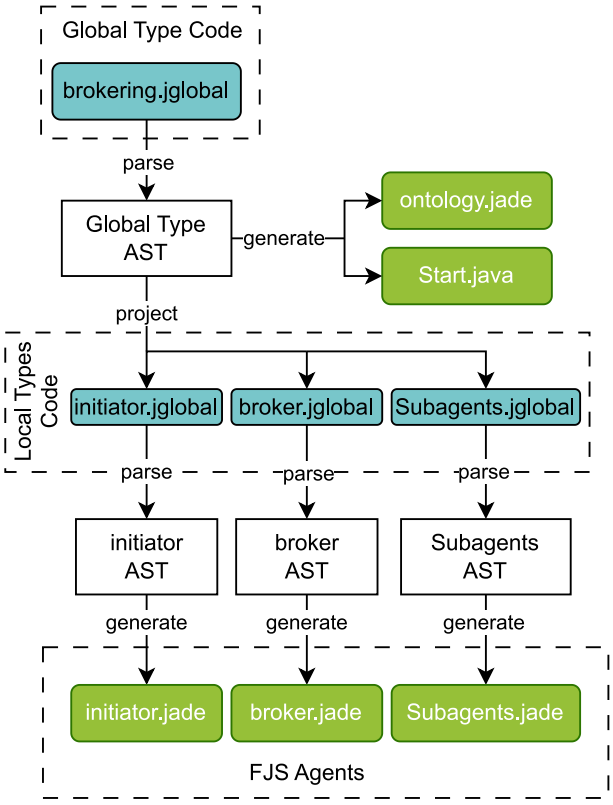

FIGURE 5
Xtext application workflow.

5 The toolchain: from global types to Jadescript agents

Jadescript is a scripting language designed to simplify the development of JADE agents. The agent behavior is specified by the definition of *behaviors*, which encapsulate their execution logic. Each behavior can be either *one-shot*, meaning it is executed only once, or *cyclic*, meaning it runs repeatedly until explicitly stopped.

The toolchain is implemented as an Xtext application that takes a global type and automatically generates the corresponding local types and FJS agents. Using Xtext, we defined the *Domain-Specific Language* (*DSL*) for both global and local types, and we employed its validation and generation modules to support the development of the translator.

The workflow of the Xtext application is illustrated in Figure 5. In the example shown, starting from the definition of the global type for the brokering interaction protocol, a set of files (highlighted in green) is generated. These files include the agent code, the shared ontology, and a Java file to start the interaction. The user-defined *.jglobal* global type is parsed to construct an *Abstract Syntax Tree* (*AST*). Validation is then performed on the AST to enforce properties that cannot be expressed syntactically, such as the correct format of the first message in a choice, the scope of variables in the `map` construct, and the proper use of role set identifiers. We assume that the file containing the initial global type also specifies the ontology, as this is required by Jadescript. From the AST, two

additional files are generated: *ontology.jade*, the shared ontology used by Jadescript agents for communication, and *Start.java*, a Java script that simplifies launching the multi-agent system.

If validation of the global type is successful, its AST is projected onto local types according to the rules in Definition Projection of global types onto participants, with each local type written into a separate file. This projection is implemented in the Xtext generation module, using a mix of Xtend and Java code. Projection may fail, since the *merge* operation, as defined in Yoshida and Gheri (2020), is not always guaranteed to succeed.

The local type files are then parsed to build their ASTs, followed by another validation phase. This additional validation is important because we want our tool to also support local types provided directly by the user, which are not necessarily generated by projection and therefore may not be correct by construction. If validation succeeds, the final FJS code is generated using Xtext's code generation tools.

The result is a collection of files that include the agents' internal code, their communication ontology, and the Java start file. At this point, the application workflow is complete, and the developer can proceed to integrate the logic and infrastructure required for protocol execution.

The generated agents contain not only code derived from their local protocol specifications but also additional initialization steps not explicitly defined in the protocol. This *initialization step* sets up variables that store the *Agent Identifiers* (*AIDs*) of other agents, since an agent must know the AID of its peers in order to communicate with them. For agents not in a role set, the AID can be retrieved directly from their name, which is hard-coded during code generation. For role sets, however, the number of agents is not known at generation time, so a different strategy is required: when an agent in a role set is created, it sends a *Hello* message to its coordinator. Upon receiving the message, the coordinator stores the agent's AID in a list. The coordinator stops waiting for additional agents after a predefined timeout. If the agent itself is a coordinator, it also instantiates and initializes map-related variables used to manage the `map` construct. This initialization code is implemented as an *on create* behavior, which is not part of FJS, whereas *on activate* has the same meaning in both frameworks.

In Jadescript, behaviors can be either *one-shot* or *cyclic*. A one-shot behavior executes its *on-execute* handler once and then checks its mailbox for a matching message. If such a message exists, the corresponding handler is executed; otherwise, the behavior is deactivated. Cyclic behaviors, by contrast, remain active after executing their *on-execute* handler, continuously monitoring the mailbox and attempting to match incoming messages to their handlers. These behaviors must be explicitly deactivated. Because message exchange is asynchronous, we cannot guarantee that a mailbox already contains a message when a behavior with a message handler is activated. To preserve the intended FJS semantics described in Section 3.1, our implementation relies on cyclic behaviors, explicitly deactivating them when the inactive process is reached.

The repository https://github.com/LMetal/Jadescript contains both the application implementation and the Eclipse plugin. A set of examples–including global types, projected local types, and the corresponding Jadescript agents generated by the translation–can be found in the top-level *Examples* directory. The implemented translation also performs some optimizations. For instance, it avoids generating code that first defines a behavior and then immediately activates it, which we included in the formal definition only to maintain uniformity and simplify proofs.

6 Related work and conclusions

While session types have not traditionally been employed within agent-oriented languages, their application in actor-based languages has been explored for some time. This interest stems from a key limitation of the actor model: it does not inherently guarantee the correct sequencing of interactions among concurrent processes. Consequently, ensuring proper coordination among actors–especially when systems scale–remains a complex and pressing challenge. Pioneering work by Mostrous and Vasconcelos (2011) introduced a session typing system for a minimal subset of Erlang, known as Featherweight Erlang. Their approach used session-typed references (similar to channels) to uniquely identify communication sessions, ensuring that processes adhered to the specified protocol behavior. However, this framework lacked tooling support for verifying Featherweight Erlang programs. Subsequent research has shifted toward runtime verification of actor interactions using MPSTs. Neykova and Yoshida (2017) presented a framework that generates runtime monitors from MPSTs. In their implementation, actors written in Python communicate using the *Advanced Message Queuing Protocol* (*AMQP*), thereby simulating actor-like behavior. In parallel, Fowler (2016) proposed an Erlang-based adaptation of this monitoring strategy. Both approaches make use of Scribble, a Java-based toolchain for specifying global communication protocols. Scribble enables the parsing, validation, and projection of local types from global MPST definitions. Nevertheless, these systems do not support static type checking to verify protocol conformance prior to execution. Building on these foundations, Harvey et al. (2021) introduced EnsembleS, an actor-oriented programming language that integrates MPSTs natively. EnsembleS enables compile-time verification of protocol adherence and supports safe, dynamic adaptation, particularly in the face of actor failures and recoveries. In Egidi et al. (2022), an Erlang implementation for multiparty protocols using a reduced yet significant set of Erlang constructs (FErlang), including constructs for delegation, was presented. A tool-chain including projection from global types on session types and a type checker for FErlang programs was implemented. The implementation was done using the meta compiler JastAdd, resulting in a system that could be easily extended to include new syntactic constructs. More recently, Tabone and Francalanza (2021, 2022) added (binary) session types to Elixir, and Francalanza and Tabone (2023) established a result of session fidelity between this session type system and the runtime behavior of a client handler.

Beyond the Erlang ecosystem, session types have also been applied to the Scala programming language. Scalas and Yoshida (2016) encoded binary session types as Scala classes, utilizing the compiler to enforce session fidelity, and then Scalas et al. (2019b,a) advanced this line of research by incorporating session types into Scala 3, employing dependent function types and model checking techniques to achieve compile-time protocol verification. Burlò et al. (2021) extended this approach by verifying one side of a

communication protocol statically while using runtime monitors for the other. Finally, Hähnle et al. (2020) extended the active object language ABS — which uses futures for handling method results — by incorporating session types and global protocols from MPSTs to define the sequence of method invocations across objects.

Our approach is different; instead of adding a session type system to the language and then type checking the system, we enforce the good properties of well-typed programs by translating the protocol into a program, i.e., in our case a set of agent definitions. By proving that our translation preserves the intended semantics of the session types, we obtain a program that is correct by construction. In the article, we focus on the typical interactions present in the FIPA interaction protocols that involve the presence of some participants and some groups of participants interacting with a coordinator. To express these protocols, we defined an extension of the standard global and session types of Honda et al. (2008). In particular, we added the notion of "role sets", inspired by Cledou et al. (2022), and its "coordinator", coupled with a construct synchronizing the interaction of a coordinator with all the participants in a role set. Moreover, a member of a role set may decide to exit from the protocol by informing its coordinator. We defined the projection of these new global types on the individual participants/agents and on role sets, and then the translation of the session types into Jadescript agents. We proved that the defined translation preserves the semantics of the session types obtained as projection from the global type. To this end, we defined a core agent language FJS incorporating the features of Jadescript needed for the translation with its operational semantics. We also implemented a plugin for Eclipse, in Xtext, that provides an editor for global and session types and the projection and translation.

This work is part of a larger research project within the Italian project *Typefull Language Adaptation for Dynamic, Interacting and Evolving Systems* (*T-Ladies*), one of whose goals is to provide support for development/maintenance, automatic property verification/enforcement, and bug detection of loosely connected, distributed, possibly heterogeneous interacting systems.

As future work, we plan to make the implementation of the translation more efficient. The current translation was defined to support its proof of correctness, but some behaviors we define (and then activate) can be avoided. Moreover, we are undertaking a similar translation for Elixir, which is an actor language with a wider distribution and a different audience than Jadescript.

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

FB: Writing – original draft, Writing – review & editing. LE: Writing – original draft, Writing – review & editing. LG: Writing – original draft, Writing – review & editing. PG: Writing – original draft, Writing – review & editing. SM: Writing – original draft, Writing – review & editing.

Funding

The author(s) declare that financial support was received for the research and/or publication of this article. This work was partially supported by the Italian Ministry of University and Research under the PRIN 2020 grant 2020TL3X8X for the project Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems (T-LADIES) and has the financial support of the Università del Piemonte Orientale.

Acknowledgments

The authors thank the referees for their constructive remarks, which helped improve the submitted version of the paper. We also thank Riccardo Nazzari for his participation in the initial phases of the implementation of the Xtext prototype.

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 Gen 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

Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fcomp.2025.1659785/full#supplementary-material

References

Bădică, C., Budimac, Z., Burkhard, H.-D., and Ivanovic, M. (2011). Software agents: languages, tools, platforms. *Comput. Sci. Inf. Syst.* 8, 255–298. doi: 10.2298/CSIS110214013B

Bergenti, F., Caire, G., Monica, S., and Poggi, A. (2020). The first twenty years of agent-based software development with JADE. *Auton. Agents Multi-Agent Syst.* 34:36. doi: 10.1007/s10458-020-09460-z

Bergenti, F., Gleizes, M.-P., and Zambonelli, F. eds. (2004). *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook* (Springer: New York). doi: 10.1007/b116049

Bergenti, F., Monica, S., and Petrosino, G. (2018). "A scripting language for practical agent-oriented programming," in *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2018) at ACM SIGPLAN Conference Systems, Programming, Languages and Applications: Software for Humanity (SPLASH 2018)* (New York, NY: ACM), 62–71.

Bergenti, F., and Petrosino, G. (2018). "Overview of a scripting language for JADE-based multi-agent systems," in *Proceedings of the 19th Workshop "From Objects to Agents" (WOA 2018) (RWTH Aachen), vol. 2215 of CEUR Workshop Proceedings* (Aachen: CEUR), 57–62.

Bergenti, F., and Ricci, A. (2002). "Three approaches to the coordination of multiagent systems," in *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing* (New York, NY: ACM), 367–372. doi: 10.1145/508791.508861

Bordini, R. H., Braubach, L., Dastani, M., Seghrouchni, A. E. F., Gomez-Sanz, J. J., Leite, J., et al. (2006). A survey of programming languages and platforms for multi-agent systems. *Informatica* 30, 33–44. Available online at: https://www.scopus.com/inward/record.uri?eid=2-s2.0-33144455844&partnerID=40&md5=24c8e4df8fdee9138cfd729c2489e2aa

Bordini, R. H., Hubner, J. F., and Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak Using Jason* (Hoboken, NJ: Wiley). doi: 10.1002/9780470061848

Burlò, C. B., Francalanza, A., and Scalas, A. (2021). "On the monitorability of session types, in theory and practice," in *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11–17, 2021, Aarhus, Denmark (Virtual Conference)*, eds. A. Møller and M. Sridharan (Schloss Dagstuhl - Leibniz-Zentrum für Informatik).

Castro-Perez, D., Hu, R., Jongmans, S., Ng, N., and Yoshida, N. (2019). Distributed programming using role-parametric session types in go: statically-typed endpoint APIS for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.* 29, 1–30. doi: 10.1145/3290342

Cledou, G., Edixhoven, L., Jongmans, S., and Proença, J. (2022). "API generation for multiparty session types, revisited and revised using Scala 3," in *ECOOP 2022*, eds. K. Ali and J. Vitek (Schloss Dagstuhl - Leibniz-Zentrum für Informatik), 1–28.

Dagnino, F., Giannini, P., and Dezani-Ciancaglini, M. (2023). Deconfined global types for asynchronous sessions. *Log. Methods Comput. Sci.* 19, 1–41. doi: 10.46298/lmcs-19(1:3)2023

Deniélou, P., and Yoshida, N. (2011). "Dynamic multirole session types," in *POPL*, eds. T. Ball and M. Sagiv (ACM), 435–446. doi: 10.1145/1926385.1926435

Deniélou, P., Yoshida, N., Bejleri, A., and Hu, R. (2012). Parameterised multiparty session types. *Log. Methods Comput. Sci.* 8:4. doi: 10.2168/LMCS-8(4:6)2012

Egidi, L., Giannini, P., and Ventura, L. (2022). "Multiparty-session-types coordination for core erlang," in *Proceedings of the 17th International Conference on Software Technologies, ICSOFT 2022, Lisbon, Portugal, July 11–13, 2022*, eds. H. Fill, M. van Sinderen, and L. A. Maciaszek (Setúbal: SCITEPRESS), 532–541. doi: 10.5220/0011316600003266

Fowler, S. (2016). "An erlang implementation of multiparty session actors," in *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8–9 June 2016*, eds. M. Bartoletti, L. Henrio, S. Knight, and H. T. Vieira (EPTCS), 36–50. doi: 10.4204/EPTCS.223.3

Francalanza, A., and Tabone, G. (2023). Elixirst: a session-based type system for elixir modules. *J. Log. Algebraic Methods Program.* 135:100891. doi: 10.1016/j.jlamp.2023.100891

Giannini, P., Sangiorgi, D., and Valente, A. (2006). Safe ambients: abstract machine and distributed implementation. *Sci. Comput. Program.* 59, 209–249. doi: 10.1016/j.scico.2005.05.002

Hähnle, R., Haubner, A. W., and Kamburjan, E. (2020). "Locally static, globally dynamic session types for active objects," in *Recent Developments in the Design and Implementation of Programming Languages, Gabbrielli's Festschrift, November 27, 2020, Bologna, Italy*, eds. F. S. de Boer and J. Mauro (Schloss Dagstuhl - Leibniz-Zentrum für Informatik), 1–24.

Harvey, P., Fowler, S., Dardha, O., and Gay, S. J. (2021). "Multiparty session types for safe runtime adaptation in an actor language," in *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11–17, 2021, Aarhus, Denmark (Virtual Conference)*, eds. A. Møller and M. Sridharan (Schloss Dagstuhl - Leibniz-Zentrum für Informatik), vol. 194 of LIPIcs, 10–30.

Honda, K., Mukhamedov, A., Brown, G., Chen, T., and Yoshida, N. (2011). "Scribbling interactions with a formal foundation," in *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9–12, 2011. Proceedings*, eds. R. Natarajan and A. K. Ojo (Springer: New York), 55–75. doi: 10.1007/978-3-642-19056-8_4

Honda, K., Yoshida, N., and Carbone, M. (2008). "Multiparty asynchronous session types," in *POPL*, eds. G. C. Necula and P. Wadler (New York: ACM Press), 273–284. doi: 10.1145/1328438.1328472

Honda, K., Yoshida, N., and Carbone, M. (2016). Multiparty asynchronous session types. *J. ACM* 63, 1–67. doi: 10.1145/2827695

Hüttel, H., Lanese, I., Vasconcelos, V. T., Caires, L., Carbone, M., Deniélou, P.-M., et al. (2016). Foundations of session types and behavioural contracts. *ACM Comput. Surv.* 49, 1–36. doi: 10.1145/2873052

Lagaillardie, N., Neykova, R., and Yoshida, N. (2022). "Stay safe under panic: affine rust programming with multiparty session types," in *ECOOP 2022*, eds. K. Ali and J. Vitek (Schloss Dagstuhl - Leibniz-Zentrum für Informatik), 4–29.

Mostrous, D., and Vasconcelos, V. T. (2011). "Session typing for a featherweight erlang," in *Coordination* (Springer: New York), 95–109. doi: 10.1007/978-3-642-21464-6_7

Neykova, R., and Yoshida, N. (2017). Multiparty session actors. *Log. Methods Comput. Sci.* 13:1. doi: 10.23638/LMCS-13(1:17)2017

Ng, N., and Yoshida, N. (2014). "Pabble: parameterised scribble for parallel programming," in *PDP 2014* (Washington, DC: IEEE Computer Society), 707–714. doi: 10.1109/PDP.2014.20

Ng, N., and Yoshida, N. (2015). Pabble: parameterised scribble. *Serv. Oriented Comput. Appl.* 9, 269–284. doi: 10.1007/s11761-014-0172-8

Petrosino, G., and Bergenti, F. (2018). "An introduction to the major features of a scripting language for JADE agents," in *Proceedings of the 17th Conference of the Italian Association for Artificial Intelligence (AI*IA 2018)* (Springer: New York), 3–14. doi: 10.1007/978-3-030-03840-3_1

Petrosino, G., Iotti, E., Monica, S., and Bergenti, F. (2021). "Prototypes of productivity tools for the jadescript programming language," in *Proceedings of the 22nd Workshop "From Objects to Agents" (WOA 2021)* (Aachen: RWTH Aachen), 14–28.

Petrosino, G., Iotti, E., Monica, S., and Bergenti, F. (2022). "A description of the jadescript type system," in *Proceedings of the 3rd International Conference on Distributed Artificial Intelligence (DAI 2022)* (Springer: New York), 206–220. doi: 10.1007/978-3-030-94662-3_13

Poslad, S. (2007). Specifying protocols for multi-agent system interaction. *ACM Trans. Auton. Adapt. Syst.* 2, 15–24. doi: 10.1145/1293731.1293735

Scalas, A., and Yoshida, N. (2016). Lightweight session programming in scala (artifact). *Dagstuhl Artifacts Ser.* 2, 11:1–11:2.

Scalas, A., Yoshida, N., and Benussi, E. (2019a). "Effpi: verified message-passing programs in dotty," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala, Scala@ECOOP 2019, London, UK, July 17, 2019*, eds. J. I. Brachthäuser, S. Ryu, and N. Nystrom (ACM), 27–31. doi: 10.1145/3337932.3338812

Scalas, A., Yoshida, N., and Benussi, E. (2019b). "Verifying message-passing programs with dependent behavioural types," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, eds. K. S. McKinley and K. Fisher (New York, NY: ACM), 502–516. doi: 10.1145/3314221.3322484

Shoham, Y. (1991). "AGENT0: a simple agent language and its interpreter," in *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI 1991)* (New York, NY: ACM), 704–709

Shoham, Y. (1993). Agent-oriented programming. *Artif. Intell.* 60, 51–92. doi: 10.1016/0004-3702(93)90034-9

Shoham, Y., and Leyton-Brown, K. (2008). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations* (Cambridge: Cambridge University Press). doi: 10.1017/CBO9780511811654

Tabone, G., and Francalanza, A. (2021). "Session types in elixir," in *AGERE 2021: Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, Virtual Event / Chicago, IL, USA, 17 October 2021*, eds. E. Castegren, J. D. Koster, and S. Fowler (New York, NY: ACM), 12–23. doi: 10.1145/3486601.3486708

Tabone, G., and Francalanza, A. (2022). "Session fidelity for elixirst: a session-based type system for elixir modules," in *Proceedings 15th Interaction and Concurrency Experience, ICE 2022, Lucca, Italy, 17th June 2022*, eds. C. Aubert, C. D. Giusto, L. Safina, and A. Scalas (EPTCS), 17–36. doi: 10.4204/EPTCS.365.2

Tomaiuolo, M., Turci, P., Bergenti, F., and Poggi, A. (2006). "An ontology support for semantic aware agents," in *Proceedings of the 7th International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS 2005)* (Springer: New York), 140–153. doi: 10.1007/11916291_10

Viering, M., Hu, R., Eugster, P., and Ziarek, L. (2021). A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proc. ACM Program. Lang.* 5, 1–30. doi: 10.1145/3485501

Yokoo, M. (2001). *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-agent Systems*. Springer Series on Agent Technology (Springer): New York. doi: 10.1007/978-3-642-59546-2

Yoshida, N., and Gheri, L. (2020). "A very gentle introduction to multiparty session types," in *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings*, eds. D. V. Hung and M. D'Souza (Springer: New York), 73–93. doi: 10.1007/978-3-030-36987-3_5