frontiers | Frontiers in Computer Science

TYPE Original Research
PUBLISHED 16 January 2026
DOI 10.3389/fcomp.2025.1723480

Check for updates

OPEN ACCESS

EDITED BY
Agostino Cortesi,
Ca' Foscari University of Venice, Italy

REVIEWED BY
Beyza Eken,
Sakarya University, Türkiye
Mathias Kleiner,
ParisTech École Nationale Supérieure d'Arts et Métiers, France

*CORRESPONDENCE
Behrouz Tork Ladani
✉ ladani@eng.ui.ac.ir

RECEIVED 12 October 2025
REVISED 12 December 2025
ACCEPTED 18 December 2025
PUBLISHED 16 January 2026

CITATION
Alsharuee HAN, Sharbaf M and Tork Ladani B (2026) A configurable approach for intra-model inconsistency management in multi-view collaborative modeling. *Front. Comput. Sci.* 7:1723480. doi: 10.3389/fcomp.2025.1723480

A configurable approach for intra-model inconsistency management in multi-view collaborative modeling

Hayder Ali Neamah Alsharuee [1,2], Mohammadreza Sharbaf [1] and Behrouz Tork Ladani [1*]

[1]Department of Software Engineering, University of Isfahan, Isfahan, Iran, [2]Department of Scholarships and Cultural Relations, Ministry of Higher Education and Scientific Research, Baghdad, Iraq

Introduction: In the software development life cycle, collaborative modeling through multiple projective views of a single, shared model is a critical activity that enables effective collaboration among experts and stakeholders. Real-time optimistic collaboration in multi-view modeling allows concurrent modifications but often introduces inconsistencies that must be resolved to achieve an integrated and valid model. Existing inconsistency management methods frequently focus on isolated repairs or offer limited alternatives, lacking support for collaborative dynamics and configurable resolution strategies. This study aims to develop a configurable framework for managing intra-model inconsistencies in real-time multi-view collaborative modeling environments.

Methods: We propose a novel framework for inconsistency management tailored to multi-view collaborative modeling, based on Model-Driven Engineering (MDE) principles. The framework supports real-time modeling scenarios and enables change propagation according to the online collaboration mode. Key components include a consistency oracle and incremental consistency checking, which together manage the integration of model changes and overlaps. We introduce the COMIM approach, which assists collaborators in handling inconsistencies by considering team interactions, individual ownership, and configurable repair strategies.

Results: The framework was evaluated through a case study involving multi-view collaborative modeling sessions. Empirical results demonstrate the feasibility and effectiveness of the COMIM approach in maintaining consistency during concurrent modeling activities. The system performed efficiently for teams of up to seven concurrent users, successfully managing change propagation, detecting inconsistencies incrementally, and supporting configurable resolution aligned with collaborative priorities.

Discussion: The proposed framework effectively addresses the complexities of repairing inconsistencies across diverse software models in a collaborative setting. By emphasizing collaborative dynamics, our approach advances traditional inconsistency management methods, which often lack personalization and configurability. Future work may explore scalability to larger teams and adaptation to additional modeling paradigms.

KEYWORDS

collaborative modeling, intra-model inconsistency management, model-driven engineering, multi-view collaboration, multi-view modeling

1 Introduction

Software engineering is a collaborative discipline where engineers work concurrently on a variety of engineering artifacts, such as requirements, use cases, designs, and code. To modify these artifacts, engineers utilize a diverse landscape of specialized tools—each tool typically focuses on specific types of artifacts (e.g., IDEs for source code, modeling tools for UML). As a result, each tool provides a partial view of the software system, as engineers generally do not need access to all engineering artifacts. This kind of distributed and concurrent modification of artifacts by multiple engineers follows a familiar pattern: typically, engineers download artifacts from a shared repository, modify them independently on their local workstations, and then upload the changes back to the repository. This nature of coordination can lead to the introduction of inconsistencies, where artifacts contradict one another. It is particularly challenging to detect such inconsistencies when different engineers modify interdependent artifacts using different modifications.

To develop modern systems in the engineering domain, models are used as primary artifacts (Whittle et al., 2013). When the software is complex, the size and complexity of models increase dramatically. This enforces that many engineers and stakeholders participate in large teams and collaborate to evolve models (Franzago et al., 2017). In this context, some of the participants may work concurrently and independently on the same model from different geographical sites. Each participant focuses on specific aspects of the system and locally modifies only a particular part of the model.

State of the art in model-based software development has primarily focused on detecting inconsistencies between software models. Many effective approaches have been developed to quickly and accurately identify inconsistencies across different views or perspectives of a system (Blanc et al., 2008; Egyed, 2006; Yu and Choi, 2023; Grundy et al., 1998; Nentwich et al., 2002). While it is important to tolerate inconsistencies during the modeling process (Balzer, 1991), they must eventually be resolved. However, detecting and repairing inconsistencies in multi-view modeling is a highly challenging problem, as it requires comprehensive coverage of overlapping elements and semantic rules. The repair process itself introduces additional complexity. The number of possible repair alternatives grows exponentially with the number of model elements and views involved (Reder and Egyed, 2012a), overwhelming developers with a vast set of options. Existing approaches either ignore this exponential growth (Nentwich et al., 2003) or focus only on a limited set of repairs (Dam and Winikoff, 2011; Puissant et al., 2010). Other approaches impose restrictions on the consistency language or address individual inconsistencies in isolation, without considering the implications across multiple views (Egyed et al., 2008; Nentwich et al., 2003; Xiong et al., 2009). These limitations are problematic because developers may struggle to select the most appropriate repairs from the large solution space, or the available repairs may not adequately address the inconsistencies across the different views of the system.

Crucially, the process of repairing inconsistencies in multi-view modeling is tightly coupled with the creative modeling work performed by developers. We argue against heuristic-based approaches that attempt to automate the repair process, as these may fail to capture the nuanced reasoning and design decisions of the developer. For example, a repair that minimizes the number of model changes may be undesirable if it favors undoing a change that was the root cause of the inconsistency (Ohrndorf et al., 2021). Furthermore, the literature has not adequately addressed the challenge of repairing inconsistencies in collaborative multi-view software engineering environments. As software development is often a team-based discipline, the repair process should consider the collaborative aspects of model maintenance across different views (Mistrík et al., 2010). Collaboration can be leveraged to filter out infeasible repair alternatives, such as by prioritizing repairs that only impact the artifacts owned by the current developer or that are deemed acceptable by the team. However, there are no existing repair approaches that explicitly consider the collaborative nature of multi-view software engineering (Macedo et al., 2016; Torres et al., 2021).

While many approaches for consistency checking exist, they often focus on checking artifacts within the local environment of a single engineering tool. These approaches rarely operate on the merged set of all engineering artifacts in the global environment, such as a shared repository. The modifications made by engineers in their local environments may have implications on the overall consistency of the system, but these implications cannot be fully considered until the local changes are first merged into the global environment. This is an important research challenge, as the typical engineer's workflow involves first storing modifications locally, before potentially introducing inconsistencies that affect the overall system. In this context, the scope of this paper is intra-model inconsistency management. We address inconsistencies that arise when collaborators concurrently edit different views (e.g., structural, behavioral) of the same underlying model. This is distinct from inter-model consistency, which deals with relationships between separate, heterogeneous models.

In this paper, we present the Collaborative Multi-view modeling Inconsistency Management (COMIM) framework for managing inconsistencies in real-time collaborative modeling of a single model through multiple projective views. COMIM integrates a consistency oracle for real-time inconsistency detection and resolution, and a personalized view definition mechanism to propagate change operations, supporting collaborative modeling. COMIM is designed to handle the complexities of multi-view modeling by enabling fine-grained consistency checks, reordering operations to resolve conflicts, and providing feedback to collaborators. Building upon our prior work on conflict management [CoMPers (Sharbaf et al., 2025)], COMIM makes several novel contributions specific to the multi-view, projective modeling context: (1) a dedicated inconsistency management architecture centered on a scope-based consistency oracle; (2) formal definitions for operations and consistency in multi-view collaboration; (3) preference-based view personalization; and (4) integrated operation reordering for inconsistency resolution.

The framework is evaluated through case studies and benchmarks, demonstrating its effectiveness in maintaining consistency across multiple views, while minimizing overhead. By addressing the challenges of inconsistency management in collaborative environments, COMIM provides a scalable and configurable solution for real-time multi-view modeling. To show the applicability of the proposed approach, we provide a proof-of-concept implementation of COMIM and demonstrate it on the Wind Turbine case study, which showcases its ability to handle real-time collaborative multi-view modeling scenarios. Additionally, we utilize the consistency management benchmark to evaluate the performance of the consistency oracle and inconsistency resolution components in our approach. Furthermore, we investigate the scalability of COMIM by analyzing its computational complexity. The results demonstrate that COMIM is capable of supporting multiple modeling languages, although model size has a noticeable, yet relatively small, impact on evaluation time.

The rest of this paper is organized as follows: Section 2 provides the background foundations on collaborative modeling and conflict management. Section 3 reviews related works in the field. Section 4 describes the overall structure of the COMIM approach, including the conceptual overview of the collaboration architecture, the definition of model views based on user intents and preferences, and the inconsistency management infrastructure, which includes the consistency oracle and inconsistency resolution mechanisms. Finally, it demonstrates the proposed approach using an illustrative example. Section 5 presents the evaluation of our approach, including case studies, benchmarks, and scalability analysis. Finally, Section 6 concludes the paper and highlights directions for the future work.

2 Background

This section provides explanations and definitions of key terms used throughout the paper.

2.1 Models and views

In our approach, we draw upon concepts of views and viewtypes as defined in ISO/IEC/IEEE 42010:2011 (Júnior et al., 2019), which are commonly used in Model-Based Systems Engineering (MBSE). Specifically, we define the following terms:

- Model: a model is an abstraction that serves a specific purpose related to an original entity.
- Metamodel: a metamodel specifies the structure and constraints of models, detailing which elements, attributes, and relationships can exist within a model.
- View: a view presents specific parts of a model, allowing for a particular perspective on the overall model. It contains only the information necessary for that perspective and is an instance of a viewtype.
- Viewtype: a viewtype represents the portions of a metamodel that specify what is permissible within its associated views.

2.2 Consistency

In modeling, consistency refers to the property of a model or system to maintain logical coherence and reliability across different scenarios, inputs, or conditions. It ensures that the model behaves predictably and produces results that align with established rules, assumptions, or expectations. Lucas et al. (2009) define consistency as "a state in which two or more elements, which overlap in different models of the same system, have a satisfactory joint description." Herzig et al. (2014) express a similar concept, stating that "inconsistency is present if two or more statements are made that are not jointly satisfiable." A more formal definition offered by David et al. (2023) translates to "two design artifacts are said to be consistent with respect to a set of properties if they satisfy exactly the same properties of the set." From a logical perspective, Persson et al. (2013) describe consistency in the context of views as not having contradictions.

We concur that the fundamental understanding of consistency between models is the absence of contradictions when these models are considered together. From our perspective, consistency between models relies on their overlaps. If models share redundant information that must be maintained in a specific form, a consistency relation exists between them. To formalize this relationship, we define a machine-readable and evaluable term called *consistency rule*. If a consistency rule is evaluated as satisfied within a given model context, the associated consistency relation (i.e., the overlap) is deemed consistent; otherwise, it is not.

In summary, two models are considered consistent if all defined consistency rules for them are satisfied. If any rule is not satisfied, the models are deemed inconsistent. Inconsistencies may arise from syntactic conflicts (e.g., concurrent edits to an attribute's type) or semantic violations (e.g., violating inheritance hierarchies) (Sharbaf et al., 2020). This foundation supports rule-based consistency checking approaches, wherein consistency rules are typically defined as logical statements, such as Object Constraint Language (OCL) rules (Cabot and Gogolla, 2012). These rules can be evaluated by a rule checker within a specific model context, yielding a result of either true or false. If all rules evaluate to true, the model context is consistent; if any rule evaluates to false, the context is inconsistent.

3 Related work

In this section, we review the existing literature relevant to consistency management, particularly in collaborative modeling environments. Research in this area can be categorized by its focus: (1) inter-model inconsistency management, which addresses consistency between separate models, and (2) intra-model inconsistency management, which addresses consistency within a single model. Our work, COMIM, belongs to the latter category with a specific focus on multi-view collaborative modeling.

Numerous surveys provide foundational insights into consistency management, covering general approaches (Spanoudakis and Zisman, 2001), cross-domain consistency (Burger, 2014), behavioral model consistency (Muram et al., 2017), and conflict management techniques for collaborative modeling (Sharbaf et al., 2023).

Inter-model inconsistency management addresses consistency between multiple, potentially heterogeneous models that are related through transformations or constraints. For instance, Feldmann et al. (2019) propose a framework for specifying, diagnosing, and handling inter-model inconsistencies in production systems engineering. Stevens (2020) explores the use of bidirectional transformations to maintain consistency in networks of models. Kleiner and Roucoules (2025) present a framework for dynamic consistency management in multi-physical systems. While these approaches are effective for managing consistency across separate models, they are not designed for real-time, collaborative editing of a single model through multiple projective views.

Intra-model inconsistency management, in contrast, focuses on consistency within a single model. Some studies (e.g., Schröpfer et al., 2019; Sharbaf et al., 2025) address inconsistencies within models but do not explicitly consider multiple views. Our work, COMIM, focuses specifically on intra-model inconsistency management in a multi-view setting, where inconsistencies arise from concurrent modifications to different projective views of the same underlying model. COMIM's configurable, scope-based inconsistency checking is tailored to this context, providing real-time detection and resolution for collaborative modeling sessions.

In the following, we discuss related work that has directly influenced our approach and provides the broader context for multi-view collaborative modeling.

Cicchetti et al. (2011) review on multi-view modeling approaches complements existing surveys by focusing on the technical characteristics and consistency management of multi-view solutions, contrasting with prior works that often address narrower aspects such as UML diagram consistency or business process modeling. The findings highlight gaps in terminology standardization, semantic consistency, and real-world validation, aligning with broader challenges identified in granularity of views.

Knapp and Mossakowski (2018) surveid and classified existing techniques on multi-view consistency in UML and proposed a distributed semantics framework using OMG's DOL to formally link UML/OCL sub-languages while preserving their distinct semantics. According to their results, ensuring consistency across multiple UML/OCL views is challenging due to the language's heterogeneity, with most existing methods limited to structural checks and partial diagram coverage.

Wen et al. (2023) proposed a formal framework using a Structure Model to represent UML artifacts and their relationships, enabling automated consistency checking and repair. It classified updates (e.g., repairs, propagations) and introduces strategies (e.g., stable-change, least-change) to restore or preserve consistency. A prototype tool, FSMS, demonstrates the approach's feasibility. However, the framework relies heavily on manual rule selection and repair decisions, which may not scale efficiently for large systems. Additionally, the tool's current implementation lacks full automation for complex behavioral consistency checks.

Vogel-Heuser and Zou (2019) presented a knowledge-based approach for managing inconsistencies in multi-view collaborative modeling of Cyber-Physical Production Systems (CPPS), addressing challenges in cross-disciplinary engineering where heterogeneous models (e.g., SysML, Simulink) often lead to semantic and functional inconsistencies. By leveraging a domain ontology and metamodels, the authors formalize dependencies and enable automated inconsistency detection through reasoning in a Web Ontology Language (OWL)-based knowledge base, reducing manual effort and improving interoperability. The approach is validated on a bench-scale CPPS demonstrator, highlighting its ability to detect static semantic inconsistencies, though limitations remain in handling dynamic behaviors and resolution of inconsistencies as well as scalability for large-scale systems.

Karagiannis et al. (2016) introduced a semantic query-based approach for managing consistency in multi-view enterprise models by leveraging Linked Data technologies to transform, synchronize, and validate heterogeneous views. The authors introduce a framework where enterprise models—spanning behavioral, structural, and procedural facets—are serialized as RDF graphs, enabling consistency checks and transformations via SPARQL queries. The approach is validated through case studies from the ComVantage and SOM projects, demonstrating its utility in maintaining cross-view dependencies and supporting knowledge-driven systems. While the method enhances semantic interoperability and automation, the inconsistency resolution is not considered and the authors note challenges in usability and the slow adoption of Linked Data in enterprise settings.

Shahvari et al. (2024) proposed an inconsistency tolerance framework to support agile development in multi-view modeling environments, addressing the challenges of concurrent collaboration across diverse stakeholder views (e.g., warehouse managers, operators, maintenance teams). The framework introduces a three-phase process—detection, analysis, and tolerance—to classify inconsistencies (syntactic, structural, semantic) and apply context-aware strategies (constraint-based, temporal, spatial) for managing conflicts without halting development. Unlike traditional approaches that enforce immediate resolution, this work emphasizes flexible tolerance mechanisms aligned with agile principles, enabling iterative refinement while maintaining progress. While the conceptual framework is well-defined, it lacks validation through real-world case studies or industrial demonstrations, limiting insights into its practical effectiveness.

Cicchetti et al. (2011) presented a hybrid approach for incremental synchronization in multi-view modeling, combining the benefits of synthetic (distinct meta-models) and projective (single meta-model with views) techniques to address consistency management and customization challenges. The proposed framework leverages model differencing and higher-order transformations to detect and propagate changes incrementally across views, reducing computational overhead compared to full model recomputation. While the approach demonstrates technical feasibility through EMF-based tooling and a school management case study, it lacks empirical validation in industrial settings or large-scale deployments. Key limitations include basic inconsistency resolution and manual workflow coordination.

However, due to the diversity of modeling languages and view definitions, a customizable solution for consistency management is essential. By "configurable," we mean a framework where development teams can tailor the inconsistency management process along multiple dimensions: (a) defining the specific model elements (scopes) to monitor, (b) specifying custom consistency rules relevant to their domain, and (c) personalizing views for different stakeholders. Such a solution would enable teams to flexibly define and enforce consistency rules tailored to their collaborative needs. Yet, none of the existing approaches directly support this level of configurable inconsistency checking in multi-view modeling, leaving a critical gap in the field.

4 The COMIM approach

In real-time multi-view collaborative modeling, managing inconsistencies is a critical challenge due to the dynamic nature of collaborative environments where multiple users interact with the system simultaneously. In this section, we present COMIM as a comprehensive approach to address this challenge by introducing a structured architecture and an inconsistency management infrastructure. The architecture facilitates the propagation of change operations applied by different users across various views through a collaboration server. This ensures that changes are synchronized and reflected in real-time across all relevant views.

The proposed inconsistency management infrastructure is built on three layers: the metamodel layer, the model layer, and the view layer. Users can define scopes based on metamodel elements and specify inconsistency detection rules for each scope. Each view can be associated with one or more scopes, and the system checks the related inconsistency rules in the consistency oracle for every change applied in a view. Detected inconsistencies are resolved through mechanisms such as reordering operations or sending feedback to users. This approach ensures that the system maintains consistency across all views, even in a highly dynamic and collaborative environment.

While the examples in this paper use UML for clarity and familiarity, the COMIM framework is built on MDE principles and is modeling-language agnostic. Its core infrastructure is Ecore-based, meaning it operates on the fundamental concepts of model elements (EObject), properties (EAttribute), and relationships (EReference). The key components (i.e., Scope Definition, Consistency Oracle, and Operation Tree) are entirely independent of any specific modeling notation. The framework can be instantiated for any domain-specific language defined as an Ecore metamodel. The UML examples serve only to illustrate the approach in a widely understood context; they do not represent a limitation of the frameworks applicability.

The proposed framework supports personalization at two levels. First, at the individual level, collaborators can define personalized views of the model through the mechanisms described in Section 4.3, ensuring that each user interacts only with the model elements relevant to their tasks. Second, at the team or project level, the inconsistency management process is configurable: teams can define custom consistency rules, specify scopes for checking, and choose resolution strategies that fit their collaborative workflow.

In the following, we first present the formal foundations of the COMIM framework. Then, we introduce the real-time multi-view collaboration architecture. Next, we focus on the approach to define projective views. After that, we present an overview of the proposed infrastructure for inconsistency management by explanation of its components. Finally, we demonstrate the integrated approach through an illustrative example.

4.1 Formal foundations of COMIM

To establish a rigorous foundation for the COMIM approach, we formally define the core concepts and structures that underpin our inconsistency management framework. The COMIM framework provides two primary operations, as well as two key properties for inconsistency management. These formal definitions provide the mathematical foundation for the COMIM approach, establishing precise semantics for the concepts and operations described in the subsequent sections.

Definition 1 (Operation). An *operation* o represents a single change in the collaborative modeling environment and is defined as a 5-tuple:

$$o = (CID, VID, EID, OP, TS)$$

where:

- $CID \in ClientID$ is the unique identifier of the client initiating the operation
- $VID \in ViewID$ is the identifier of the view where the operation was performed
- $EID \in ElementID$ is the unique identifier of the model element being modified
- $OP \in \{Create, Delete, Update\}$ is the type of operation, where $Update$ can be further specified as $Update(Property, NewValue)$
- $TS \in \mathbb{N}$ is a logical timestamp representing the operation's global order

Definition 2 (Operation Tree). The global state of the collaborative session is maintained in an *Operation Tree* $T = (N, E)$, where:

- N is a set of nodes, each representing an operation o
- E is a set of edges representing the happens-before relationships between operations
- The root of T represents the initial empty model state
- Each path from root to leaf represents a sequence of operations leading to a specific model state
- Branches in T represent concurrent modifications that have not yet been reconciled

Definition 3 (Scope). A *scope* S is a set of model elements $\{e_1, e_2, \ldots, e_n\}$ defined by a query over the metamodel. Formally:

$$S = \{e \in M \mid \text{query}(e) = \text{true}\}$$

where M is the model and query is a predicate defined over the metamodel elements.

Definition 4 (Consistency Rule). A *consistency rule* R is a predicate associated with a scope S_R, defined as:

$$R(S_R) \rightarrow \{\text{True}, \text{False}\}$$

The rule R evaluates to True if the elements in scope S_R satisfy the consistency condition, and False otherwise.

Definition 5 (Model State). The *state* of a model M at time t is determined by applying all operations along a path P in the operation tree T:

$$\text{state}(M, t) = \text{apply}(\{o_1, o_2, \ldots, o_n\}, M_0)$$

where $o_1, o_2, \ldots, o_n$ are operations along path P with $TS(o_i) \leq t$, and M_0 is the initial model.

Definition 6 (Consistent State). A model M is in a *consistent state* at time t if and only if:

$$\forall R_i \in \mathcal{R}, R_i(S_{R_i}) = \text{True}$$

where $\mathcal{R}$ is the set of all active consistency rules in the system.

Definition 7 (View Projection). A *view* V is a projection of model M defined by a view definition function:

$$V = \text{project}(M, \text{viewdef})$$

where viewdef specifies which elements and properties of M are visible in V.

Definition 8 (Evaluate Operation). The EVALUATE operation checks if a new operation o maintains consistency:

$$\textsc{Evaluate}(o, T, \mathcal{R}) \rightarrow (\text{result}, \text{feedback})$$

where result $\in$ {CONSISTENT, INCONSISTENT} and feedback provides diagnostic information when inconsistent.

Definition 9 (Query Operation). The QUERY operation identifies equivalent and conflicting operations across views:

$$\textsc{Query}(o, T, M) \rightarrow (O_{\text{equivalent}}, O_{\text{conflicting}})$$

where $O_{\text{equivalent}}$ and $O_{\text{conflicting}}$ are sets of operations that are equivalent to or conflict with o, respectively.

Definition 10 (Reorder Operation). The REORDER operation computes a valid sequence of operations from a set of conflicting operations that, when applied, restores consistency:

$$\textsc{Reorder}(O_{\text{conflict}}, T, \mathcal{R}) \rightarrow (O_{\text{valid}}, \text{feedback})$$

where:

- $O_{\text{conflict}} \subseteq \mathcal{O}$ is a set of conflicting operations identified by the QUERY function,
- T is the current Operation Tree,
- $\mathcal{R}$ is the set of all active consistency rules,
- $O_{\text{valid}} = \langle o_1, o_2, \ldots, o_k \rangle$ is a sequence (permutation) of operations from O_{conflict} that, when applied in order to the current model state, results in a consistent state (if such a permutation exists),
- Feedback provides diagnostic information when no valid permutation exists.

The operation attempts different permutations of O_{conflict} and checks each candidate sequence against the consistency rules $\mathcal{R}$. The first permutation that satisfies $\forall R \in \mathcal{R} : R(S_R) = \text{True}$ is returned as O_{valid}. If no permutation yields consistency, the operation returns NONE for O_{valid} and generates appropriate feedback.

Property 1 (Incremental consistency checking). COMIM performs consistency checking incrementally by only re-evaluating rules whose scopes intersect with modified elements:

$$\text{RulesToCheck}(o) = \{R \in \mathcal{R} \mid S_R \cap \text{affectedElements}(o) \neq \emptyset\}$$

Property 2 (Multi-view coordination). Operations across different views are coordinated through shared model elements:

$$\text{RelatedViews}(VID) = \{v \in \text{Views} \mid \exists e \in M : e \in V_{VID} \cap V_v\}$$

4.2 Conceptual overview of the collaboration architecture

The proposed collaboration architecture is designed to support real-time collaboration among multiple users working on different views of one or more models. As shown in Figure 1, at the center of the architecture is the Collaboration Server, which acts as the hub for managing and propagating changes across views. The server ensures that changes made by one collaborator are synchronized with the relevant views of other collaborators. This setup allows users to work on specific views without being overwhelmed by changes that are irrelevant to their tasks.

The architecture involves multiple views (e.g., View1, View2, View3) that represent different perspectives or representations of the underlying model (e.g., Model of a Video On-demand System). Where collaborators interact with these views to make changes.

This separation allows collaborators to focus on specific aspects of the project. For instance, one collaborator might work on View1, while another works on View3, without the need to see each others changes unless the views are interdependent.

The architecture uses channels to propagate changes between the Collaboration Server and the views. Each view has its own dedicated channel (e.g., Channel for View1). These channels ensure that changes are routed efficiently and only to the relevant views. For example:

- Changes made in View1 are sent through the Channel for View1 to the Collaboration Server.
- The server then propagates these changes to the collaborators on View1, as well as collaborators on other views (e.g., View2) if they are interdependent with View1.

This channel-based approach ensures that collaborators only receive updates that are relevant to their work, reducing unnecessary noise and improving efficiency.

The Collaboration Server plays a critical role in managing change propagation and ensuring consistency across views by acting as the central authority for receiving changes from collaborators via dedicated channels, evaluating the impact of changes on interdependent views, propagating changes synchronously or asynchronously based on the relationship between the views, and resolving inconsistencies through mechanisms like reordering operations or sending feedback to collaborators. For example, if a change in View1 affects View2 of the underlying model, the server ensures the change is propagated immediately to View2, and if an inconsistency is detected, it may reorder operations or notify collaborators to resolve the issue.

4.3 View definition

Figure 2 presents an overview of the proposed process to define views. The process starts with receiving an *original metamodel*, which is the input for a step named *Analyser*. In the *Analyser* step, users can define *priorities* as well as *required metamodel elements*. This process results in a *Syntax Graph*. The resulting syntax graph goes as input to the *Query Generation* step. This step also receives a *Model* as another input. The query generation is executed and

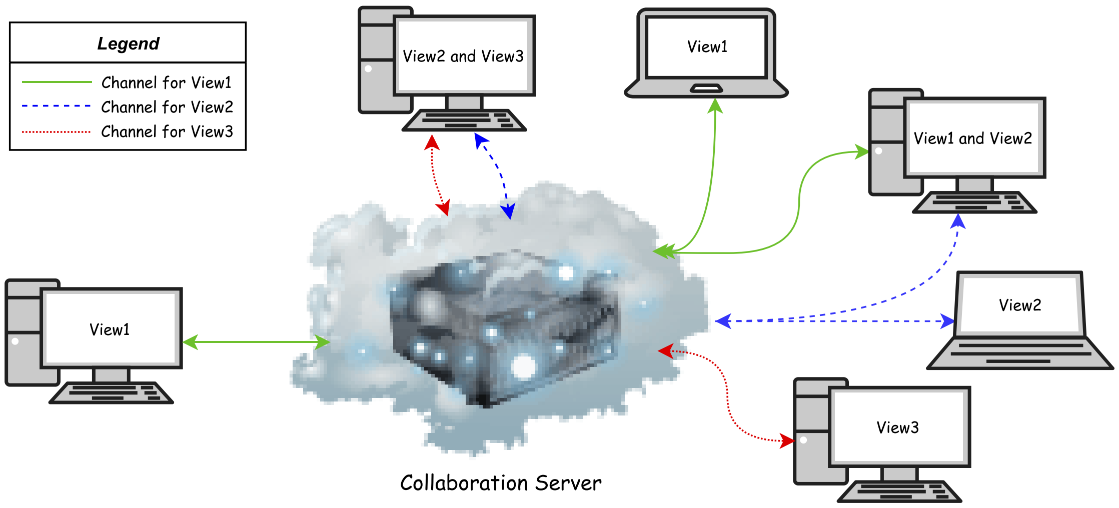

FIGURE 1
Overview of real-time multi-view collaborative modeling architecture.

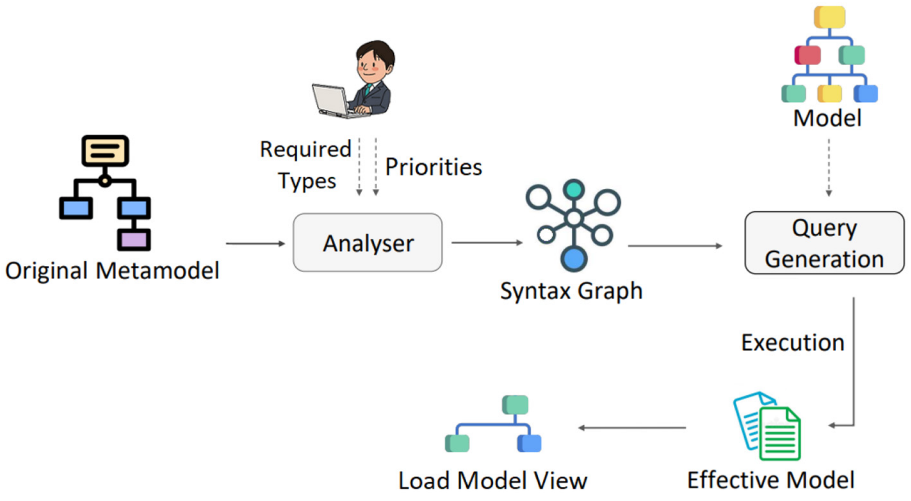

FIGURE 2
An overview of the proposed process for definition of views.

leads to an *Effective Model*, which can load and represent the *Model View*. To select required element types in the *Analyser* step of our approach, we have two strategies. The first strategy is user-based selection, and the second is preference-based selection. In the following, we explain each strategy. These strategies enable configurable view personalization, ensuring that each collaborator interacts with a model projection tailored to their specific tasks and concerns.

4.3.1 User-based selection

In model-driven engineering, a metamodel serves as the foundational structure defining the elements and rules for a family of models. In the user-based selection strategy, users review the metamodel and can interactively select specific elements that are pertinent to their needs. This selection process allows users to focus on relevant aspects of the metamodel, tailoring it to their specific context. Once the user has made their selections, the system automatically filters a received model, extracting only those elements that match the selected types. This results in a simplified model view, which presents a focused representation of the model, highlighting only the elements of interest. This approach enhances usability by allowing users to concentrate on specific components, facilitating easier analysis and understanding while minimizing extraneous details.

4.3.2 Preferences-based selection

The preferences module is a configurable component that operates alongside the core view definition logic, allowing for a versatile implementation of specific preferences that can be utilized to address both syntactic errors and design smells within models.

This modularity ensures that the same quality characteristics can be applied across various issues, as long as the models being analyzed share a common format, such as Ecore class diagrams in our previous discussions. To date, we have enhanced this module by incorporating three key preferences: quality characteristics, model distance, and coupling. Below, we elaborate on the implementation of each preference.

1. Quality characteristics. Our implementation of quality characteristics draws from established literature (Boehm et al., 1976; Ortega et al., 2003) and integrates user-defined preferences to evaluate model quality. This quality assessment tool not only provides an evaluation framework but also allows users to define custom quality metrics. Currently, we have identified and implemented the following quality characteristics as user preferences: maintainability, understandability, complexity, and reusability. By leveraging this implementation, we systematically enhance Ecore class diagrams, ensuring that the selected quality characteristics are prioritized during the element selection process. The evaluation values for these characteristics serve as rewards within our model, enabling the COMIM framework to select view elements that yield models of superior quality. When projecting a view of a software class diagram, a user may set a preference to prioritize high maintainability. In practice, the system calculates maintainability scores for candidate elements based on measurable attributes like coupling and cohesion. Elements with higher scores are prioritized for inclusion in the view. For instance, given a choice between *OrderProcessor* (tightly coupled to 10 other classes) and *Logger* (minimal dependencies), the system would assign *Logger* a higher maintainability score and preferentially include it when constructing a view focused on maintainability.
2. Model distance. We utilize a model distance metric to inform the element selection process for Ecore class diagrams. The concept of model distance has been extensively explored in the literature, particularly in the context of model comparison. The distance between two models is determined by analyzing their conceptual similarities and differences, factoring in shared elements and unique components (Sharbaf et al., 2022). We derive this distance metric using a dedicated model distance calculator. By applying this metric, we encourage the preservation of the original model structure during view projection, thereby minimizing unwanted side effects in the modified model. This calculation is executed through an Eclipse plugin, which incorporates a model matching algorithm defined using an Epsilon Comparison Language (ECL) script (Kolovos, 2009). This script is customizable, allowing for tailored adjustments to the distance metric as required. The system evaluates potential view projections by calculating their structural distance from the original model. Among candidate projections, it selects the one that minimizes this distance. For a class diagram with 20 classes, projections including only peripheral classes would have high distance scores, while projections preserving the central 5-class core would have low distance scores and thus be preferred.
3. Coupling. To guide the view projection of UML models, we employ metrics provided by SDMetrics (Wust, 2006). Specifically, we utilize the MsgSent (number of messages sent) and MsgRecv (number of messages received) metrics (Briand et al., 1997) to assess coupling in UML sequence diagrams. By analyzing the number of messages exchanged between lifelines, we quantify the interdependence among them. The coupling for each lifeline is calculated by aggregating these message counts, enabling users to address inconsistencies between class and sequence diagrams while maintaining loose coupling in the sequence diagram. This approach empowers users to implement element selection strategies that prioritize minimal interdependencies, ultimately leading to cleaner and more maintainable designs.

The three preference strategies described above (i.e., quality characteristics, model distance, and coupling) are illustrative, pluggable modules within COMIM. While the coupling metric example is specific to UML, the framework allows for the integration of analogous, domain-specific metrics for other modeling languages. These preferences are optional extensions that demonstrate how personalized view selection can be tailored to different domains and quality concerns.

4.4 Inconsistency management infrastructure overview

We propose a structured framework for managing intra-model inconsistencies in collaborative modeling environments. Figure 3 shows an overview of the proposed appoach that includes *Scope Definition*, inconsistency evaluation using *Consistency Oracle* based on the scopes, and *Inconsistency Resolution* by reordering the execution of change operations or computing feedbacks for inconsistency repair. By defining scopes, conducting real-time inconsistency checks, leveraging a consistency oracle for conflict resolution, and computing valid operation orders, the system enhances the reliability and integrity of multi-view collaboration. This holistic strategy not only improves the efficiency of inconsistency management but also supports seamless collaboration among engineers working on complex models.

The objective of *Scope Definition* is to create a clear boundary around the model elements relevant to specific consistency rules. Each consistency rule evaluation (CRE) assesses the related consistency rules for each scope. A scope is a set of model elements and their properties that are accessed during the evaluation of the rule. The system manager declares a scope by specifying a query (e.g., over the metamodel) that defines the region of interest. Then, during the first execution of the Consistency Rule Evaluation (CRE), the system automatically populates this scope with the concrete set of model elements and their properties that are both accessed during the rule evaluation and match the declared query. This populated set is then dynamically maintained as the model evolves. By limiting the evaluation to only the elements in the scope, the system can efficiently manage inconsistencies without re-evaluating the entire model, allowing for more fine-grained checks.

Inconsistency checking based on scopes continuously monitor changes in the model elements defined in the scopes to detect inconsistencies. The system performs incremental consistency checks at regular intervals or immediately after changes, focusing

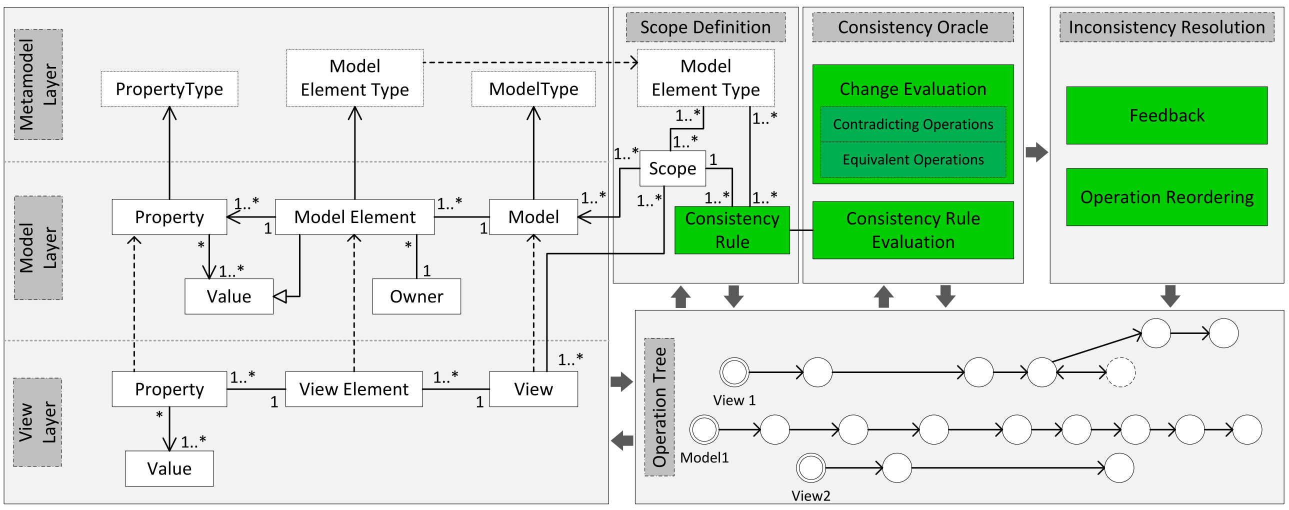

FIGURE 3
The COMIM inconsistency management infrastructure data flow. The process begins with a user defining Scopes and Consistency Rule Definitions (CRDs). When a change operation is received, the Scope Definition module identifies the affected scopes. The Consistency Oracle then evaluates the relevant CRDs against the current state of the Operation Tree. If an inconsistency is detected, the Inconsistency Resolution module either computes a valid reordering of operations (updating the Operation Tree) or generates feedback for the user.

on the elements within the defined scopes. When a change occurs in an element within a scope, the corresponding CRE is re-evaluated to determine if the model remains consistent. This allows inconsistencies to be identified soon after they are introduced.

The *Consistency Oracle* is the core component responsible for detecting and managing inconsistencies. It achieves this by evaluating incoming change operations against the set of active consistency rules. It contains change evaluation module to detect contradicting as well as equivalent operations that lead to model inconsistency. Unlike general-purpose constraint solvers or model checkers, our consistency oracle specializes in real-time collaborative contexts by: (1) operating incrementally on scoped model fragments, (2) prioritizing repair strategies based on user preferences, and (3) maintaining operation trees for inconsistency resolution—balancing automation with collaborative control. The consistency oracle operates as a centralized module that interprets change operations based on a specified consistency model. It maintains tree of operations for each model and related views and computes valid responses for queries based on the defined consistency guarantees (e.g., strong consistency, eventual consistency). This allows for systematic inconsistency resolution, providing a coherent view of the model state across multiple collaborators and ensuring that all operations align with the specified consistency model.

The *Operation Reordering* computes order of operations for inconsistency repair to determine the valid order of operations that can be applied to resolve inconsistencies. The consistency oracle computes possible interpretations of operation order based on the consistency model in use. For strong consistency, the total order matches the global history, while for eventual consistency, all permutations of previous operations may be considered. For models like monotonic reads or read-my-writes, the oracle identifies unordered operations and applies the appropriate conflict resolution strategies. This ensures that the resolution of inconsistencies respects the logical relationships defined by the consistency model, facilitating a consistent and reliable state in the collaborative environment. In a real-time multi-view collaboration environment, managing inconsistencies arising from concurrent modifications can be performed in following desired times:

1. After each modification: implementing real-time checks immediately after a collaborator makes a change allows for quick detection of inconsistencies. This approach provides immediate feedback, enabling users to address issues before proceeding further.
2. Before committing changes: before any collaborator commits their modifications to the target model, a thorough check can ensure that their changes do not conflict with others. This step acts as a safeguard, preventing problematic changes from being integrated into the final model.
3. At regular intervals: establishing periodic checks (e.g., every few minutes) can help identify inconsistencies that may arise from multiple collaborators' changes over time. This approach balances real-time responsiveness and system performance, reducing the likelihood of significant discrepancies accumulating unnoticed.
4. On model view synchronization: whenever a collaborator synchronizes their view with the target model or with other views, inconsistency checks should be performed. This ensures that any new changes are compatible with the current state of the model, facilitating smoother integration.
5. Before final rendering: prior to generating the final model from the various views, a comprehensive inconsistency check should be performed. This is a crucial step to ensure that all modifications align with the desired state of the model, allowing for a coherent final output.

6. User-triggered checks: providing collaborators the option to manually trigger inconsistency checks at any point can empower users to verify their work whenever they feel necessary, adding flexibility to the collaboration process.

In COMIM's real-time collaborative environment, inconsistency checking is performed at two key points to balance responsiveness with comprehensive coverage. First, real-time checks occur immediately when the collaboration server receives a change operation, before the operation is committed to the shared operation tree and propagated to other views. This provides the submitting user with immediate feedback. Second, periodic checks run at predefined intervals as a safety net, evaluating the current state of the operation tree to detect inconsistencies that may arise from complex, transitive interactions between multiple changes over time. This hybrid approach ensures both low-latency feedback and long-term model consistency.

The proposed environment facilitates collaboration among engineers by enabling the sharing of knowledge related to their artifacts, specifically models. This collaboration is supported by an operation-based generic infrastructure. Within this infrastructure, models are composed of multiple elements, each containing properties that adhere to a specific metamodel. Changes made to these models are systematically recorded through operations that include creating, deleting, and modifying model elements and their properties. These operations are organized into an operation tree, which branches to represent models and each view developed collaboratively. The infrastructure supports the concurrent work of multiple users by allowing for the splitting and merging change operations to this operation tree.

In this infrastructure, a meta-metamodel defined as a streamlined MOF enables the creation of a domain-specific metamodel. A collaborative session in COMIM operates on views of a model that conform to this metamodel. By leveraging this infrastructure, our approach can support collaborative work, maintain a history of change operations, and be adapted to different modeling domains by reconfiguring the framework with the appropriate metamodel. Additionally, the infrastructure facilitates communication with various services that extract and manipulate critical information from these views. We adapt established approaches to enhance the consistency checking service by implementing immediately check after each modification as well as periodic checks at regular intervals. By performing immediately check, the system evaluates the consistency oracle to prepare on-demand feedback and prevent propagation of inconsistency. In addition, by conducting consistency checks at predefined intervals, the system evaluates the operation tree for potential inconsistencies, ensuring that all collaborators are informed of any conflicts that may arise. This proactive approach allows users to address issues promptly, maintaining the integrity of the models as they evolve.

Figure 3 illustrates the environment composed of the operation-based tree that logs all changes applied within the streamlined MOF framework. In this context, a view presents an projection of model elements and related properties. A model consists of model elements, each containing properties. A property can hold a single value or multiple values (collections). The types of the model, model elements, and properties are determined by the metamodel of the artifacts being represented. The model types are defined at the metamodel level, allowing for instantiation as model elements with specific property values.

Changes made to define the types, property types, model elements, and so forth, are stored as a continuous stream of events within the operation tree (top-right part of Figure 3). This operation tree serves as the architecture's backbone, providing the operations necessary for services, including consistency checking, to analyze relevant changes (e.g., the establishment of equivalent or contradicting operations). In the following, we will focus on the incosistency checking modules, including Scope Definition, Consistency Oracle, and Inconsistency Resolution.

4.4.1 Scope definition

Incremental consistency checkers allow for fine-grained reactions to changes in model elements, thereby avoiding the need to re-evaluate the entire model. To enhance inconsistency management, we utilize the concept of scope (Egyed, 2010). A scope is defined as a set of model elements associated with a consistency rule evaluation (CRE). When any element within this scope is modified or a new element related to this scope is added, it triggers the re-evaluation of the corresponding CRE. The scope is automatically populated during the initial execution of the CRE, resulting in a list of model elements accessed during the rule evaluation on the UML model. Only modifications to these accessed elements can potentially alter the CRE state from consistent to inconsistent or vice versa.

This mechanism implements incremental consistency checking (Property 1), which ensures that only consistency rules whose scopes intersect with the modified elements are re-evaluated. Formally, for an operation o, the set of rules to check is $\mathit{RulesToCheck}(o)$. Thus, a CRE is re-executed only when an element within its defined scope changes, avoiding a full model re-evaluation. To support ongoing consistency checks, our system implements a strategy of periodic evaluations at regular intervals. This means that, in addition to immediate re-evaluations triggered by scope changes, the consistency checker periodically assesses the state of the model, allowing for the detection of inconsistencies that may arise from concurrent modifications by different engineers. This proactive approach helps maintain the integrity of the model throughout the collaboration process. The consistency checker utilizes a uniform representation of artifacts to manage both rule definitions (CRDs) and evaluation results (CREs). It creates instances of two distinct engineering artifact types:

- Consistency rule definition (CRD) artifact: this artifact contains information about the type of artifact to which a rule applies (the context) and the rule itself, typically defined in OCL. CRDs serve as the foundational representation for consistency checks.
- Consistency rule evaluation (CRE) artifact: these artifacts implement CRDs for specific artifact instances (context elements). They store the current consistency result and maintain a reference to the associated CRD. Additionally, CREs track their "scope," which is a list of artifact/property pairs relevant to the evaluation of a particular rule within a specific context.

When an engineer creates a type, it can be designated as the context for a CRD. Upon the creation of a CRD—facilitated by a specialized tool—a corresponding CRE is automatically generated for each artifact of the specified context type. These CREs are stored within the workspaces of the respective artifacts. During the initial evaluation, the scope of each CRE is constructed, encompassing all relevant artifacts and their properties traversed during the rule evaluation. A reference to the CRE is added to the properties involved, enabling efficient tracking. Should any of these properties change, the consistency checker follows the reference back to the CRE to initiate a re-evaluation. By integrating periodic consistency checks with targeted evaluations driven by changes within the defined scope, our approach provides a configurable foundation for inconsistency management, allowing teams to focus checking efforts on specific, critical areas of the model relevant to their project.

4.4.2 Consistency oracle

A consistency oracle is a software artifact designed to manage inconsistencies within the defined scopes of multi-views in collaborative modeling environments. It features an interface similar to that of a generic distributed storage system, but its primary goal is not to store data efficiently. Instead, the consistency oracle evaluates all possible inconsistencies for any change operation performed by the client, making its architecture fundamentally different from traditional storage systems. The consistency oracle is a core component of the proposed framework, designed to ensure the consistency and integrity of models in a multi-view collaborative environment. It operates as a centralized module that monitors changes, evaluates consistency rules, and detects inconsistencies arising from conflicting or equivalent operations. The oracle uses a specified consistency model to provide systematic inconsistency identification and resolution.

At the heart of the consistency oracle is the *Consistency Rule Evaluation (CRE)* module. Each consistency rule is defined based on the elements within a specific scope, which represents a set of model elements and their properties relevant to the rule. The CRE assesses these rules to determine whether the model remains consistent after changes are made. During the evaluation, the oracle automatically computes the relevant consistency rules based on the scope of elements involved in the modifications. These rules are dynamically selected as changes occur, ensuring that the evaluation remains accurate and up-to-date. By focusing only on the elements within the defined scope, the CRE enables efficient and fine-grained consistency checks without the need to re-evaluate the entire model.

The consistency oracle also includes a *Change Evaluation* module, which plays a critical role in detecting inconsistencies. This module monitors changes and identifies inconsistencies [i.e., contradicting operations (changes that conflict with each other) and equivalent operations (changes that lead to the same meaning and redundancy)], leveraging multi-view coordination (Property 2) to determine which views are affected. When an operation o is performed in view VID, the system computes $RelatedViews\ (VID)$ to identify views sharing elements and propagates changes accordingly.

For example, if two collaborators simultaneously modify the same model element in incompatible ways, the Change Evaluation module detects this contradiction by checking operation trees and flags it as an inconsistency. Similarly, if multiple operations result in the same redundant state, the module identifies them as equivalent and ensures they are addressed collectively. This proactive detection mechanism allows the system to identify inconsistencies early, minimizing their impact on the collaborative workflow.

Figure 4 depicts the structure of our consistency oracle module. This module is designed to handle fundamental operations such as addition, deletion, and update, enabling it to replicate more intricate operations commonly encountered. Whenever a user modifies a view, these change operations are transmitted to the Consistency Oracle module in real-time to ensure consistency. Each change *operation* as defined in Section 4.1 is an <INPUT> tuple including <(CID) (VID) (EID) (OP) (TS)>.

The Consistency Oracle offers two primary interface calls: *Evaluate((INPUT), (CRDs), (M))* and *Query((INPUT), (M))*. The parameter (M) identifies the model under which the operation is performed. When Evaluate is called, (M) is also used to update the session information accordingly. The Evaluate request fetches the relevant consistency rules based on the (INPUT) tuple, enabling a query-matching process. The oracle then returns responses derived from the model's operation tree. On the other hand, the Query request traverses the operation tree of views to identify equivalent or conflicting change operations. Since the operation trees maintain a comprehensive history of all changes applied to a view, they facilitate the detection of both equivalencies and contradictions.

Each Evaluate request adds an (INPUT) tuple to the operation tree of the associated view, while a Query request enables updates to the operation tree of the model based on the output from the consistency oracle and the inconsistency resolution module. If the consistency oracle identifies an inconsistency, it forwards

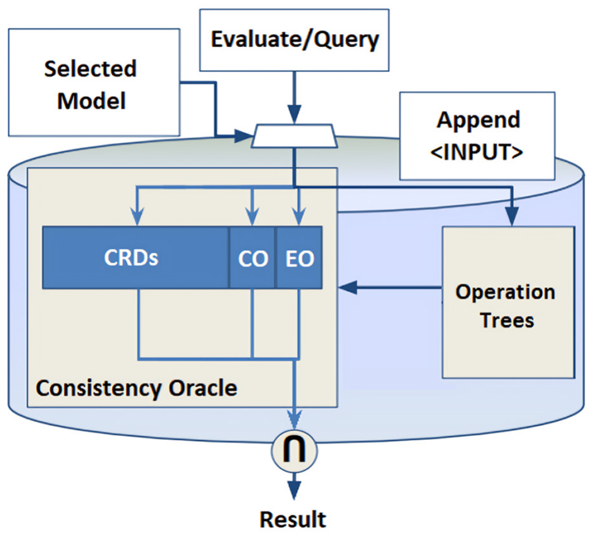

FIGURE 4
An overview of the consistency oracle.

the relevant details to the Inconsistency Resolution component for further processing.

The Inconsistency Oracle performs real-time inconsistency checks when change operations are received by the server, as well as periodic checks at regular intervals on the current operation tree state. Real-time checks ensure that inconsistencies are detected and addressed as soon as they occur, preventing their propagation through the model. Periodic checks, on the other hand, evaluate the entire operation tree for potential inconsistencies, ensuring that all collaborators are informed of any conflicts that may have arisen over time. This combination of real-time and periodic checks supports consistency maintenance in dynamic collaborative sessions. The consistency oracle's design allows for easy extension to support new consistency models simply by implementing and integrating new modules. This adaptability is crucial for effectively managing and repairing inconsistencies in collaborative modeling environments. Furthermore, by maintaining a history of change operations and leveraging a streamlined meta-metamodel, the oracle supports multiple modeling languages and domains, making it a versatile tool for collaborative modeling.

4.4.3 Inconsistency resolution

The resolution process involves either reordering the execution of change operations or computing feedback for modelers to repair the inconsistency. For example, if two operations conflict, the Inconsistency Resolution module based on the oracle result tries to determine a valid order in which they can be applied to restore consistency. Alternatively, it may provide feedback to the collaborators, suggesting modifications to their changes to align with the consistency rules. This dual approach ensures that inconsistencies are resolved systematically, maintaining the integrity of the model while supporting collaborative work. A key feature of the Inconsistency Resolution Module is its ability to reorder operations based on the specified consistency evaluation rules. For instance, under strong consistency, the oracle ensures that the total order of operations matches the history of changes, providing a single, coherent view of the model state. In contrast, for eventual consistency, the oracle considers all possible permutations of previous operations applied on views, allowing for more flexible conflict resolution.

The inconsistency resolution module is designed to function within a testing environment where both clients and servers can operate on a single machine. This configuration enables the use of a single clock to establish a total order of all operations recorded in the history. This total order reflects the behavior of a distributed system in which all servers have perfectly synchronized clocks and consistently timestamp every operation. Given this total order, the inconsistency resolution module is tasked with computing all possible interpretations of operation sequences based on the specified consistency evaluation rules. In the simplest case of strong consistency, the only valid interpretation of the operation sequence is the total order defined in the model tree operation history. Here, all operations must occur sequentially, preserving their exact order.

In contrast, under eventual consistency, the total order can be interpreted in any arbitrary sequence by considering the ownership of elements in the applied operations. For queries referencing the model tree operation history, the inconsistency resolution module must evaluate all permutations of operations that precede the current operation. This exhaustive evaluation ensures that the module accounts for every possible logical sequence in which operations could occur. For cases involving limited partial orders—such as monotonic adds or deletes—the inconsistency resolution module must identify operations that are considered unordered according to the model's definitions (i.e., where no happens-before relationship exists). In such scenarios, the module applies conflict resolution strategies as dictated by the specific consistency evaluation rule, ensuring that the resolution process adheres to the intended guarantees of the model. Through these mechanisms, the consistency oracle effectively navigates the complexities of operation ordering, enabling robust inconsistency resolution in collaborative multi-view environments.

4.4.4 Algorithms for inconsistency management

The formal operations and properties defined in Section 4.1 are implemented by the algorithms presented in this subsection. These algorithms form the executable core of COMIM's inconsistency detection and resolution mechanism.

Algorithm 1 (Evaluate) implements the EVALUATE operation (Definition 8). It is the main entry point for consistency checking when a new change operation is submitted. The algorithm determines which rules are affected (using Algorithm 3), evaluates them, and if inconsistencies are detected, attempts resolution by identifying conflicts (Algorithm 2) and reordering operations (Algorithm 5). Algorithm 2 (Query) implements the QUERY operation (Definition 9). It identifies operations that are equivalent or conflicting with a given operation by examining related views (Algorithm 4) and rules with overlapping scopes (Algorithm 3).

Algorithm 3 (RulesToCheck) and Algorithm 4 (getRelatedViews) implement the incremental consistency checking (Property 1) and multi-view coordination (Property 2) respectively. They enable efficient, scope-based evaluation by limiting checks to relevant rules and views. Algorithm 5 (Reorder) implements the REORDER operation (Definition 10). It attempts to resolve conflicts by systematically testing permutations of conflicting operations until a valid order that satisfies all consistency rules is found. These algorithms are referenced in the illustrative example that follows to demonstrate COMIM's workflow in practice.

4.5 Illustrative example

To demonstrate our intra-model inconsistency management approach, we use a single, unified UML model of a Video on Demand (VOD) system (Egyed, 2006). A UML model is a semantic repository of system elements. This model can be represented through various diagrams, which are diagrammatic projections that highlight significant information while omitting less important details (Rumpe, 2016). For this example, we project the VOD model into two such diagrams: a class diagram showing static structure, and a sequence diagram showing dynamic behavior. Each diagram provides an incomplete view of the whole system model, focusing on specific parts or functions. Figure 5 shows the UML model of

```
1: Evaluate($o, T, \mathcal{R}$).
Require: A new operation $o = (CID, VID, EID, OP, TS)$, the current Operation Tree $T$, the set of all consistency rules $\mathcal{R}$.
Ensure: A consistency result ($CONSISTENT$, $INCONSISTENT$), an optional set of feedback messages, and optionally a reordered sequence $O_{valid}$.
2: Step 1: Append & propagate
3: Append $o$ to the relevant branch in $T$ corresponding to view $VID$.
4: Propagate the effect of $o$ to compute a candidate model state $M_{candidate}$.
5: Step 2: Determine rules to check
6: Let $R_{relevant} \leftarrow$ RulesToCheck($o, \mathcal{R}$)    ▷ Call Algorithm 3
7: Step 3: Evaluate rules
8: Let $violations \leftarrow \emptyset$
9: for each rule $R$ in $R_{relevant}$ do
10:    $result \leftarrow R(M_{candidate})$
11:    if $result$ is $False$ then
12:       $violations \leftarrow violations \cup \{R\}$
13:    end if
14: end for
15: Step 4: Check for violations and attempt reordering
16: if $violations$ is empty then
17:    return ($CONSISTENT, \emptyset, \emptyset$)
18: else
19:                              ▷ Try to resolve by reordering
20:    Let $(O_{equivalent}, O_{conflicting}) \leftarrow$ Query($o, T, M_{candidate}$)    ▷ Call Algorithm 2
21:    Let $O_{set} \leftarrow O_{conflicting} \cup \{o\}$
22:    Let $(O_{valid}, feedback_{reorder}) \leftarrow$ Reorder($O_{set}, T, \mathcal{R}$)    ▷ Call Algorithm 5
23:    if $O_{valid}$ is not None then
24:                                  ▷ Valid reordering found
25:       Update operation tree $T$ according to $O_{valid}$
26:       return ($CONSISTENT, feedback_{reorder}, O_{valid}$)
27:    else
28:                  ▷ Reordering failed, generate feedback
29:       Let $feedback \leftarrow \emptyset$
30:       for each rule $R$ in $violations$ do
31:          Generate a user-friendly message $m$ describing the violation of $R$ by $o$
32:          $feedback \leftarrow feedback \cup \{m\}$
33:       end for
34:       return ($INCONSISTENT, feedback, \emptyset$)
35:    end if
36: end if
```

Algorithm 1. Evaluate.

```
1: Query($o_{query}, T, \mathcal{R}$).
Require: A query operation $o_{query} = (CID, VID, EID, OP, TS)$, the current Operation Tree $T$, the set of all consistency rules $\mathcal{R}$.
Ensure: A set of equivalent and conflicting operations $\{O_{equivalent}, O_{conflicting}\}$.
2: Step 1: Get Current Model State and Identify Related Views
3: Let $M_{current} \leftarrow$ getCurrentModelState($T$)    ▷ Extract state from main branch of $T$
4: Let $V_{related} \leftarrow$ getRelatedViews($VID(o_{query}), M_{current}$)    ▷ Using Property 2 and Algorithm 4
5: Step 2: Collect Operations from Related Views
6: Let $O_{candidate} \leftarrow \emptyset$
7: Let $O_{equivalent} \leftarrow \emptyset$, $O_{conflicting} \leftarrow \emptyset$
8: for each view $v$ in $V_{related}$ do
9:    Let $B_v \leftarrow$ getBranch($T, v$)    ▷ Get operation branch for view $v$
10:    for each operation $o_i$ in $B_v$ where $TS(o_i) < TS(o_{query})$ do
11:       if RulesToCheck($o_i, \mathcal{R}$) $\cap$ RulesToCheck($o_{query}, \mathcal{R}$) $\neq \emptyset$ then
12:          $O_{candidate} \leftarrow O_{candidate} \cup \{o_i\}$
13:       end if
14:    end for
15: end for
16: Step 3: Detect Equivalent Operations
17: for each operation $o_c$ in $O_{candidate}$ do
18:    if areEquivalent($o_c, o_{query}, M_{current}$) then
19:       $O_{equivalent} \leftarrow O_{equivalent} \cup \{o_c\}$
20:    end if
21: end for
22: Step 4: Detect Conflicting Operations
23: for each operation $o_c$ in $O_{candidate}$ do
24:    if areConflicting($o_c, o_{query}, M_{current}$) then
25:       $O_{conflicting} \leftarrow O_{conflicting} \cup \{o_c\}$
26:    end if
27: end for
28: Step 5: Return Results
29: return ($O_{equivalent}, O_{conflicting}$)
```

Algorithm 2. Query.

the VOD system that both Harry and Bob are familiar with. The VOD system model is represented through two diagrams: (a) a class diagram showing the entities *Media*, *Audio*, *Video*, *Streamer*, and *Logger*, and (b) a sequence diagram outlining the interaction among those *Video* and *Streamer* classes to describe the process of *streaming*, *pausing*, and *stopping* a video.

Furthermore, to provide personalized multi-view collaborative modeling, the design manager follows the view definition step of the proposed approach to prepare a projection on the class diagram based on the separation of concern and coupling preferences to display the elements of class diagram into the *Structural* view and the *Streaming* view. Figure 6 shows the first version of the resulted views. He also defined the *Streamer Interaction* view as a distinct view on the sequence diagram. The manager also defined two scopes. Scope 1 focus on class diagram elements, and Scope 2 contains sequence diagram elements.

```
1: RulesToCheck($o$, $\mathcal{R}$).
Require: An operation $o = (CID, VID, EID, OP, TS)$, the set of all consistency rules $\mathcal{R}$
Ensure: The set of consistency rules that need to be checked for operation $o$
2: Step 1: Determine Directly Affected Elements
3: Let $E_{direct} \leftarrow$ getDirectlyAffectedElements($o$)
4: Step 2: Determine Transitively Affected Elements
5: Let $E_{transitive} \leftarrow$ getTransitivelyAffectedElements($E_{direct}$)
6: Step 3: Combine All Affected Elements
7: Let $E_{affected} \leftarrow E_{direct} \cup E_{transitive}$
8: Step 4: Find Rules with Overlapping Scopes
9: Let $R_{relevant} \leftarrow \emptyset$
10: for each consistency rule $R \in \mathcal{R}$ do
11:     Let $S_R \leftarrow$ getScope($R$)  ▷ Get the scope of rule $R$
12:     if $S_R \cap E_{affected} \neq \emptyset$ then
13:         $R_{relevant} \leftarrow R_{relevant} \cup \{R\}$
14:     end if
15: end for
16: Step 5: Return Relevant Rules
17: return $R_{relevant}$
```

Algorithm 3. RulesToCheck.

To ensure consistency in the final product, the UML diagrams must adhere to the following rules:

- Rule 1: a subclass cannot declare an attribute with the same name of its superclass (e.g., the Video and Audio classes cannot contain new attributes with name of title and format).
- Rule 2: the message direction must match the Class Association direction (e.g., Sending a message from the Streamer to Video instance in sequence diagram is not possible since the Streamer class don't have association to the Video class).
- Rule 3: the messages used in the sequence diagram must be present as an operation in their respective receiver's class (e.g., the stream() message must be present in the Streamer class).

Inconsistencies occur when models violate fundamental well-formedness constraints, which we formalize as consistency rules. These rules can be precisely specified using Object Constraint Language (OCL) (Cabot and Gogolla, 2012). We formalize these expressions as CRDs—see CRD1 through CRD3 in Listings 1–3. Note that:

- CRD1 applies to Scope 1 (class diagram).
- CRD2 and CRD3 apply to Scope 2 (sequence diagram).

In the modeling time and based on the expertise, Harry focuses on the structural view, while Bob works in the streaming view as well as the streamer interaction view. Harry added *duration* attribute to *Media* class and *resolution* attribute to *Video* class. In parallel, Bob added the association from *Streamer* class to *Logger* class, and added *title* attribute to *Video* class in the streaming view, and *pause()* method from *Streamer* to *Video* instances in the streamer interaction view.

```
1: getRelatedViews($vid$, $M$).
Require: A view identifier $vid$, the target model $M$
Ensure: A set of view identifiers that are related to view $vid$
2: Step 1: Get Elements in Current View
3: Let $elements_{current} \leftarrow$ getElementsInView($vid, M$)
4: Let $related\_views \leftarrow \{vid\}$  ▷ Start with current view
5: Step 2: Find Views Sharing Elements
6: for each view $v$ in getAllViews($M$) do
7:     if $v \neq vid$ then  ▷ Skip the current view
8:         Let $elements_v \leftarrow$ getElementsInView($v, M$)
9:         if $elements_{current} \cap elements_v \neq \emptyset$ then
10:            $related\_views \leftarrow related\_views \cup \{v\}$  ▷ Direct element sharing
11:        end if
12:    end if
13: end for
14: Step 3: Find Views Connected via Consistency Rules
15: for each view $v$ in getAllViews($M$) do
16:     if $v \notin related\_views$ then  ▷ Only check views not already related
17:         Let $elements_v \leftarrow$ getElementsInView($v, M$)
18:         if shareConsistencyRule($elements_{current}$, $elements_v$, $M$) then
19:             $related\_views \leftarrow related\_views \cup \{v\}$  ▷ Connected via rules
20:         end if
21:     end if
22: end for
23: Step 4: Return Related Views
24: return $related\_views$
```

Algorithm 4. getRelatedViews.

Figure 7 shows the timestamp one, where all the aforementioned changes committed to the server as an INPUT tuple (e.g., added *resolution* attribute by Harry submitted as ***<(Client 1) (Structural View) (resolution attribute ID) (Add) (TS1)>***) to be checked by consistency oracle. Algorithm 1 shows the process of Evaluate operation inside the Consistency Oracle. Algorithm 2 shows the process of Query operation inside the Consistency Oracle, which complements the Evaluate algorithm. Algorithms 3, 4 present the required properties for Evaluate and Query algorithms as defined in Section 4.1. Moreover, Algorithm 5 provides the algorithmic details of Reorder operation for computing valid operation sequences from conflicting sets.

Here, the ***Evaluate((<(Client 1) (Structural View) (resolution attribute ID) (Add) (TS1)>), (Operation Tree), (CRDs))*** interface called to check the inconsistency of adding resolution attribute that successfully pass related CRDs. The *Evaluate* interface applies incremental checking (Property 1) by selecting only the consistency rules relevant to the changed elements. Furthermore, the ***Query((<(Client 1) (Structural View) (resolution attribute ID) (Add) (TS1)>), (Operation Tree), (VOD System Model))*** interface and lead to add it to the operation tree for the structural view. Multi-view coordination (Property 2) ensures

```
1: Reorder($O_{\text{conflict}}, T, \mathcal{R}$).
Require: A set of conflicting operations $O_{\text{conflict}}$,
    the current Operation Tree $T$, the set of all
    consistency rules $\mathcal{R}$.
Ensure: A sequence $O_{\text{valid}}$ (a permutation of $O_{\text{conflict}}$)
    that leads to a consistent state, or None if no
    such sequence exists, and feedback messages.

2: function Reorder($O_{\text{conflict}}, T, \mathcal{R}$)
3:   Let $M_{\text{current}} \leftarrow$ getCurrentModelState($T$)   ▷ Get state from the main branch of $T$
4:   Let Perms $\leftarrow$ generateAllPermutations($O_{\text{conflict}}$)
5:   Let feedback $\leftarrow \emptyset$
6:   for each permutation $P$ in Perms do ▷ Test each ordering
7:     Let $M_{\text{test}} \leftarrow$ copy($M_{\text{current}}$)
8:     $M_{\text{test}} \leftarrow$ applyOperations($M_{\text{test}}, P$) ▷ Apply $P$ to the model copy
9:     if isConsistent($M_{\text{test}}, \mathcal{R}$) then   ▷ Check if this order is valid
10:      return ($P, \emptyset$) ▷ Return the valid sequence and empty feedback
11:    end if
12:  end for
13:                    ▷ If no valid permutation found
14:  feedback $\leftarrow$ feedback $\cup$ {"No valid order found to resolve the conflict."}
15:  return (None, feedback)
16: end function
```

Algorithm 5. Reorder.

that changes in one view (e.g., the structural view) are propagated to related views when they share model elements. While, assessing the ***Evaluate((<(Client 2) (Streaming View) (title attribute ID) (Add) (TS1)>), (Operation Tree), (CRDs))*** and ***Evaluate((<(Client 2) (Streamer Interaction View) (pause() message ID) (Add) (TS1)>), (Operation Tree), (CRDs))*** lead to rejection of mentioned change operations and sending feedbacks to the relevant client.

In the second timestamp, as depicted in Figure 8, Bob received the addition of *resolution* attribute in the *Video* class. However, both of Bob's changes led to the inconsistencies based on CRD1 for adding *title* attribute and CRD2 and CRD3 rules for adding *pause()* message and rejected to the Bob with adequate feedbacks. Therefore, in the second timestamp, Bob decided to remove the *title* attribute of *Video* class and added *title* parameter to *play()* method of *Video* class and correct the direction of *pause()* message and added *stop()* message. Harry in parallel renamed method *play()* in *Video* class to *playVideo()*. After second timestamp and by renaming *play()* method before adding *title* parameter for *play()* method, the *Query* interface detects these contradicting change operations and reorder them to resolve the inconsistency. This reordering is performed by the *Reorder* algorithm (Algorithm 5), which tests permutations of the conflicting rename and parameter-addition operations. The algorithm finds that applying the rename ($play() \rightarrow playVideo()$) first, followed by the parameter addition ($playVideo() \rightarrow playVideo(title)$), yields a consistent state. This valid sequence is computed and applied automatically, demonstrating the integrated resolution mechanism. In addition, the *Evaluation* interface finds *stop()* method cannot pass CRD2, since there isn't any association from *Streamer* class to *Video* class.

Figure 9 shows third timestamp, where Bob added the required association and add again the *stop()* message. Figure 10 depicts the resulted VOD system model after merging changes applied on views and confirmed by consistency oracle and inconsistency resolution modules.

5 Evaluation

In Section 4, we elaborated on the details of COMIM and explained how our approach addresses inconsistency management in real-time collaborative modeling for multiple views. While the theoretical foundation of this approach demonstrates its potential to solve inconsistency management problems, its practical applicability requires further validation. This is a critical concern, as the scale of typical industrial projects can overwhelm experimental approaches, potentially rendering them ineffective.

For an approach to effectively support real-time collaboration, it must exhibit minimal delay and efficiently handle a large number of collaborators without performance degradation. To validate the applicability of our approach, we focus on three key aspects: (1) evaluating its applicability using Wind Turbine case study, (2) assessing its ability to detect various types of inconsistencies that may arise when engineers work concurrently, and (3) analyzing its performance, which we define as the computational efficiency and scalability of COMIM's inconsistency management mechanisms. Specifically, we measure the time required for inconsistency detection and resolution as model size and the number of concurrent collaborators increase.

To demonstrate the feasibility of our approach, we developed a proof-of-concept implementation. This prototype showcases how immediate inconsistency management can be utilized in real-time collaborative multi-view modeling. In the following sections, we provide a detailed discussion of each aspect of the evaluation.

5.1 Prototype implementation

To evaluate our approach, we implemented a prototype tool in Java as a server that supports COMIM infrastructure. In this section, we delve into the practical implementation of our proposed solution, providing insights into the development and deployment of the consistency management for multi-view collaborative modeling. We detail the architecture and functionalities of the COMIM, highlighting its ability to facilitate multi-model inconsistency detection without direct data exchange between the modeling tools and parties. The implementation is available on GitHub[1] under the Apache 2.0 license.

Figure 11 illustrates the internal structure of COMIM. The entry point to COMIM is the Consistency Server, which waits for

1 https://github.com/hayderalsharuee/COMIM

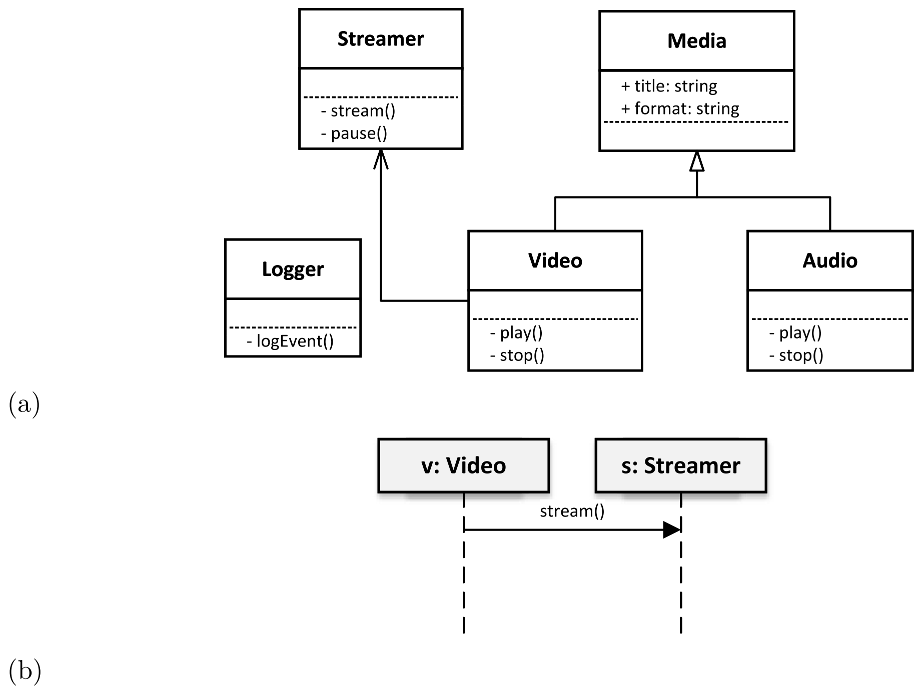

FIGURE 5
Two diagrammatic projections of the unified VOD system model. **(a)** Class diagram. **(b)** Sequence diagram.

Evaluate or *Query* consistency checking requests. In addition to the Consistency Server, COMIM includes a Consistency Oracle component. This component is initiated by the server and handles all preparation and checking steps. For an *Evaluate* request, it identifies the necessary rules and elements, while for a *Query* request, it creates the required data requests. The Consistency Oracle then calls the Rule Evaluator to evaluate the necessary rules or uses the Change Evaluator to check for equivalent or contradicting operations. To facilitate this, COMIM also includes a Data Requester component, which is responsible for sending data requests to the adapters and collecting the responses. Additionally, COMIM provides an interface for the Rule Evaluator and an implementation of it. Once the evaluation is complete, the Consistency Oracle returns the consistency results to the server. If inconsistencies are found, they are forwarded to the Inconsistency Resolution component, which prepares feedback or reordering decisions for the inconsistent operations.

5.2 Correctness and applicability

We evaluated the correctness of our approach regarding applicability through the Wind Turbine case study (Gómez et al., 2020). The Wind Turbine case study focuses on the design and management of a cooling control system for a wind turbine generator. The system includes input temperature sensors, output signals, and controller units that manage fans and brakes. System parameters define temperature limits that trigger the cooling system.

In an online collaboration senario, all stakeholders work in real-time, and changes are propagated immediately to ensure that everyone has access to the most up-to-date version of the model. While this approach minimizes conflicts, it introduces challenges related to synchronization and inconsistency resolution. For example, if two stakeholders (e.g., the WT Manager and System Manager) make contradicting changes to the same model element simultaneously, the system must detect and resolve these inconsistencies immediately. If both modify the temperature limits at the same time, the system should flag the inconsistency and require on-demand resolution.

Temporary inconsistencies may also arise during the brief period between a change being made and propagated. For instance, if the IO Manager removes the *GeneratorTemperature* input, there may be a short delay before the WT Manager and System Manager see the updated model. Additionally, if a stakeholder makes a change without realizing that another stakeholder has already modified the same element, the second change might overwrite the first. For example, if the System Manager updates a parameter while the WT Manager is also working on it, the WT Manager's changes might be lost if not properly synchronized. Therefore, effective inconsistency management is essential to ensure that all stakeholders work with a consistent and up-to-date model view, preventing errors and improving collaboration efficiency. In this case study, personalization was utilized in both dimensions. Each stakeholder (e.g., WT Manager, IO Manager) worked within a personalized view of the wind turbine model, configured to show only the elements relevant to their role. Furthermore, the inconsistency management was configured with project-specific consistency rules and scopes, enabling the system to detect and

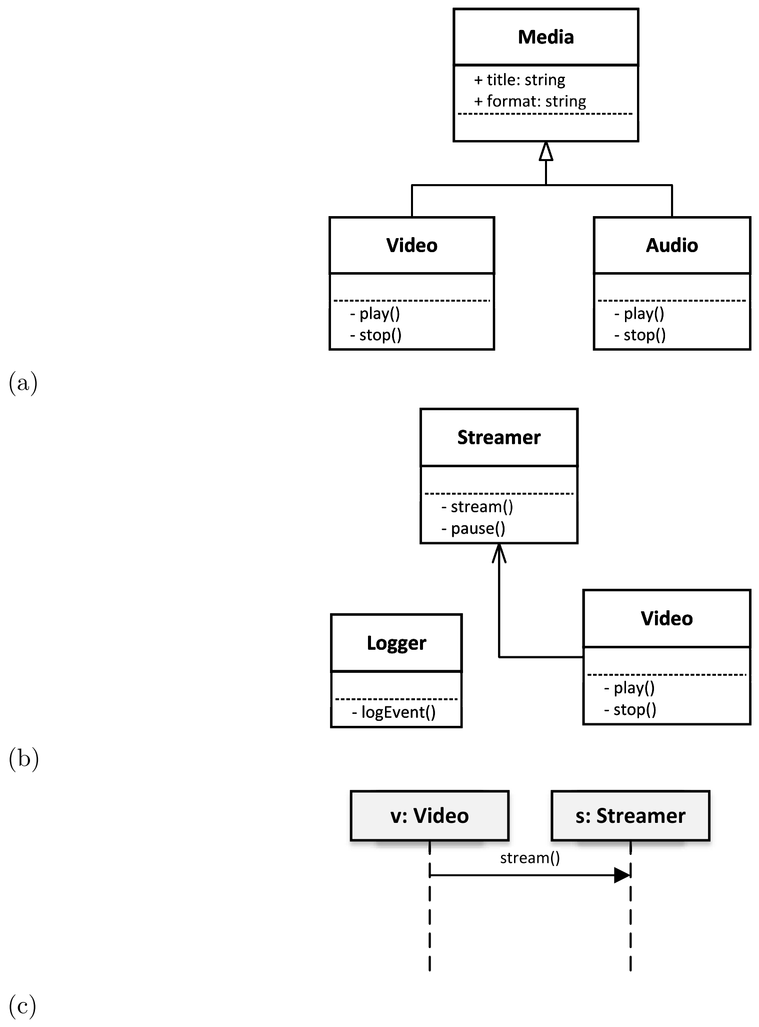

FIGURE 6
Three personalized views defined on the unified VOD system model. **(a)** Structural view. **(b)** Streaming view. **(c)** Streamer interaction view.

```
inherited attribute
self.ownedAttribute->forAll(subAttr |
    {not} self.allInheritedAttributes()->exists(
   superAttr | superAttr.name = subAttr.name ))
```

Listing 1 (CRD1) A subclass cannot declare an attribute with the same name of its superclass.

```
context Message inv:
   self.receiveEvent -> exists ( lr : Lifeline |
   self.sendEvent->exists( ls : Lifeline |
      ls.attributes->exists (a: Attribute |
      not (a = null) implies a.type= lr.type)))
```

Listing 2 (CRD2) The Message direction must match the Class Association direction.

```
context Message inv:
   self.receiveEvent->exists( lr : Lifeline |
   self.receiveEvent->forAll( l : Lifeline |
      l.operations->exists (o:Operation |
      o.name = self .name))
```

Listing 3 (CRD3) The Message must be defined as an Operation in the Receiving Class.

resolve conflicts in a manner tailored to the project's engineering context.

In the Wind Turbine case study, the COMIM framework was evaluated through four collaborative scenarios involving three stakeholders working on three different views of a model with an average of 5,441 elements. Across these scenarios, the system detected all 14 inconsistencies that arose. Of these, four inconsistencies (35.7%) were automatically resolved through operation reordering by the Consistency Oracle, while eight inconsistencies (57.1%) required user feedback for manual resolution. The remaining inconsistency (0.7%) were semantical conflict that were prevented by immediate validation before commit. These results demonstrate COMIM's effectiveness in both detecting and resolving inconsistencies in a real-world, multi-view collaborative setting, with a significant portion of conflicts being resolved automatically through reordering.

The aim of this case study was to evaluate whether the solution meets the most ambitious target measures set by industrial needs. Initially, three different users—WT Manager, IO

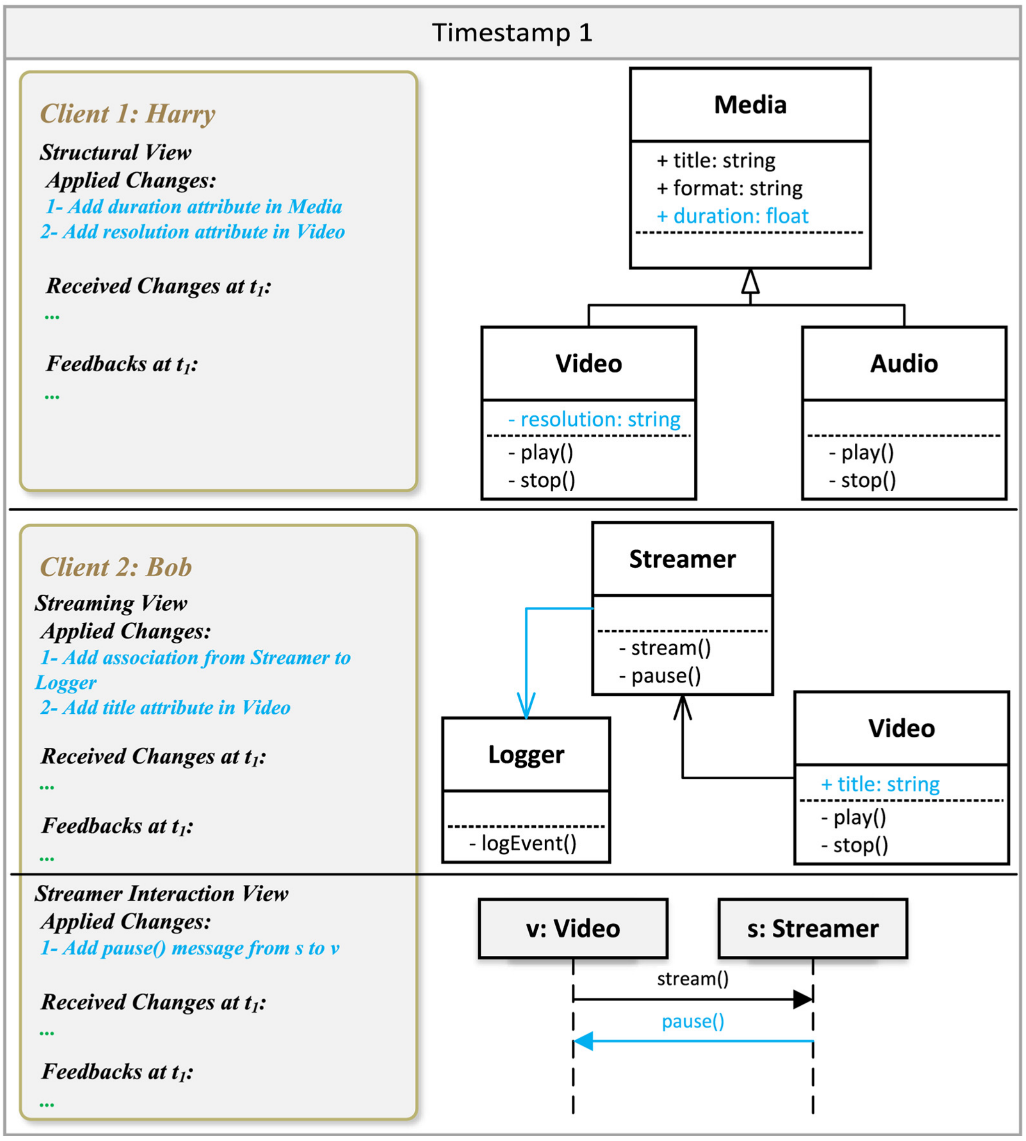

FIGURE 7
An snapshot of the first timestamp.

Manager, and System Manager—participated in the scenario. To further test the system, we expanded the number of participants to seven by adding a Principal Engineer, Pitch Manager, Converter Manager, and Generator Manager. All seven users worked in the same shared modeling session. As a result, every change made by one user was automatically propagated to all other users, and their editors were refreshed in real-time to display the updated model. This propagation is managed by the Collaboration Server (Figure 1), with consistency validated via the Evaluate interface of the Consistency Oracle (Figure 4).

From the analysis of the measures, we observed that increasing the number of collaborators with common scopes led to a longer time required for inconsistency evaluation. However, for consistent changes, operations were automatically propagated to all relevant users, and their editors were refreshed immediately to show the updated view. To assess inconsistency management, various sets of changes were applied to the models to determine whether the type and amount of changes significantly impacted response time and view consistency. The results indicate that the type of changes had no significant effect on the accuracy of the inconsistency oracle. However, the amount of detected inconsistencies did affect the time required to propagate changes to all views. Consequently, the volume of inconsistencies directly impacted the time needed to synchronize changes across all views, while the type of changes had no significant effect on the inconsistency oracle's accuracy.

5.3 Inconsistency management evaluation

To assess the inconsistency management of our approach, we utilized the benchmark (Langer and Wimmer, 2013) presented

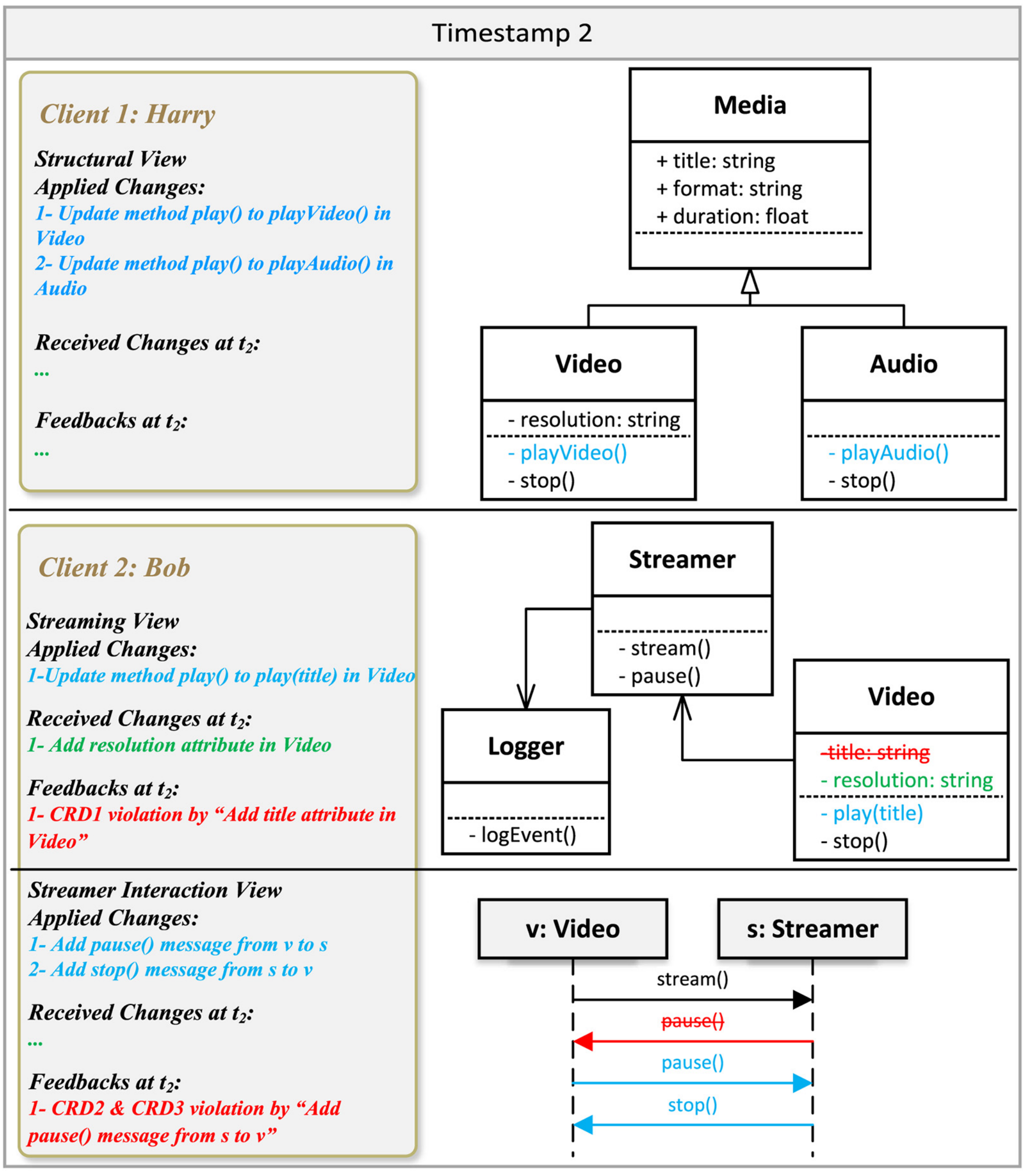

FIGURE 8
An snapshot of the second timestamp.

in the AMOR project. This benchmark is designed to evaluate the accuracy of our consistency oracle component and assess COMIM's capability to manage contradicting changes while producing a consistent model. It comprises 23 test scenarios (12 with inconsistency and 11 without inconsistency) aimed at measuring the precision of detecting inconsistencies during the operation-based evolution of an Ecore metamodel. These test scenarios are derived from the collaborative conflict lexicon and encompass a variety of inconsistency types arising from atomic change operations, including concurrent updates, conflicting delete/update, update/update, add/add, and delete/add operations. Among these, 11 test cases are designed to produce no inconsistencies, serving as control scenarios.

The benchmark's 12 inconsistent test scenarios encompass three primary conflict types derived from concurrent, operation-based edits: (1) Semantic Inconsistencies (two scenarios), where operations violate domain-specific logic; (2) Contradicting Changes (six scenarios), such as concurrent updates to the same attribute with incompatible values or conflicting delete/update operations; and (3) Constraint Violations (four scenarios), where operations break structural metamodel constraints.

The results demonstrate that COMIM successfully detected all 12 expected inconsistencies across the 23 test scenarios, achieving 100% accuracy in inconsistency management. Additionally, the benchmark confirmed that COMIM consistently generated valid tree operations for the target model in all 23 scenarios, ensuring

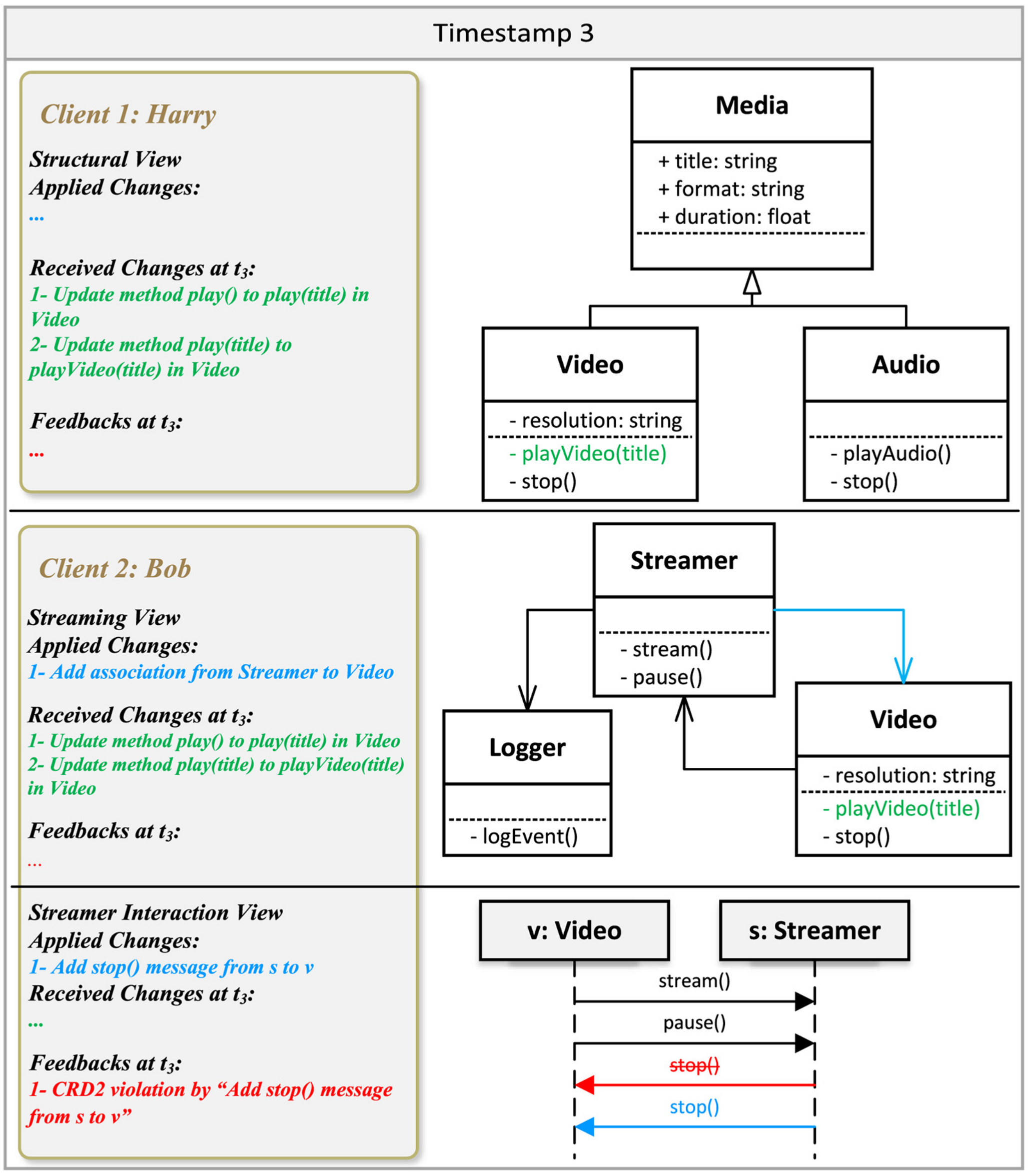

FIGURE 9
An snapshot of the third timestamp.

the final model remained consistent. These findings highlight COMIM's effectiveness in handling contradicting changes and maintaining model consistency.

5.4 Performance

The applicability of our approach hinges on its performance scalability. Scalability can be evaluated by analyzing various system attributes, such as memory usage, network usage, and CPU usage. In this work, however, we focus our scalability evaluation on the algorithmic and computational ability of our approach.

The performance scalability of consistency checking, implemented via the Evaluate interface (Algorithm 1), is primarily influenced by the number of required evaluations, which is minimized by our scoping mechanism (Property 1). Changes that trigger inconsistency oracle evaluations are limited to the creation and modification of model elements—deletions do not necessitate re-evaluations. As noted by Egyed (2010), prior validations of the Model/Analyzer indicate that the average number of evaluations per change ranges between 3 and 11, depending on the size of the model.

To empirically evaluate the overhead introduced by our approach, we replicated an evaluation of the Model/Analyzer using our method. For this evaluation, we utilized the same UML models described by Reder and Egyed (2012b), which include 30 UML models from both academic and industrial sources, ranging in size from 90 to 64,061 model elements. We simulated random changes to all model elements within each model and measured the time required to re-evaluate all affected CREs and persist the updated results. The 30 models used in this evaluation represent diverse Ecore-based models from multiple domains, including

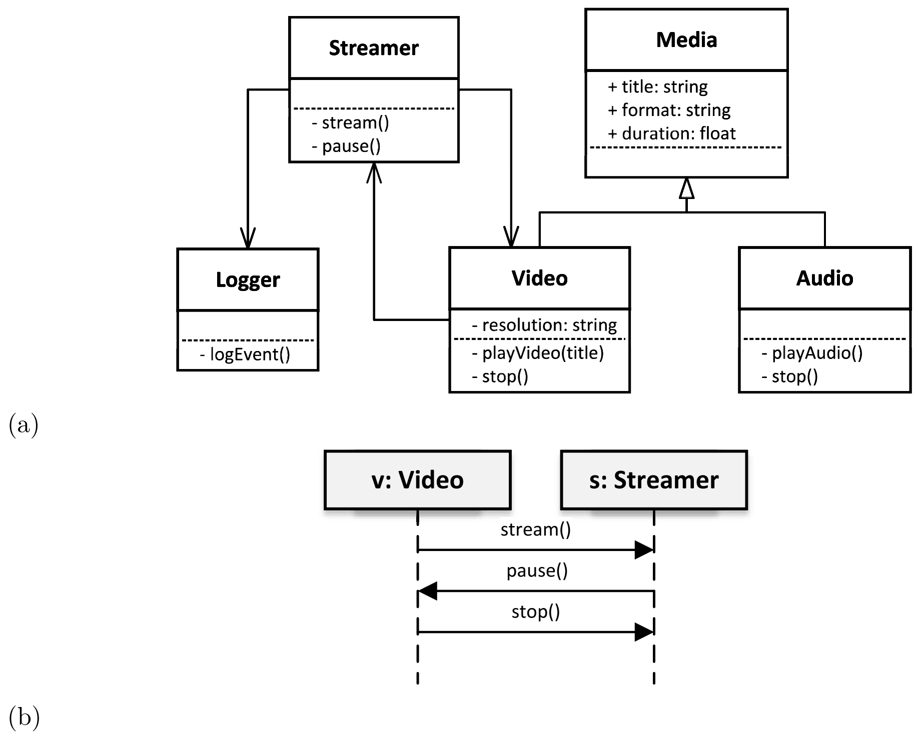

FIGURE 10
The unified model of VOD system after timestamp 3. **(a)** The resulted class diagram. **(b)** The resulted sequence diagram.

UML models and domain-specific languages, demonstrating the framework's language-agnostic nature and the scalability of its scoping mechanism.

Each change was executed multiple times, and the raw data for each iteration was recorded. The experiments were conducted on a Windows 10 Professional PC equipped with an Intel(R) Core(TM) i7-8665U CPU @ 2.1GHz and 32GB of RAM. Table 1 summarizes the results obtained from applying the experiments. The results demonstrate that while model size has a noticeable impact on evaluation time, this effect is relatively small. Additionally, the findings reveal that neither the number of affected CREs nor the evaluation time per CRE individually significantly influences the total processing time. However, when considered together, these factors become key determinants of the overall processing time.

Table 1 demonstrates that COMIM's performance is robust to variations in model size and rule composition. While evaluation time per instance increases modestly (0.50–0.84 ms) across models ranging from 90 to 64,061 elements, this growth is sub-linear relative to model scale. Notably, models with similar sizes but differing rule counts [e.g., "ANTS Visualizer" (1,282 elements, 125 rules) vs. "Inventory and Sales" (1,296 elements, 81 rules)] show comparable latency, confirming that our scoping mechanism effectively isolates rule evaluation to relevant changes. This validates COMIM's two key advantages: (1) checking overhead depends primarily on the scope of modifications rather than total model size, and (2) rule extensibility does not compromise performance, as only active rules for modified elements are triggered. These properties ensure scalability for both large-scale models and evolving rule sets in practice.

5.5 Threats to validity

This section discusses internal, external, and conclusion threats to validity and outlines how we mitigated them, following the guidelines provided by Wohlin et al. (2012).

5.5.1 Internal validity

Internal threats primarily concern the selection of models used in the evaluation. For instance, using only small or large models could lead to an over- or under-representation of inconsistencies, which could also affect repair generation. To mitigate this threat, we utilized 30 UML models and four Java systems, with model sizes ranging from 100 to 64,061 elements. This resulted in a range of 3–1,000 inconsistencies per model, totaling 3,953 inconsistencies across all models. The diversity in model sizes and the resulting inconsistency counts demonstrate that the selected models are sufficiently varied to support our findings.

Another internal threat relates to the consistency rules used in the evaluation, as the size and complexity of the evaluation and repair trees may be influenced by these rules. To address this, we defined a diverse set of rules with varying sizes, expression types, and evaluation contexts. The results, particularly the number of evaluations per model, support our claim that this threat has been effectively mitigated. A further internal threat concerns the scope of detectable inconsistencies. Our evaluation focused on inconsistency types expressible as OCL constraints within the defined case studies and benchmark (e.g., syntactic conflicts, structural violations). While this covers common collaborative

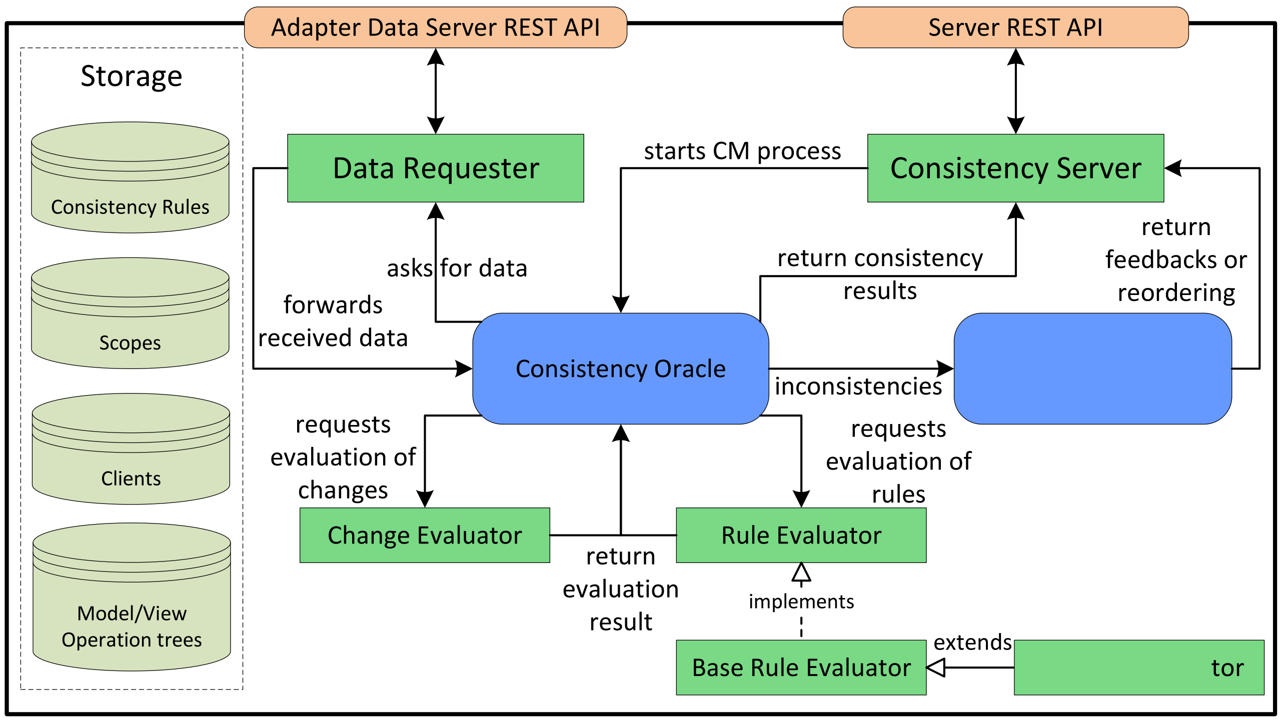

FIGURE 11
Overview of the internal structure of the COMIM prototype implementation.

conflicts, the framework's effectiveness for more complex, domain-specific semantic inconsistencies, which may require richer reasoning or external knowledge, represents a threat to the internal validity of claims about general inconsistency management.

5.5.2 External validity

External threats pertain to the generalizability of our findings. One such threat is the focus on a single industrial case study, which imposed constraints on the initial selection of frameworks. As a result, the findings from this case study should not be generalized to the broader population of modeling tools and technologies. To mitigate this, we incorporated 30 UML models from diverse domains in the computational complexity evaluation. While this evaluation used UML models and Java source code, limiting the evidence to these domains, the successful application of our approach to Ecore and UML models suggests its potential for extension to other domains.

Two additional threats to external validity concern practical adoption: First, the effort required to define a comprehensive set of Consistency Rule Definitions (CREs) depends on domain complexity and could be substantial, potentially limiting uptake in practice. Second, COMIM is designed for intra-model inconsistency management across views of a single model. Its mechanisms and evaluation do not address inter-model consistency (consistency across different, related models), which is a common challenge in heterogeneous toolchains. This limits the generalizability of our findings to that broader context.

5.5.3 Conclusion validity

A key threat to conclusion validity lies in the scenarios considered for evaluating inconsistency management. To address this, we employed a benchmark comprising 23 purposefully designed modeling scenarios that cover a wide range of conflicting situations. While we believe these scenarios encompass the most common cases of inconsistency detection, further investigation of the consistency oracle component through additional scenarios is recommended in future work. Furthermore, our evaluation of the resolution mechanism showed that not all inconsistencies can be resolved automatically; some require user feedback. The distribution between auto-resolved and user-resolved conflicts in our case study is specific to that scenario, and the general effectiveness of the reordering strategy across all inconsistency types remains a threat to conclusions about fully automated resolution.

6 Conclusion

This paper presents COMIM, a configurable approach for managing intra-model inconsistencies in real-time collaborative editing of a single model through multiple views. By leveraging MDE principles, COMIM introduces a structured framework that supports real-time change propagation, incremental consistency checking, and operation-based resolution. The framework addresses the challenges of inconsistency management by integrating a consistency oracle, enabling efficient detection and resolution of inconsistencies

TABLE 1 Performance evaluation results for models of varying sizes.

Name	#Model element	#Consistency rule	Evaluation time per affected instance (m)
Video on demand	90	6	0.50
ATM	220	30	0.53
Microwave oven	290	38	0.52
Model view controller	418	39	0.59
eBullition	513	41	0.59
Curriculum	763	59	0.52
Teleoperated robot	1,115	85	0.57
Dice 3	1,274	99	0.57
ANTS visualizer	1,282	125	0.58
Inventory and sales	1,296	81	0.57
Course registration	1,406	73	0.61
UML IOC F05a T12	1,453	98	0.65
VOD 3	1,558	183	0.66
Vacation and sick leave	1,658	104	0.66
Home appliance	1,707	84	0.65
HDCP defect seeding	1,784	98	0.67
DESI 2.3	1,974	208	0.67
iTalks	2,212	289	0.68
Hotel management sys.	2,583	202	0.67
Biter robocup	2,632	334	0.69
Calendarium	2,809	294	0.68
UML LCA F03a T1	2,983	143	0.69
<unnamed>	5,373	292	0.73
NPI	7,110	303	0.75
Word pad	8,078	318	0.75
dSpace 3.2	8,761	205	0.76
OODT	9,828	406	0.79
Insurance network fees	16,255	425	0.81
<unnamed>	33,347	451	0.82
<unnamed>	64,061	489	0.84

across multiple views. Evaluations using case studies and benchmarks demonstrate COMIM's effectiveness in handling contradicting changes and maintaining model consistency, with empirical results confirming scalable performance for teams of ten concurrent users and models exceeding 60,000 elements.

Future research could explore extending COMIM to support additional modeling languages and domains beyond UML and Ecore, further enhancing its versatility. We plan to extend COMIM to handle a broader range of inconsistency types, including those requiring complex domain reasoning, by integrating more sophisticated resolution strategies. To reduce the effort of writing consistency rules, we will investigate techniques for automatic rule mining from model histories. Investigating advanced machine learning techniques to automate inconsistency resolution and improve the efficiency of conflict detection could also be beneficial. Additionally, integrating COMIM with blockchain technology for secure and transparent collaboration in distributed environments presents an exciting avenue for exploration. Finally, conducting large-scale industrial case studies would provide deeper insights into COMIM's applicability and performance in real-world scenarios, helping to refine and optimize the framework for broader adoption.

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

HA: Software, Writing – original draft, Formal analysis, Visualization, Validation. MS: Validation, Conceptualization, Investigation, Writing – review & editing, Methodology. BT: Writing – review & editing, Investigation, Supervision, Conceptualization, Project administration.

Funding

The author(s) declared that financial support was not received for this work and/or its publication.

Conflict of interest

The author(s) declared that this work 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) declared that generative AI was not 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

References

Balzer, R. (1991). "Tolerating inconsistency (software development)," in *Proceedings-13th International Conference on Software Engineering* (Austin, TX: IEEE Computer Society), 158–159. doi: 10.1109/ICSE.1991.130638

Blanc, X., Mounier, I., Mougenot, A., and Mens, T. (2008). "Detecting model inconsistency through operation-based model construction," in *Proceedings of the 30th International Conference on Software Engineering* (New York, NY: ACM), 511–520. doi: 10.1145/1368088.1368158

Boehm, B. W., Brown, J., and Lipow, M. (1976). "Quantitative evaluation of software quality," in *Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research* (San Francisco, CA: IEEE), 25.

Briand, L., Devanbu, P., and Melo, W. (1997). "An investigation into coupling measures for C++," in *Proceedings of the 19th International Conference on Software Engineering* (New York, NY: ACM), 412–421. doi: 10.1145/253228.253367

Burger, E. (2014). *Flexible Views for View-Based Model-Driven Development, Volume 15*. Karlsruhe: KIT Scientific Publishing.

Cabot, J., and Gogolla, M. (2012). "Object constraint language (OCL): a definitive guide," in *International School on Formal Methods for the Design of Computer, Communication and software Systems* (Cham: Springer), 58–90. doi: 10.1007/978-3-642-30982-3_3

Cicchetti, A., Ciccozzi, F., and Leveque, T. (2011). "Supporting incremental synchronization in hybrid multi-view modelling," in *International Conference on Model Driven Engineering Languages and Systems* (Cham: Springer), 89–103. doi: 10.1007/978-3-642-29645-1_11

Dam, H. K., and Winikoff, M. (2011). An agent-oriented approach to change propagation in software maintenance. *Auton. Agent Multi. Agent Syst*. 23, 384–452. doi: 10.1007/s10458-010-9163-0

David, I., Vangheluwe, H., and Syriani, E. (2023). Model consistency as a heuristic for eventual correctness. *J. Comput. Lang.* 76:101223. doi: 10.1016/j.cola.2023.101223

Egyed, A. (2006). "Instant consistency checking for the UML," in *Proceedings of the 28th International Conference on Software Engineering* (New York, NY: ACM), 381–390. doi: 10.1145/1134285.1134339

Egyed, A. (2010). Automatically detecting and tracking inconsistencies in software design models. *IEEE Trans Softw. Eng.* 37, 188–204. doi: 10.1109/TSE.2010.38

Egyed, A., Letier, E., and Finkelstein, A. (2008). "Generating and evaluating choices for fixing inconsistencies in uml design models," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC: IEEE), 99–108. doi: 10.1109/ASE.2008.20

Feldmann, S., Kernschmidt, K., Wimmer, M., and Vogel-Heuser, B. (2019). Managing inter-model inconsistencies in model-based systems engineering: application in automated production systems engineering. *J. Syst. Softw.* 153, 105–134. doi: 10.1016/j.jss.2019.03.060

Franzago, M., Di Ruscio, D., Malavolta, I., and Muccini, H. (2017). Collaborative model-driven software engineering: a classification framework and a research map. *IEEE Trans. Softw. Eng.* 44, 1146–1175. doi: 10.1109/TSE.2017.2755039

Gómez, A., Mendialdua, X., Barmpis, K., Bergmann, G., Cabot, J., De Carlos, X., et al. (2020). Scalable modeling technologies in the wild: an experience report on wind turbines control applications development. *Softw. Syst. Model* 19, 1229–1261. doi: 10.1007/s10270-020-00776-8

Grundy, J., Hosking, J., and Mugridge, W. B. (1998). Inconsistency management for multiple-view software development environments. *IEEE Trans. Softw. Eng.* 24, 960–981. doi: 10.1109/32.730545

Herzig, S. J., Qamar, A., and Paredis, C. J. (2014). An approach to identifying inconsistencies in model-based systems engineering. *Procedia Comput. Sci.* 28, 354–362. doi: 10.1016/j.procs.2014.03.044

Júnior, A. A., Misra, S., and Soares, M. S. (2019). "A systematic mapping study on software architectures description based on ISO/IEC/IEEE 42010: 2011," in *Computational Science and Its Applications-ICCSA 2019: 19th International Conference, Saint Petersburg, Russia, July 1-4, 2019, Proceedings, Part V 19* (Cham: Springer), 17–30. doi: 10.1007/978-3-030-24308-1_2

Karagiannis, D., Buchmann, R. A., and Bork, D. (2016). *Managing Consistency in Multi-view Enterprise Models: An Approach Based on Semantic Queries* (Istanbul: European Conference on Information Systems).

Kleiner, M., and Roucoules, L. (2025). Farfadet: contribution to consistency management in multi-physical systems engineering. *Int. J. Comput. Integr. Manuf.* 38, 1870–1887. doi: 10.1080/0951192X.2025.2461024

Knapp, A., and Mossakowski, T. (2018). "Multi-view consistency in UML: a survey," in *Graph Transformation, Specifications, and Nets: In Memory of Hartmut Ehrig* (Cham: Springer), 37–60. doi: 10.1007/978-3-319-75396-6_3

Kolovos, D. S. (2009). "Establishing correspondences between models with the epsilon comparison language," in *European Conference on Model Driven Architecture-Foundations and Applications* (Cham: Springer), 146–157. doi: 10.1007/978-3-642-02674-4_11

Langer, P., and Wimmer, M. (2013). A benchmark for conflict detection components of model versioning systems. *Softwaretechnik-Trends* 33, 91–94. doi: 10.1007/s40568-013-0060-y

Lucas, F. J., Molina, F., and Toval, A. (2009). A systematic review of uml model consistency management. *Inf. Softw. Technol.* 51, 1631–1645. doi: 10.1016/j.infsof.2009.04.009

Macedo, N., Jorge, T., and Cunha, A. (2016). A feature-based classification of model repair approaches. *IEEE Trans. Softw. Eng.* 43, 615–640. doi: 10.1109/TSE.2016.2620145

Mistrík, I., Grundy, J., Van der Hoek, A., and Whitehead, J. (2010). *Collaborative Software Engineering: Challenges and Prospects*. Cham: Springer. doi: 10.1007/978-3-642-10294-3

Muram, F. U., Tran, H., and Zdun, U. (2017). Systematic review of software behavioral model consistency checking. *ACM Comput. Surv*. 50, 1–39. doi: 10.1145/3037755

Nentwich, C., Capra, L., Emmerich, W., and Finkelsteiin, A. (2002). Xlinkit: a consistency checking and smart link generation service. *ACM Trans. Internet Technol*. 2, 151–185. doi: 10.1145/514183.514186

Nentwich, C., Emmerich, W., and Finkelstein, A. (2003). "Consistency management with repair actions," in *25th International Conference on Software Engineering, 2003. Proceedings* (Portland, OR: IEEE), 455–464. doi: 10.1109/ICSE.2003.1201223

Ohrndorf, M., Pietsch, C., Kelter, U., Grunske, L., and Kehrer, T. (2021). History-based model repair recommendations. *ACM Trans. Softw. Eng. Methodol*. 30, 1–46. doi: 10.1145/3419017

Ortega, M., Pérez, M., and Rojas, T. (2003). Construction of a systemic quality model for evaluating a software product. *Softw. Qual. J*. 11, 219–242. doi: 10.1023/A:1025166710988

Persson, M., Loiret, F., Westman, J., Chen, D.-J., T́orngren, M., and Biehl, M. (2013). *Multi-Viewed components*. Available online at: https://www.diva-portal.org/smash/record.jsf?pid=diva2:622666 (Accessed October 10, 2025).

Puissant, J. P., Mens, T., and Van Der Straeten, R. (2010). "Resolving model inconsistencies with automated planning," in *Proceedings of the 3rd Workshop on Living with Inconsistencies in Software Development (LWI/SE 2010)* (Antwerp), 8–14.

Reder, A., and Egyed, A. (2012a). "Computing repair trees for resolving inconsistencies in design models," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY: ACM), 220–229. doi: 10.1145/2351676.2351707

Reder, A., and Egyed, A. (2012b). "Incremental consistency checking for complex design rules and larger model changes," in *Model Driven Engineering Languages and Systems: 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings 15* (Cham: Springer), 202–218. doi: 10.1007/978-3-642-33666-9_14

Rumpe, B. (2016). *Modeling with UML, Vol. 98*. Cham: Springer. doi: 10.1007/978-3-319-33933-7

Schröpfer, J., Schwägerl, F., and Westfechtel, B. (2019). "Consistency control for model versions in evolving model-driven software product lines," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)* (Munich: IEEE), 268–277. doi: 10.1109/MODELS-C.2019.00043

Shahvari, Y., Sharbaf, M., Rahimi, S. K., and Tehrani, S. Y. (2024). "Towards agile collaborative multi-view modeling with inconsistency tolerance," in *Proceedings of the STAF 2024 Workshops: AgileMDE* (Enschede).

Sharbaf, M., Zamani, B., and Sunyé, G. (2020). "A formalism for specifying model merging conflicts," in *Proceedings of the 12th System Analysis and Modelling Conference* (New York, NY: ACM), 1–10. doi: 10.1145/3419804.3421447

Sharbaf, M., Zamani, B., and Sunyé, G. (2022). Automatic resolution of model merging conflicts using quality-based reinforcement learning. *J. Comput. Lang*. 71:101123. doi: 10.1016/j.cola.2022.101123

Sharbaf, M., Zamani, B., and Sunyé, G. (2023). Conflict management techniques for model merging: a systematic mapping review. *Softw. Syst. Model* 22, 1031–1079. doi: 10.1007/s10270-022-01050-9

Sharbaf, M., Zamani, B., and Sunyé, G. (2025). Compers: a configurable conflict management framework for personalized collaborative modeling. *J. Syst. Softw*. 219:112227. doi: 10.1016/j.jss.2024.112227

Spanoudakis, G., and Zisman, A. (2001). "Inconsistency management in software engineering: Survey and open research issues," in *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals* (London: World Scientific), 329–380. doi: 10.1142/9789812389718_0015

Stevens, P. (2020). Maintaining consistency in networks of models: bidirectional transformations in the large. *Softw. Syst. Model* 19, 39–65. doi: 10.1007/s10270-019-00736-x

Torres, W., Van den Brand, M. G., and Serebrenik, A. (2021). A systematic literature review of cross-domain model consistency checking by model management tools. *Softw. Syst. Model* 20, 897–916. doi: 10.1007/s10270-020-00834-1

Vogel-Heuser, B., and Zou, M. (2019). Leveraging inconsistency management in the multi-view collaborative modelling of cyber-physical production systems. *IET Collab. Intell. Manuf*. 1, 126–129. doi: 10.1049/iet-cim.2019.0019

Wen, H., Wu, J., Jiang, J., Tang, G., and Hong, Z. (2023). A formal approach for consistency management in uml models. *Int. J. Softw. Eng. Knowl. Eng*. 33, 733–763. doi: 10.1142/S0218194023500134

Whittle, J., Hutchinson, J., and Rouncefield, M. (2013). The state of practice in model-driven engineering. *IEEE Softw*. 31, 79–85. doi: 10.1109/MS.2013.65

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Cham: Springer Science & Business Media. doi: 10.1007/978-3-642-29044-2

Wust, J. (2006). *Sdmetrics: The Software Design Metrics Tool for UML*. Available online at: https://www.sdmetrics.com/index.html (Accessed October 10, 2025).

Xiong, Y., Hu, Z., Zhao, H., Song, H., Takeichi, M., Mei, H., et al. (2009). "Supporting automatic model inconsistency fixing," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (New York, NY: ACM), 315–324. doi: 10.1145/1595696.1595757

Yu, E., and Choi, J. (2023). Development of building information modeling-based automation assessment process for universal design of public buildings. *J. Comput. Des. Eng*. 10, 641–654. doi: 10.1093/jcde/qwad018