

Real-time Collaborative Multi-Level Modeling by Conflict-Free Replicated Data Types

Istvan David · Eugene Syriani

Received: date / Accepted: date

Abstract The need for real-time collaborative solutions in model-driven engineering has been increasing over the past years. Conflict-free replicated data types (CRDT) provide scalable and robust replication mechanisms that align well with the requirements of real-time collaborative environments. In this paper, we propose a real-time collaborative multi-level modeling framework to support advanced modeling scenarios, built on a collection of custom CRDTs, specifically tailored for the needs of modeling environments. We demonstrate the benefits of the framework through an illustrative modeling case and compare it with other state-of-the-art modeling frameworks.

Keywords Collaborative modeling, Real-time collaboration, Multi-level modeling, Conflict-free replicated data types, Model-driven engineering

1 Introduction

Collaborative Model-Driven Software Engineering (MDSE) [38] aims to establish a sound interplay between physically distanced stakeholders by combining the techniques of collaborative software engineering [60] and model-driven techniques [44]. Recent systematic studies [14, 13, 23] report a substantial shift towards real-time collaboration in MDSE. While real-time collaborative MDSE opens up many opportunities, it also gives rise to unique challenges, primarily: ensuring appropriate convergence of distributed data, while still guaranteeing timely execution [49].

Optimistic replication has been suggested by Saito and Shapiro [43] as a possible treatment, in which the

local replicas of stakeholders are allowed to diverge temporarily. This divergence is admissible due to eventual consistency mechanisms [59] ensuring that each remote change will be observed by every stakeholder eventually; thus, enabling the converge of replicas in the long run. Strong eventual consistency (SEC) [39] augments eventual consistency with the safety guarantee that two elements that have received the same set of change updates will be in the same state, regardless of the order of updates. Although SEC aligns well with the requirements of real-time collaborative MDSE settings, it is not trivial to implement correctly [18].

Conflict-free Replicated Data Types (CRDT) have been suggested by Shapiro et al [46] as a scalable implementation of SEC. CRDTs have been traditionally geared to support linear data, such as text. Since traditional software engineering relies on textual artifacts to persist source code, mechanisms of real-time collaborative *textual* editors can address the main challenges of real-time collaborative source code development. However, MDSE relies on richer data types, such as multi-graphs, that are also potentially disconnected. Thus, CRDT cannot efficiently accommodate *models* as first-class citizens. Existing solutions either (i) focus exclusively on textual modeling and reduce collaboration to textual primitives which are well-supported by current CRDT frameworks [42]; or (ii) work on models of limited complexity and do not support proper graph semantics at the data level [20]. As a consequence, current CRDT-based techniques fall short of supporting intricate modeling scenarios, such as multi-level modeling [3].

Augmenting multi-level modeling with real-time collaboration capabilities enables teamwork between stakeholders acting (i) at different levels of abstraction, or (ii) at different levels of decision making. Typical examples

include (i) changing a modeling language during operation to incorporate language elements required by the technical stakeholders [28]; and (ii) restricting values of attributes at higher levels of decision making and enforcing these values through the notion of potency [55].

The main **contribution** of our work is a novel real-time collaborative multi-level modeling framework, called *lowkey*¹. The framework supports a wide range of modeling scenarios across an arbitrary number of modeling and linguistic meta-levels. We achieve this flexibility by separating the linguistic metamodel(s) from the physical metamodel. In this setting, domain-specific models always conform to their linguistic metamodels, and they both (models and metamodels) conform to the uniform physical metamodel provided by the framework. Real-time synchronization between collaborating stakeholders is achieved by persisting the physical metamodel in CRDTs that ensure the consistency of domain-specific models in a domain-agnostic fashion. This makes *lowkey* especially suitable for supporting approaches such as multi-view modeling [40]. Furthermore, *lowkey* subsumes traditional modeling frameworks, such as UML [22] and EMF [48], offering a promising integration potential with existing model editors.

2 Illustrative case

We rely on an illustrative case of developing a collaborative editor for modeling Mind maps [10]. The case highlights multiple structural facets of (meta)modeling, such as typing, inheritance, and various forms of well-formedness through the collaborative modeling of mind maps. We show how an editor built on top of *lowkey* would enable this.

The metamodel of the case is shown in Figure 1. *MindMap* serves as the root element, containing the various types of *Topics*, and the *Markers* that can be associated with specific *Topics*. The mindmap is hierarchical: the single *CentralTopic* further contains an arbitrary number of *MainTopics*, each containing an arbitrary number of *SubTopics*. Finally, *SubTopics* can contain *SubTopics* to arbitrary depths.

Based on the metamodel, the language engineer must provide an editor that enables real-time collaborative modeling of a *MindMap* instance. This entails the following operations: creating, editing, and deleting instances of metamodel concepts; creating, editing, and deleting links between elements. In addition, a mechanism for reading the state of the local model is required. Here are some modeling scenarios that may occur as a result of applying these CRUD operations.

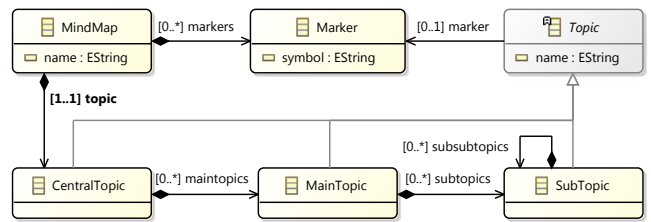


Fig. 1 The metamodel of the Mind maps.

Cooperation. User A creates a *MindMap* instance and User B sets its *title*.

Cooperation with of linguistic inconsistencies.

User A creates a *CentralTopic* instance, but does not link it to the *MindMap* instance. User B links the *CentralTopic* instance to the *MindMap* via a composition reference.

Conflict. User B removes the *CentralTopic* instance; in parallel, however, User A creates a *MainTopic* instance and links it to the *CentralTopic* instance.

Multi-level cooperation. Users A and B have created multiple *Markers*, and would like to categorize them (e.g., as textual and graphical). The Language Designer, who works one metalevel above Users A and B, sets the *potency* of the *Marker* type from one to two; thus, allowing a templating mechanism for A and B.

3 Background

We discuss the existing work related to collaborative modeling.

3.1 Collaborative MDSE

Collaborative software engineering enables effective cooperation among stakeholders [60], often in distributed settings [27]. Distributed teams introduce challenges to collaboration in terms of processes, project management, artifact sharing, and consistency [36]. These challenges are further exacerbated in the engineering of complex software-intensive systems, such as cyber-physical systems, that require collaboration between stakeholders of highly diverse expertise. MDSE [44] provides stakeholders with techniques for reasoning about the system at higher levels of abstraction than source code. As the combination of collaborative software engineering and MDSE, *collaborative MDSE* exhibits the traits of both disciplines. Collaborative MDSE has become a prominent feature of nowadays' software engineering practice [8]. Version control for modeling artifacts has been extensively employed to facilitate collaboration. Such approaches rely either on lock mecha-

¹ <https://github.com/geodes-sms/lowkey>

nisms [30] or manual conflict resolution [53]. As a consequence, they are not suitable for real-time collaboration.

3.2 Real-time collaboration

Recent studies by David et al [13] and Franzago et al [23] show a strong shift towards real-time collaboration in MDSE. The main challenge in such a shift is ensuring appropriate convergence of distributed replicas, while still guaranteeing timely execution [49]. Convergence of replicas is ensured by the consistency model a distributed setting chooses. Strict consistency is a theoretical model for guaranteeing deterministic consistency by the total order of change updates that are exchanged instantaneously. However, due to its limited usability, various relaxations have been provided, such as sequential consistency [31], causal consistency [21], and eventual consistency [59]. Strong eventual consistency (SEC) [39] augments the liveness property of eventual consistency (all change updates will be observed eventually) with a safety guarantee: two nodes that have received the same set of change updates will be in the same state, regardless of the order of updates. SEC is an efficient resolution of the CAP theorem by Brewer [9], suggesting that distributed systems cannot provide more than two out of the three properties of strong consistency, availability, and partition tolerance. SEC removes the problem of conflict resolution on local replicas by introducing rules to ensure a unique outcome for concurrent changes, deterministically resolving any conflict. There is no need for a consensus or synchronization since any kind of change is allowed and conflicts are removed altogether. As such, this consistency model is especially appropriate for real-time collaboration. Nevertheless, SEC may be challenging or even impossible to implement for certain data types.

3.3 Conflict-free replicated data types

CRDTs eliminate conflicts between the distributed stakeholders' operations; thus avoiding the complexity of conflict resolution and roll-back. As a result, CRDTs exhibit promising fault tolerance and reliability properties.

CRDTs come in two flavors. State-based CRDTs are structured in a way that they adhere to a monotonic semi-lattice. Shapiro et al [46] show that state-based objects that satisfy this property are SEC, hence they converge to a consistent state. Operation-based CRDTs require that concurrent operations are commutative. Independently from the order of the received change

updates, the state of the local copy converges to the same state. To achieve such a behavior, the supporting communication protocol has to provide a causal ordering mechanism, such as global timestamps. In our approach, we have opted for the operation-based CRDT scheme because of its reduced costs when exchanging model changes between replicas.

Operation-based model representation [32] has been proposed for reasoning about streams of model operations. The C-Praxis approach [35] defines six CRUD model operations and defines how these operations interact. The CRDTs of *lowkey* are geared towards the more complex semantics of graphs and, thus, they provide a superset of these operations. Additionally, *lowkey* enables working with multigraphs, hypergraphs, and disconnected graphs, allowing for more flexibility in modeling. Traditional operation-based approaches, such as C-Praxis are built on MOF. Consequently, they fall short of supporting arbitrary meta-levels of modeling.

3.3.1 The Last-Writer-Wins (LWW) paradigm

To ensure the convergence of local replicas, their differences have to be resolved in an automated fashion. Such a resolution mechanism can be implemented either in the application or at the data level [34]. The LWW paradigm [29] has been widely adopted as a data-level implementation of operation-based conflict resolution [54, 41]. Conflicting operations are resolved using a global ordering operator, e.g., a timestamp. Given two changes, it is the more recent one that will prevail [45]. To avoid potential data loss, each change update is stored locally and the resolution of conflicts is carried out by the local replica.

Figure 2 shows an example resolution scenario under LWW. User A (top blue) and User B (bottom green) initially have their local replicas in identical states: the value of x and its timestamp t . At $t = 1$, User A executes the update $x = 15$ on his local copy. A message with this updated value and the timestamp is sent to User B. However, before the message arrives, User A executes another update: $x = 20$, at time $t = 2$. Again, an update message is composed and sent to User B. Due to network delays, the second update arrives to User B earlier than the first. Upon receiving the update message, User B will reconcile this new value with his local replica. As it stands, User B has $x = 10$ timestamped with $t = 0$; and an update that says $x = 20$ timestamped with $t = 2$. Under the LWW paradigm, the latter value prevails, due to the more recent timestamp. Eventually, the first message arrives. User B has $x = 20$ timestamped with $t = 2$; and an update of $x = 15$ timestamped with $t = 1$. Under the LWW paradigm,

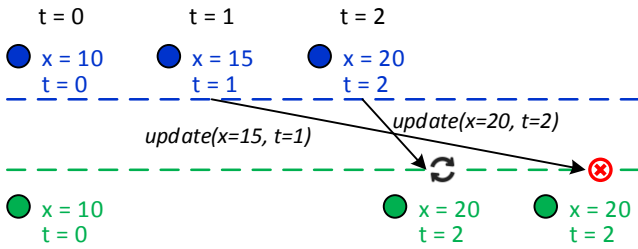


Fig. 2 Total order of updates in the LWW paradigm.

the former value should prevail. Thus, the update is not performed on the local copy. Eventually, the replicas are in identical states: $x = 20$.

Despite the omitted update after receiving $x = 15$, $t = 1$, the message can still be stored in User B's local replicas. It is the responsibility of the CRDT implementation to decide whether to store outdated data or not. Some use-cases might require such behavior; but the performance of CRDTs is proportional with the data they store [51], and often requires implementing complex garbage collection mechanisms [6].

The LWW paradigm satisfies the requirements for real-time collaboration defined by Sun et al [49]: (i) convergence, i.e., every stakeholder's local data must exhibit the same state after updates have been applied; (ii) user intention preservation, i.e., the original user's intention must be preserved; (iii) causality preservation, i.e., updates must be ordered in the same causal way by each stakeholder; and (iv) timely execution, i.e., operations must propagate and the system must reconcile within a deadline that provides a smooth user experience.

3.3.2 CRDT frameworks

Yjs² is an open-source framework for peer-to-peer shared editing of structured data, such as rich-text, or XML. Operations are stored in a linked list, resulting in a total order of operations, thus implementing CRDTs. Teletype³ provides string-wise sequence CRDTs for peer-to-peer collaborative editing in the Teletype for Atom⁴ cooperative source code development environment. AutoCouch [25] is a JSON framework combining the benefits of the Automerge CRDT library⁵ and the CouchDB⁶ database engine. Conflict-free JSON documents are replicated both on the server side and client side, while ensuring a responsive real-time user experience for web-based applications.

² <https://github.com/yjs/yjs>

³ <https://github.com/atom/teletype-crdt>

⁴ <https://teletype.atom.io/>

⁵ <https://github.com/automerge/automerge>

⁶ <http://couchdb.apache.org/>

These frameworks are similar to *lowkey* in their aim to augment engineering tools with a CRDT-based collaboration service. However, they are primarily geared towards linear data types, whereas the CRDT layer of *lowkey* is primarily aimed at supporting a wide range of modeling scenarios. Currently, no other modeling framework implements support for graph CRDTs as *first-class citizens*. Shapiro et al [46] formalize graph CRDT but provide no implementation. SyncMeta [20] (built on top of Yjs) provides support for modeling graph-like data structures. However, graphs are emulated by linked lists and string comparisons at the CRDT level.

3.4 Multi-level modeling

A formal framework of modeling at arbitrary meta-levels has been developed by Atkinson and Kühne [2], introducing the core concepts of deep instantiation [4] and deep characterization [5]. Deep instantiation extends the traditional two-level instantiation and allows classes to be instantiated transitively. This is achieved by the notion of potency that defines how many levels of instantiation the class supports. Deep characterization allows meta-types to influence the characteristics of their instances beyond those in the level immediately below. Our framework embraces these concepts to provide collaborative support in a highly generalized fashion. As shown by Atkinson and Kühne [4], deep meta-modeling can naturally accommodate traditional modeling frameworks built on shallow instantiation, such as UML and EMF. Consequently, our framework is a good fit with modeling tools supporting UML and EMF modeling, but lacking collaborative features. The feasibility of multi-level modeling has been demonstrated in tools such as MetaDepth [16] and Melanee [1].

4 A framework for real-time collaborative metamodeling

We present the different components of *lowkey*, our real-time collaborative modeling framework.

4.1 Architecture

Figure 3 outlines the architecture of the framework. In a typical modeling setting, users are provided with (domain-specific) *Editors* that enable interacting with the (meta)models of the *Linguistic layer*, through a generated *Domain-specific API*. The Linguistic layer enforces well-formedness rules defined by the static semantics of the language. Each element at the *Linguistic*

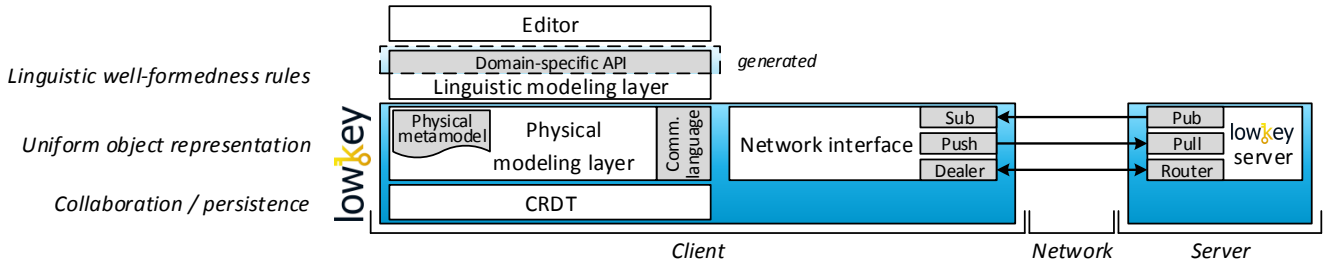


Fig. 3 Overview of the architecture and the roles of each layer

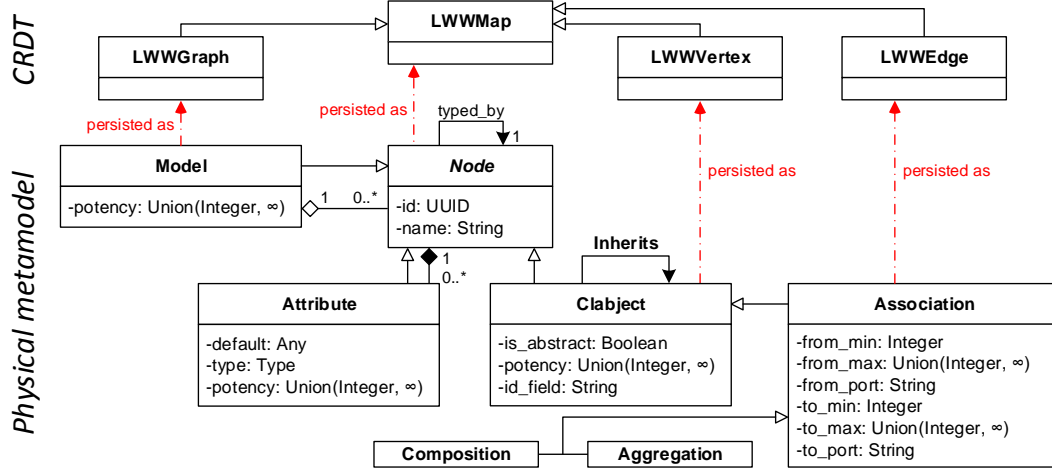


Fig. 4 The Physical metamodel and CRDT layers

layer conforms to a *Physical metamodel*, as described by Van Mierlo et al [55]. The *Physical layer* is responsible for the uniform representation of objects of the linguistic models and metamodels. Instances of the Physical metamodel are serialized and propagated to the collaborating stakeholders through their *Network interface* which communicates with the server. Other clients receive these instances from the server and merge them into the local working data represented in CRDTs. The *Command language* of the Physical layer enables a uniform treatment of local updates (from the Linguistic layer), and remote updates (from the Network interface).

4.2 Linguistic modeling layer

The Linguistic modeling layer provides mechanisms for modeling at arbitrary levels of abstraction. As a consequence, this layer contains every domain model and its metamodels the Editor manipulates. To this end, a *Domain-specific API* is generated for the metamodels. In the illustrative case, the *MindMap* class and its instance(s) are situated at this modeling layer; and methods for CRUD operations are generated.

The main responsibility of this layer is to enforce the language-specific well-formedness rules defined by

the meta-model and static semantics. Typical examples of language-specific well-formedness rules include multiplicities, transitive containment by compositions, and uniqueness of marker names.

4.3 Physical modeling layer

The main responsibility of the Physical modeling layer is to provide a metamodel that every linguistic concept can conform to, regardless of which linguistic meta-level they are situated at. For example, the Physical metamodel has to accommodate both the *MindMap* class; its instance *mindmap_0*; and the *Class* class the *MindMap* class corresponds to. In addition, the Physical modeling layer may impose language-independent well-formedness rules, such as each model having to exhibit a graph structure.

4.3.1 Physical metamodel

We adopt the Physical metamodel from previous work [55], as shown at the bottom of Figure 4.

Node is the foundational concept of the metamodel, which can be organized into *Models*. Models, in turn, are *Nodes* themselves, allowing for the hierarchical composition of *Models*.

A consequence of metamodeling—and multi-level modeling in particular—is that instantiable model elements can play the role of both instances and types [2]. To accommodate this dichotomy, *Clabjects* serve as the physical metatype for every linguistic class and instance. For example, in the mindmap metamodel, both the *CentralTopic* class and its instance(s) are mapped onto the *Clabject* physical type.

Associations link *Clabjects* to each other. The *Association* inherits from the *Clabject*, and transitively from the *Node*. Due to the latter, *Associations* can be typed by other *Nodes*. Due to the former, *Associations* can be abstract and link other *Associations*. We allow this flexibility to accommodate the models of various modeling frameworks and formalisms that typically implement a subset of these modeling options. UML, for example, restricts *Associations* from being abstract, and only allows linking *Clabjects*. *Composition* and *Aggregation* are specialized types of *Association* with conventional semantics.

Attributes store information of specific *Nodes*. Specifically, in *Attributes*, the *type* property maintains information about the linguistic type the *Attribute* corresponds to. For example, storing textual information (such as the *name* of a *Topic* in the illustrative case) in the Eclipse Modeling Framework (EMF) would mean the type of the *Attribute* is *EString*. The *typed_by* relationship inherited from the *Node* provides a mechanism for typing an *Attribute* at the physical level.

4.3.2 Need for an explicit Physical metamodel

Instantiation mechanisms of current modeling frameworks typically consider two levels: classes and their instances (objects). Although this covers the majority of practical use cases, some scenarios might require multiple levels of type-instance relationships [17]. Such a type-instance hierarchy enables various beneficial mechanisms, such as deep instantiation and deep characterization [4]. The role of a physical metamodel is to represent the objects of such a type-instance hierarchy uniformly, irrespective of what metalevel a specific object is situated at; and to provide services such as the serialization of these objects. By this separation of concerns, objects of the type-instance hierarchy become independent of their physical representation, and their roles are determined by purely linguistic concepts, such as being metatypes or instances of each other. Therefore, we refer to this level of objects as the *Linguistic metamodel*.

The physical metamodel of traditional modeling frameworks is typically coupled with the topmost meta-level of their linguistic metamodels. For example, Ecore serves as the core metamodel of EMF and also defines the

rules governing the serialization of EMF models into XMI files.

By providing mechanisms for multi-level metamodeling, and clearly separating the *Physical metamodel* from the *Linguistic metamodel*, our framework is able to (i) support advanced metamodeling scenarios; and (ii) accommodate traditional modeling frameworks, for example, by restricting the flexibility of the *Physical metamodel*. Similar avenues have been explored by multiple modern modeling frameworks, such as Melanee [1], metaDepth [16], and the Modelverse [56].

4.3.3 Command language

The *Physical metamodel* can be accessed and interacted with through a command language. The brief definition of the language is shown in Listing 1. The command language defines four CRUD operations. Therefore, integrating an editor with the *Physical metamodel* requires the appropriate facility that translates domain-specific modeling operations to the operations of the command language. We have opted for providing an external textual DSL because such languages are more trivially serialized and propagated through standard network protocols than binary data.

4.4 CRDT layer

The CRDT layer is responsible for persisting instances of the Physical metamodel. As shown in Figure 4, each element of the Physical metamodel is associated with exactly one CRDT. At run-time, as the classes of the Physical metamodel get instantiated, a corresponding CRDT is instantiated as well. The model element maintains a reference to its persisting CRDT instance during execution. Our CRDT implementations follow the LWW paradigm and have been mainly implemented by following the specifications outlined by Shapiro et al [45]. In the following, we briefly elaborate on these types.

4.4.1 Timestamps for total order

To ensure the commutativity of operation-based CRDT, an operator for total order is required. The most natural choice in a distributed setting is a global timestamping mechanism, as it makes the fewest assumptions about the system. Another useful operator could be the priority of messages, e.g., to define hierarchical stakeholder roles in which higher ranked roles can overwrite the changes of lower-ranked roles. An important difference between timestamps and priorities is the level of granularity. Timestamps (e.g., at the level of nanoseconds)

```

1 CREATE -name {name} -typedBy {type} [-attributeName {value}]*
2 LINK -from {fromClobject}.{associationName} -to {toClobject} [-attributeName {value}]*
3 UPDATE (-name {name} | -id {id}) [-attributeName {newValue}]*
4 DELETE (-name {name} | -id {id})

```

Listing 1 Command language for interacting with the Physical metamodel

provide better chances to unambiguously order two operations, as compared to priorities (e.g., assigned from a range between 1-5). Therefore, the ordering operator should be carefully designed to ensure the total order. Furthermore, compound ordering operators can be used as well. For example, the shortcomings of the priority operator can be circumvented by combining it with timestamps. Such compound operators are best defined by linking them through the absorption identity over the lattice of change operations [12] to ensure a valid and sound composition.

In the remainder of the paper, we showcase the use of timestamps to define a total order. `lowkey` uses the Unix epoch time in nanoseconds to timestamp model updates with. As an alternative, Lamport clocks [31] or vector clocks [47] can be used as suggested by Shapiro et al. [45]. Timestamped model updates are subsequently forwarded to the collaborating stakeholders. The global nature of the timestamp ensures that each local replica sorts the updates in the same order, ensuring the safety property of SEC (c.f. Section 3.3).

The appropriate granularity of timestamps is paramount in supporting complex CRDTs. Consider the underlying data structure in the illustrative example in Figure 2 being a graph, in which vertices represent entities with associated attributes. Assigning a timestamp to the whole graph would not allow independent changes at finer-grained levels, such as vertices. Thus, we assign a timestamp to each instance of the elements of the *Physical metamodel*, and update it on each CRUD operation.

4.4.2 LWWRegister

The `LWWRegister` is the simplest CRDT in `lowkey`, containing a single value. To implement LWW semantics, its value v is equipped with a timestamp t . Thus, an `LWWRegister` r is defined as $r = (v, t)$.

update The *update* operation permits to modify the value of the register. Given a new timestamped value (v', t') , it is defined as $r.v := v'$ iff $t' > r.t$, and `NOP` otherwise.

4.4.3 LWWSet

The `LWWSet` S contains an arbitrary number of timestamped values (v, t) under set semantics. That is, for $S = \langle (v, t) \rangle$, it holds that $\forall s_i, s_j \in S : s_i.v \neq s_j.v$. We have implemented the `LWWSet` as an LWW-element-Set [45], in which the `LWWSet` is partitioned into two disjoint subsets $A, R \subseteq S$. A is the *add-set* containing the values added to S and B is the *remove-set* containing the values removed from S . This structure ensures the commutativity of *add* and *remove* operations: $add(v, t) \circ remove(v, t') \equiv remove(v, t') \circ add(v, t)$.

lookup The *lookup* operation indicates if an element is in the set. A value is considered to be in the `LWWSet` iff it can be found in the *add-set*, and it cannot be found in the *remove-set* with a higher timestamp. It is defined as $lookup(v) = \exists v, t, t' : (v, t) \in A \wedge (v, t') \in R, t' > t$.

add The *add* operation inserts new values in the *add-set*. It is defined as $add(v, t) : S.A \rightarrow S.A \cup \{(v, t)\}$.

remove The *remove* operation deletes values from the set by adding them to the *remove-set*. It is defined as $remove(v, t) : S.R \rightarrow S.R \cup \{(v, t)\}$.

4.4.4 LWWMap

The `LWWMap` is defined as an extension of the `LWWSet`. The data is stored as a key/value pair $((k, v), t)$.

lookup, query The *lookup* operation is modified so that it looks up the key instead of the value. That is, $lookup(k) = \exists k, t, t' : ((k, v), t) \in A \wedge ((k, v), t') \in R, t' > t$. The *query*(k) method returns the value for a key k .

add, remove Analogously to the *add* and *remove* of the `LWWSet`: $add((k, v), t) : S.A \rightarrow S.A \cup \{((k, v), t)\}$; and $remove((k, v), t) : S.R \rightarrow S.R \cup \{((k, v), t)\}$.

update This operation is the only substantial difference the `LWWMap` introduces to the `LWWSet`. Updating a key $((k, v), t)$ entry with a value v' timestamped with t' is defined as $update(k, v, v', t, t') = add((k, v'), t') \circ remove((k, v), t - \epsilon)$. First, the entry with key k is re-

moved, and the time of removal is timestamped with a value that is older than the new timestamp t' by the minimal time interval ϵ the system is able to detect. In `lowkey` $\epsilon = 1$ ns. Subsequently, the entry with key k and the new value v' is added with the timestamp t' .

4.4.5 LWWGraph

The `LWWGraph` G is the extension of the `LWWMap` that enables persisting attributes of the graph. This mechanism is used for storing the vertices and edges of the graph. Vertices are stored in an `LWWSet`, denoted by V . Each element of V is an `LWWVertex`. To enable storing attributes and metadata of vertices, each `LWWVertex` is an extension of the `LWWMap`. Analogously, the edge set E is an `LWWSet`, containing `LWWEdege` instances. Formally: $G = (V, E)$, where

- $V, E \vdash \text{LWWSet}$;
- $V = \langle (v, t) \rangle$, where $v \vdash \text{LWWVertex}$ and t is a timestamp;
- $E = \langle (e, t) \rangle$, where $e \vdash \text{LWWEdege}$ and t is a timestamp;
- $\forall e \in E : e.\text{query}(\text{"source"}), e.\text{query}(\text{"target"}) \in V$ (denoted $e.\text{source}$ and $e.\text{target}$).
- `LWWGraph`, `LWWVertex`, `LWWEdege` \vdash `LWWMap`.

Due to the invariant property of $E \subseteq V \times V$, operations on V and E are not independent. Shapiro et al [45] suggest multiple ways to manage this issue. In `lowkey`, we chose prioritizing the `removeVertex` operation to ensure CRDT behavior. That is, the operation is only allowed to be executed if it does not leave a dangling edge behind.

lookup and query Since both the `LWWVertex` and the `LWWEdege` extend the `LWWMap`, their lookup and query methods are identical to the ones discussed in Section 4.4.4.

addEdge, addVertex These operations reuse the API of the `LWWSet` directly. Adding an edge is achieved by adding the edge to the add-set A of the edge set E of graph G . Thus, $\text{addEdge}(e, t) : G.E.A \rightarrow G.E.A \cup \{(e, t)\}$; and $\text{addVertex}(v, t) : G.V.A \rightarrow G.V.A \cup \{(v, t)\}$.

removeEdge, removeVertex These operations reuse the API of the `LWWSet` directly. Removing an edge is achieved by adding the edge to the remove-set R of the edge set E of graph G . Thus, $\text{removeEdge}(e, t) : G.E.R \rightarrow G.E.R \cup \{(e, t)\}$; and $\text{removeVertex}(v, t) : G.V.R \rightarrow G.V.R \cup \{(v, t)\}$. Furthermore, to ensure the CRDT behavior: $\text{removeVertex}(v, t) \Rightarrow \nexists e \in E : e.\text{source}=v \vee e.\text{target}=v$.

Additional methods of the `LWWGraph` are defined for querying various properties of the graph; and for adding and removing vertices and edges by name and identifier. Additional methods of the `LWWEdege` and `LWWVertex` types are defined, e.g., for querying incoming and outgoing edges of a vertex, cascading the removal of dangling edges upon a vertex removal, and assigning direction to edges.

4.5 Network architecture

`lowkey` follows a client-server network architecture with multiple clients connecting to the same server, as outlined already in Section 4.1. The network architecture is built on top of the ZeroMQ⁷ asynchronous messaging library. It offers efficient scalability and latency properties; thus it aligns well with the requirements of real-time collaboration.

4.5.1 Server and Client components

Our framework provides two network components for connecting remote collaborating stakeholders. The server component is responsible for two tasks: (i) collecting from, and distributing updates among clients; and (ii) providing newly joined clients with the snapshot of the system so they have an initial local replica.

The client component is responsible for providing networking capabilities to modeling tools and editors. The *Client interface* is accessible from the API of the framework and it is the tool builder's responsibility to properly implement its required functionality. Specifically, the *Client interface* is properly implemented by defining the action the *Sub* socket executes periodically during its polling loop.

4.5.2 Communication patterns

Different responsibilities of the server are implemented with different communication patterns, as summarized in Figure 3.

Exchanging updates. To *receive updates*, a client must first *subscribe* to the updates by connecting to the *Pub* (publisher) socket of the server with its own *Sub* (subscriber) socket. The Pub-Sub pattern establishes a one-way asynchronous communication channel with the client receiving and processing messages in a polling loop. The *Pub* socket broadcasts messages to every client connected to the server. To *send updates*, a client must first connect to the *Pull* socket of the server with its *Push*

⁷ <https://zeromq.org/>

socket. Similar to the Pub-Sub pattern, the Pull-Push pattern establishes a one-way connection. Note that the *Push* socket is geared towards supporting pipelining mechanisms, hence it does not broadcast messages.

Distributing system snapshots. Upon connecting to the server, the client needs to obtain a snapshot of the system. This is achieved by connecting to the *Router* socket of the server via the *Dealer* socket of the client. The Router-Dealer pattern implements a request-reply mechanism with both ends acting asynchronously. As opposed to the patterns used for exchanging updates, this pattern does not maintain a connection after the request-reply pair of messages has been exchanged.

To provide a snapshot to its clients, the server is equipped with a memory that stores the history of previously exchanged messages (model updates). Once the newly connected client requests the snapshot, the history of messages is replayed to it. The state of the system is then built up locally by the client. We opted for this mechanism to keep it aligned with our choice of operation-based CRDT semantics (Section 3.3), and to shift the workload to the client instead of the server. Since operation-based and state-based CRDTs are able to emulate each other, it would be possible to communicate the whole state at once, but this would come at the price of increased network traffic.

4.5.3 Update messages

Every update message has the following signature: $\langle clientId, command, timestamp \rangle$, where the *clientId* is a UUID assigned to the client upon its creation, used for avoiding double delivery problems; *command* corresponds to the command language of the Physical metamodel shown in Listing 1; and *timestamp* is the timestamp of creation, as discussed in Section 4.4.1. Messages are serialized as text and sent through the TCP/IP stack.

5 Feasibility evaluation

Based on the case study, we evaluate the feasibility of *lowkey* from two points of view. In both cases, our objective is to assess whether *lowkey* can accommodate the challenges outlined in the case study. First, in Section 5.1, we provide a language engineer’s view by presenting the process of metamodeling in *lowkey*. Then, in Section 5.2, we provide a domain expert’s view by demonstrating four collaborative modeling scenarios.

5.1 Language engineer’s view: metamodeling in *lowkey*

Figure 5 shows how the metamodel of the illustrative case (Figure 1) is defined within the *lowkey* framework in relation to the Physical model (Figure 4). The *Linguistic metamodel* represents the same domain as the metamodel in Figure 1, but instead of using UML to express it, here, we use the *Physical metamodel* of the *lowkey* framework.⁸ The color coding shows how every concept *physically conforms* to the *Clabject*. This is due to the *Linguistic metamodel* being an *instance* of the *Physical metamodel*. In addition, the *Linguistic metamodel linguistically conforms* to the *Linguistic metamodel*. The latter is another instance of the *Physical metamodel*; thus, its elements *physically conform* to the specific elements of the *Physical metamodel* (shown by color-coding). Specific Mind map instances are created in the *Linguistic instance model* that *linguistically conforms* to the *Linguistic metamodel*, and *physically conforms* to the *Physical metamodel*. As emphasized by the arrow notation, the *mindmap_0* object is a linguistic instance of the *MindMap* class; which is, in turn, a linguistic instance of the *Class* class. Each of these are physical instances of the *Clabject* class. Similarly, the title of the mindmap "*Improve publication record*" is a linguistic instance of *title: String = ""*; which is, in turn, a linguistic instance of the *Attribute* class.

Thanks to the clear physical conformance relationships, every element at the *Physical instance* level will be persisted as a CRDT, irrespectively of their linguistic meta-level. Therefore, stakeholders can safely collaborate by editing any of the three linguistic metamodels.

Generating a domain-specific API. To enable interacting with the *Linguistic metamodel*, an API has to be produced. This process is fully automated with a template-based code generator that produces a Python class for every *Clabject*, and generates the appropriate accessors (e.g., *get*, *set*, *add*, *remove*) for attributes and references. Listing 2 shows the method signatures of the Python code generated for the *MindMap* class.⁹

```

1 class MindMap(Clabject):
2     # Attribute: title, Type: String,
3     # Multiplicity: 1
4     def getTitle(self)
5     def setTitle(self, title)

```

⁸ We remark, that multi-level modeling approaches traditionally rely on the orthogonal linguistic and *ontological* dimensions, and do not consider the physical dimension. In this paper, we only consider the linguistic and physical dimensions to allow an easier discussion. Our approach can be safely extended with ontological aspects, similarly to the work of [55].

⁹ The full example is available at <https://github.com/geodes-sms/lowkey>.

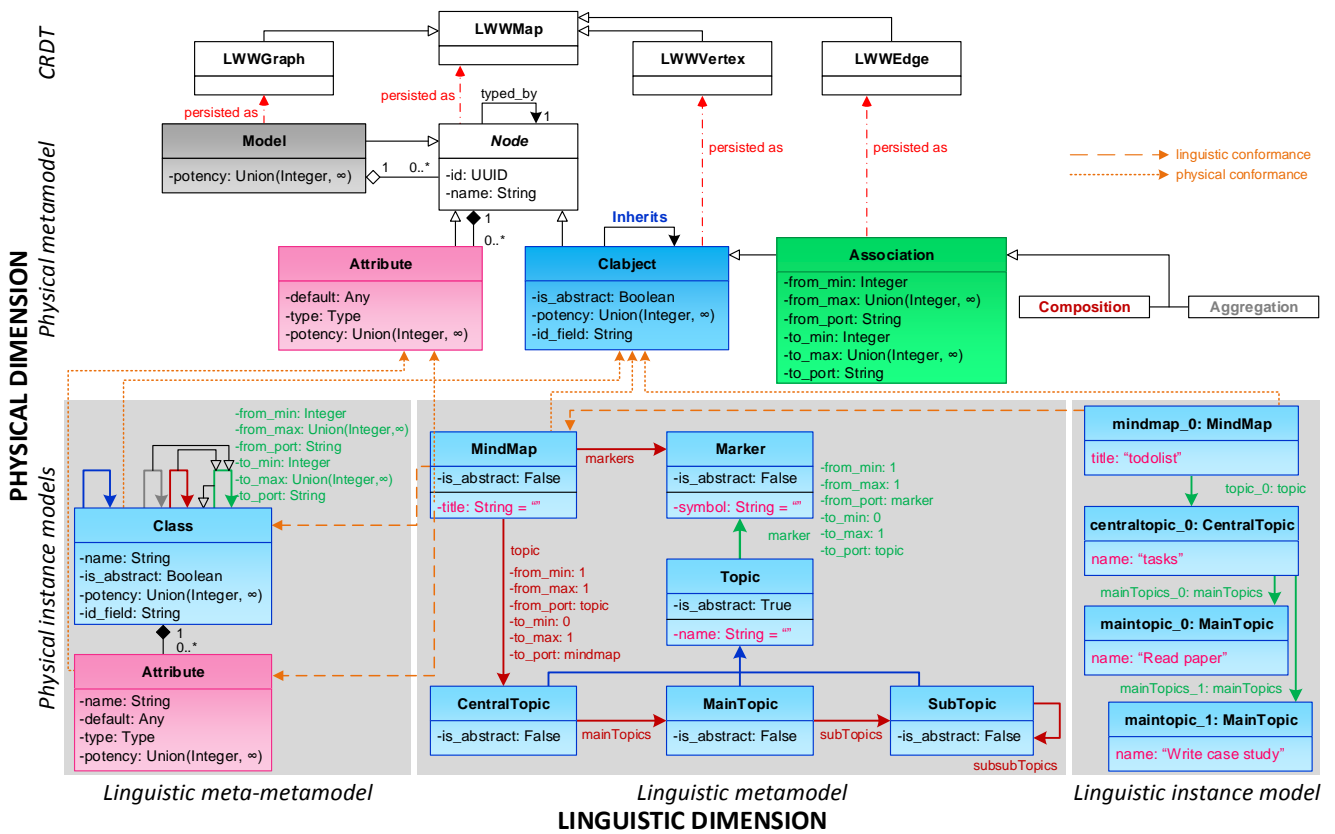


Fig. 5 The three-level hierarchy of linguistic models corresponding to the Physical metamodel in the mindmap case

```

7 # Reference: topic, Type: CentralTopic
8 # MultiplicityFrom: 1..1, MultiplicityTo: 1..1
9 # IsComposition: True
10 def getTopic(self)
11 def setTopic(self, topic: CentralTopic)
12 def removeTopic(self)

14 # Reference: markers, Type: Marker
15 # MultiplicityFrom: 0..1, MultiplicityTo: 0..*
16 # IsComposition: True
17 def getMarkers(self)
18 def addMarker(self, marker)
19 def removeMarker(self, marker)

```

Listing 2 Excerpt of the generated model API.

5.2 Domain expert's view: collaboration in lowkey

After the *Linguistic metamodel* has been defined, the domain experts can build the *Linguistic instance model* in a collaborative fashion. Figure 6 illustrates this collaboration by showing the objects of the Linguistic metamodel and the CRDTs at the local replicas. For space considerations, we omit the technical details, such as clients connecting to the server; and only show the CRDTs for *Client A*. For testing, evaluation, and demonstration purposes, we implemented a simple Mind map editor via the command line. The commands of the editor are shown in Listing 3. In the following,

we outline typical collaboration scenarios based on our experiments with the editor.

1. Cooperation

- Client A creates a MindMap instance by issuing the `CREATE MindMap mindmap_0` command. As shown in Figure 3, the command is parsed into Python source code that uses the Domain-specific API (Listing 2) of the linguistic level. The role of the linguistic level is to enforce linguistic well-formedness rules. Since the model with the single `mindmap_0` object is well-formed, the command is further translated to the command language of the physical layer (Listing 1).
- The client sends this command to the Server, using their respective *Push-Pull* sockets. The message is stored in the list of updates at the Server.
- At the CRDT level, an `LWWVertex` is instantiated to store the local data at the client side. The key-value pairs of the `LWWVertex` store every information required to reconstruct the linguistic object, i.e., $(type, MindMap)$; $(name, mindmap_0)$; and $(title, mindmap_0)$. In the current implementation of *lowkey*, an `LWWGraph` is automatically instantiated to accommodate the `LWWVertex` and `LWWEdege` instances created throughout the collaboration.

```

1 READ -- Returns the mindmap model in a readable form
2 OBJECTS -- Lists every object in the local session
3 CREATE {type} {name} -- Creates an instance with name of the domain-specific type
4 LINK {source}.{port} TO {target} -- Links object target to object source via port
5 UPDATE {name} {attribute} {newValue} -- Updates attribute with name to newValue
6 DELETE {name} -- Deletes object name

```

Listing 3 MindMap DSL of the Editor

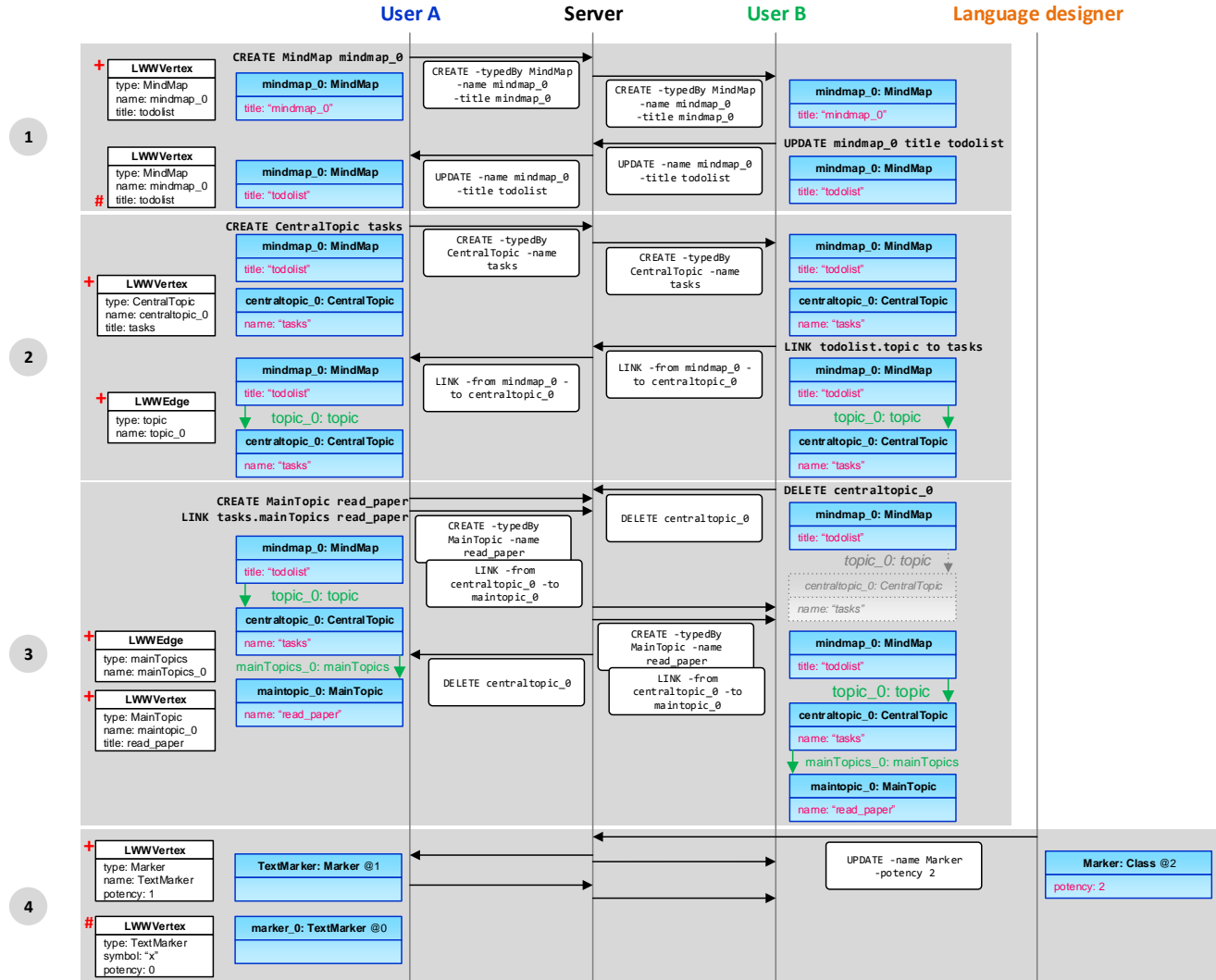


Fig. 6 Collaboration in lowkey outlining changes in the CRDTs shown for Client A

- The Server broadcasts the change to the collaborators, i.e., Client B. Subsequently, an object from the `MindMap` class is instantiated with the same ID as the one at User A's side.
- Client B changes the `title` attribute of the `mindmap_0` object by issuing the `UPDATE` command. The change is applied locally and forwarded to the Server after the required conversion to the command language. Client A receives the change from the Server and updates its local replica.
- At the CRDT level, the `LWWVertex` is updated by changing the `title` from `mindmap_0` to `todolist`, reflecting the changes by *Client B*. As explained in Section 4.4, the previous and current values are stored in the *Add-set* of the `LWWVertex` (inherited from the `LWWMap`). Querying the CRDT instance yields `todolist` as the value with the latest timestamp.

2. Cooperation with linguistic inconsistencies

- Client A creates a `CentralTopic` instance by the name *tasks*. The *centraltopic_0* object is not linked to the *mindmap_0* object, violating the 1-1 relationship between the *MindMap* and *CentralTopic* classes, prescribed by the static semantics of the metamodel (Figure 1). In this example, we assume a less restrictive linguistic layer, in which static semantics are not enforced. The physical layer, however, enforces physical well-formedness rules, e.g., the model has to correspond to a graph, which is allowed to be disconnected. (This rule is defined by the aggregation between the *Model* and *Node* types in Figure 4.) The editor-level user operation, thus, is propagated to the physical layer, translated into the command language: `CREATE -typedBy CentralTopic -name tasks`. Because of the valid physical model, the message is forwarded to the server and distributed to Client B.
- At the CRDT level, a new `LWWVertex` is created to store the *CentralTopic* instance.
- Client B links the `CentralTopic` instance to the `MindMap` instance. The distribution of the tasks of object creation (Step 5) and linking (Step 6) creates a truly collaborative setting, demonstrating cases in which stakeholders of specialized expertise take care of partial tasks, not necessarily resulting in a valid linguistic model on every atomic operation. Client B creates an association of type *topic* between the two objects, named *topic_0*. The operation results in a valid linguistic model, the changes are propagated in the usual way, and eventually, the change is reflected in the replica of Client A.
- At the CRDT level, a new `LWWEdege` is created to store the association. The `LWWEdege` is added to the `LWWGraph` with the previous two `LWWVertex` as its source and destination.

3. Conflict

- Client A works on objects that Client B has deleted. First, Client B deletes the *centraltopic_0* object, and as a consequence, the object is marked deleted in the local replica. As explained in Section 4.4, CRDTs use soft delete to be able to restore data if needed. The update message is sent to the server. In the meantime, Client A creates an instance of the *MainTopic* class and links it to the *centraltopic_0* object via the *mainTopics_0* reference. The two update messages are propagated to the server. Client A started its changes before the updates about Client B's changes have reached him. Subsequently, both clients' updates are sent to the other client, causing

inconsistency in the local replicas. However, both inconsistencies are resolved immediately by the LWW semantics of CRDTs. Since Client A's updates have been created later, the timestamping mechanism will resolve the inconsistencies by (i) leaving Client A's replica intact and (ii) re-adding the *centraltopic_0* object to Client B's replica.

- At the CRDT level, the local replica of *Client B* removes the `LWWVertex` storing the *CentralTopic* instance: the vertex is removed from the vertex set of the `LWWGraph`. This is achieved by adding the vertex to the *Remove-set* of the graph, as explained in Section 4.4.4. Before this information is propagated, *Client A* performs editing operations resulting in a new `LWWEdege` and a new `LWWVertex`. The source of the newly created `LWWEdege` is the `LWWVertex` removed by *Client B*, hence the conflict. However, due to the more recently created (and timestamped) `LWWEdege`, the `LWWVertex` is not removed in the local replica of *Client A*. After *Client B* is notified about the operations of *Client A*, the `LWWVertex` storing the *CentralTopic* instance is re-added to its local replica, restoring the consistency between the clients.

4. Multi-level cooperation

- Users A and B have created multiple instances of the *Marker* class and would like to categorize them into textual and graphical markers. One way to achieve this is by allowing a templating mechanism to these users so that they can create specific sub-classes of the *Marker* class. These classes can be then instantiated with a specific *symbol*.
- The Language designer increases the *potency* of the *Marker* class from 1 to 2. The change is distributed to the users.
- User A creates a new instance of the *Marker* class by the name *TextMarker*. The *potency* of the newly created object is set to 1, allowing one more instantiation. Subsequently, he modifies the previous *marker_0* in a way that it is typed by the newly created *TextMarker* class. The *potency* of *marker_0* is set to 0, preventing any further instantiation.

5.3 Summary

In this example, we have demonstrated the usage of `lowkey` in advanced modeling scenarios, such as meta-modeling at arbitrary levels of abstraction and seamless collaboration in the presence of inconsistencies and non-conformance. The language engineer's point of view

Table 1 Comparison of related modeling frameworks

Framework	Metamodeling	Multi-level modeling	Real-time collaboration	Consistency model	Conflict mgmt/ Resolution
AToMPM [52]	●	⦿	●	Strong	Manual resolution
Modelverse [56]	●	●	○	N/A	N/A
MetaDepth [16]	●	●	○	N/A	N/A
Melanee [1]	●	●	○	N/A	N/A
WebGME [33]	●	○	●	Eventual	Manual resolution
SpacEclipse [24]	●	○	●	Strong	Manual resolution
FlexiSketch [61]	●	○	●	Strong	Prevention
SyncMeta [20]	⦿	○	●	SEC	Prevention (CRDT)
MetaEdit+ [30]	●	○	○	Strong	Pessimistic locking
TGRL [42]	○	○	●	SEC	Prevention (CRDT)
MONDO [19]	○	○	○	Strong	Pessimistic locking
lowkey	●	●	●	SEC	Prevention (CRDT)

(Section 5.1) has shown the development of domain-specific modeling languages with multi-level modeling capabilities. **lowkey** enabled reusing the metamodel of the case (Figure 1) that was previously defined in an external tool. The domain expert’s point of view (Section 5.2) has demonstrated how real-time collaboration is achieved by CRDTs, irrespectively of the level of linguistic abstraction. In addition, we have highlighted how the combination of CRDTs and the Physical metamodel allows for seamless collaboration in the presence of linguistic inconsistencies.

6 Discussion

In this section, we assess how **lowkey** compares to other modeling and collaborative frameworks. Then, we reflect on various aspects of the approach.

6.1 Comparison

We compare **lowkey** with frameworks that are closest in their aim and feature set, shown in Table 1. These are typically either modeling tools with multi-level modeling capabilities or tools with real-time collaborative features. Our objective is to assess how **lowkey** compares in terms of the key functionality of (i) metamodeling, (ii) multi-level modeling, and (iii) real-time collaboration. We find that some tools partially overlap with **lowkey** in terms of functionality; however, the combination of the three key features is unique to **lowkey**.

Metamodeling, i.e., the ability to construct new metamodels and modeling languages, is supported by the majority of the sampled tools. MetaEdit+ [30] is a widely adopted metamodeling framework. Locking at the class level (not attributes) is the primary collaborative mechanism, but there is no support for real-time

collaboration. MetaEdit+ represents a class of modeling tools that gained substantial industrial adoption, and could benefit from a real-time collaborative framework such as **lowkey**. The two exceptions are TGRL [42] and the MONDO framework [19]. TGRL is a tool for requirements modeling in a collaborative way. MONDO provides collaborative mechanisms for domain-specific modeling.

Multi-level modeling, i.e., the ability to define metamodels at an arbitrary number of meta-levels, is supported by the mechanisms of deep characterization [5] and deep instantiation [4]. AToMPM [52] is a web-based multi-view modeling tool that allows for defining metamodels at arbitrary number of levels and instantiating them by bootstrapping mechanisms. However, deep characterization and deep instantiation are not supported. The Modelverse [56] is a modelware back-end for storing and simulating models. It achieves multi-level modeling by using a physical metamodel similar to **lowkey** [55]. The same approach has been used by deep metamodeling frameworks MetaDepth [16] and Melanee [1]. Similar to **lowkey**, these tools use graphs at the meta-circular level, i.e., the topmost linguistic meta-level.

Real-time collaboration is becoming increasingly adopted in model editors [13]. Tools such as WebGME [33], SpacEclipse [24], FlexiSketch [61], and to some extent SyncMeta [20] support metamodeling by shallow instantiation, augmented with real-time collaboration capabilities. As a consequence of shallow instantiation, these tools are restricted to a three-level meta-hierarchy, such as OMG’s MOF or EMF [48]. A variety of **consistency models** are employed in these tools to support collaboration. Strong consistency, employed in AToMPM, SpacEclipse, and FlexiSketch, ensures that all participating nodes hold the exact same state of the model at all times. However, due to its underlying

mechanisms, it significantly hinders the scalability and user experience of collaborative modeling tools [31]. In AToMPM, real-time collaboration is supported in two ways: at the levels of the abstract and concrete syntax. In both cases, changes in the abstract syntax are shared; in the latter case, changes in the representation are shared as well. Collaboration in SpacEclipse and AToMPM requires manual **conflict resolution**, while FlexiSketch uses preventive conflict management techniques. WebGME relies on eventual consistency, that provides the weaker guarantee that changes will be eventually observed across each node [59]. However, conflict resolution is not automated. Novel types of real-time collaborative tools, such as TGRL and SyncMeta employ strong eventual consistency (SEC) that combines the benefits of strong and eventual models [45] and avoids conflicts altogether. TGRL and SyncMeta implement real-time collaboration using the Teletype and Yjs CRDT frameworks, respectively.

Summary. As shown in Table 1, *lowkey* provides a unique combination of features for real-time collaborative multi-level modeling. Typically, modeling frameworks either provide multi-level modeling capabilities without support for real-time collaboration (e.g., Modelverse, MetaDepth); or provide real-time collaboration capabilities without support for multi-level modeling (e.g., WebGME, FlexiSketch). Closest to our work is AToMPM, which provides limited facilities for multi-level modeling, and supports real-time collaboration by conservative consistency model and without automated conflict resolution.

6.2 Physical metamodel

One of the main benefits of the Physical metamodel is the uniform representation of objects and models, irrespective of the linguistic meta-level they are situated at. This mechanism allows for the co-existence of models with different syntaxes and semantics. As a consequence, the collaboration between different modeling tools becomes a more manageable endeavor.

We have chosen *graphs* as the meta-circular level. That is, the Physical metamodel corresponds to graphs, and all linguistic models correspond to graphs as well. As discussed in Section 6.1, graphs have been shown to be an appropriate and versatile choice for such purposes. We have implemented a directed multigraph formalism, i.e., edges have an unambiguous source and target vertex, and multiple edges are allowed between vertices. Directed edges enable navigability of associations in linguistic models, and the multigraph nature enables defining multiple different associations between

the same pair of classes. Additionally, the Physical metamodel supports disconnected graphs. We found this property useful in enabling the temporal tolerance of linguistic inconsistencies. Additional graph properties can be implemented and enforced, depending on the use-cases to be supported by the framework.

6.3 Temporal tolerance of linguistic inconsistencies

The separated Physical metamodel allows for collaboration in the presence of linguistic non-conformance (vertical inconsistencies), enabling advanced collaboration scenarios. For example, ensuring the consistency of a model during the collaboration of stakeholders with different expertise might require tolerating linguistic non-conformance. This has been demonstrated in Steps 5 and 6 of Figure 6. Here, the expertise of Client A is the instantiation of objects and the expertise of Client B is organizing dangling objects into models. Already in this simple example, we were able to introduce a non-conformance between the instance model and the metamodel when Client A did not link the newly created instance to the root object immediately. In practical applications, this issue is vastly exacerbated, as the number of stakeholders, domains, formalisms, and tools increases. The CRDTs persisting the Physical metamodel ensure horizontal consistency [58], i.e., stakeholders have a consistent view of the system, even if these views are linguistically incorrect. This enables a smooth collaboration in the presence of linguistic non-conformance.

The separation of physical and linguistic concerns enables well-formedness and consistency rules to be captured at the most appropriate levels of abstraction. For example, one might constrain their models at the physical level by enforcing conformance to graphs and at the linguistic level to class diagrams. Other examples of inconsistencies can also be tolerated. *lowkey* provides mechanisms to handle violated constraints on multiplicities, potency, and OCL rules. Advanced synchronization mechanisms might make use of these layered rules, for example by suppressing the propagation of updates until the desired level of horizontal consistency is achieved, and then propagating atomic updates in a batch fashion.

6.4 Support for complex modeling operations

Throughout this paper, we have assumed atomic change operations (e.g., CRUD operations in the running example). In practical modeling scenarios, more complex

domain-specific operations are often required to be supported, e.g., in the automation of large refactoring on the model(s). In the running example, a *promote* operation could be defined to move a *subtopic* with all its children directly under the *central topic*, converting the *subtopic* into a *maintopic* automatically. In our prototype editor, the same effect is achieved by multiple atomic CRUD operations. Such complex modeling operations could be supported by the presented CRDT mechanisms, e.g., by accumulating atomic operations at the client-side until the desired effect is achieved.

However, complex modeling operations are not compatible with the point semantics of LWW and require reasoning based on interval semantics [26]. Questions such as how to timestamp complex modeling operations to ensure user intention preservation and how to merge complex modeling operations into local CRDT replicas are still open research challenges. Constructing operations is mainly an editor-level concern, i.e., it is the tool builder, in collaboration with the language engineer, who has to define these complex modeling operations. However, making these operations first-class citizens at the level of the CRDT API improves the safety of applications built on top of the framework.

6.5 Opportunities in multi-view modeling

We anticipate multi-view modeling (MVM) [40] being one of the main application areas of our approach. Views are projections of one or multiple underlying models [11], presenting stakeholders with only the essential information they require for their work. Views typically pertain to domains and expertise (e.g., the mechanical and the electrical views in the design of a mechatronic system), but they can pertain to specific use cases (e.g., electro-mechanic view) or expertise (e.g., chassis design). The architecture and services of *lowkey* align well with the requirements of MVM. It is able to accommodate multiple different metamodels, their instances, and their views in a uniform fashion; thus allowing for change propagation between linguistically and semantically different views. Model- and screen sharing [57] are straightforward to implement, as tool builders can outsource the data layer of their tools to *lowkey*.

Other approaches relying on an ensemble of multiple models—such as multi-paradigm modeling (MPM) [37], multi-modeling [7] and blended modeling [15]—can benefit from this approach as well. MPM advocates modeling every aspect of the system at the most appropriate level(s) of abstraction using the most appropriate formalism(s). The Physical metamodel provides a basis for synchronization among stakeholders, while different formalisms can be implemented at the Linguistic level.

Similar techniques have been employed in the MPM tool Modelverse [56].

6.6 Accommodating mainstream modeling frameworks

As demonstrated by Atkinson and Kühne [4], multi-level modeling frameworks can emulate traditional modeling frameworks using the notion of potency. Potency is a constraint that specifies how many times a class can be instantiated transitively. As shown in Figure 4, the Physical metamodel of *lowkey* defines a potency to its *Clabject* element. By that, *lowkey* subsumes traditional modeling frameworks that operate on the mechanism of shallow instantiation. We see an opportunity in developing libraries for *lowkey* implementing the meta-facilities of these traditional frameworks. The process of adopting *lowkey* in already existing modeling tools relying on such frameworks, e.g., MOF or EMF, can be vastly improved and automated.

6.7 Limitations

Performance. The performance of CRDTs is subject to the number of objects present in the specific application [50]. *lowkey* CRDTs implement a soft delete mechanism, i.e., objects are never removed from the specific CRDT, but rather, “marked” as removed. The *LWWSet*, for example, contains the removed elements in its remove-set (see Section 4.4). As a consequence, the performance of the CRDT layer will gradually decrease as the number of objects increases. Our preliminary measurements show linear degradation. Garbage collection mechanisms have been suggested for managing such limitations of CRDT-based applications [6].

Intention preservation [49] plays a key role in achieving an intuitive human-computer interaction and a smooth user experience in collaborative settings. While the LWW paradigm usually preserves the user’s intention, some corner cases might result in model changes that are less intuitive. The user makes decisions on changing the model based on the model’s materialized view in the modeling tool or browser. However, there might be change updates from other collaborators on their way that might arrive after the user carried out his changes. These updates retroactively change the model in a way that affects the user’s reasoning. Since there is no way to account for such messages, the user’s intention in such corner cases cannot be guaranteed. While manual conflict resolution would solve this issue, it would also render real-time collaboration infeasible.

Timestamping mechanism. The current prototype implementation of *lowkey* uses the `time.time_ns()` Python

function for timestamping changes. This function returns time as an integer number of nanoseconds. However, this approach is prone to clock drift, which could render CRDT inconsistent. Our working assumption is that nanosecond-level drift does not affect the dynamics of collaborative modeling between humans, as the interactions in such settings are at the level of seconds. Nonetheless, *lowkey* can be extended by additional mechanisms to regularly synchronize clocks using the Network Time Protocol (NTP)¹⁰ and other mechanisms described in Section 4.4.1.

7 Conclusion

In this paper, we have presented a real-time collaborative framework for a wide range of advanced modeling scenarios, supported by techniques of multi-level modeling. Our framework provides a unique combination of modeling capabilities and real-time collaboration. It is built on a custom implementation of CRDTs, geared towards graph models, providing promising real-time capabilities and scalability. We have defined a mapping of physical metamodels onto CRDTs and demonstrated the approach through an illustrative case.

We have identified multiple lines of future work. We are planning to develop state-of-the-art collaborative multi-view modeling mechanisms based on the framework. To enable easier integration with legacy models, we will provide profiles for UML and EMF models, effectively reproducing the respective meta-levels of the OMG superstructure and Ecore. We plan to build a family of modeling editors integrated with existing modeling frameworks to augment them with real-time collaborative capabilities. In more technical terms, we are looking to improve the performance of the framework by implementing advanced garbage collection mechanisms and a network stack with less overhead. We will develop a benchmark for collaborative MDSE frameworks to evaluate their scalability (with respect to connected clients and messages exchanged), performance (response time of a local operation being propagated to each remote client), and usability in typical modeling scenarios.

References

1. Atkinson C, Gerbig R (2016) Flexible Deep Modeling with Melanee. In: *Modellierung, GI, LNI*, vol 255, pp 117–122
2. Atkinson C, Kühne T (2000) Meta-level independent modelling. In: *European Conference on Object-Oriented Programming*, vol 12, p 16
3. Atkinson C, Kühne T (2001) The essence of multi-level metamodeling. In: *The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, Springer, LNCS, vol 2185, pp 19–33
4. Atkinson C, Kühne T (2002) Rearchitecting the UML infrastructure. *ACM Trans Model Comput Simul* 12(4):290–321
5. Atkinson C, Kühne T (2008) Reducing accidental complexity in domain models. *Software & Systems Modeling* 7(3):345–359
6. Bauwens J, Boix EG (2019) Memory efficient CRDTs in dynamic environments. In: *International Workshop on Virtual Machines and Intermediate Languages*, ACM, pp 48–57
7. Boronat A, Knapp A, Meseguer J, Wirsing M (2008) What is a multi-modeling language? In: *Recent Trends in Algebraic Development Techniques*, 19th International Workshop, Springer, LNCS, vol 5486, pp 71–87
8. Brambilla M, Cabot J, Wimmer M (2017) *Model-Driven Software Engineering in Practice*, Second Edition. Morgan & Claypool Publishers
9. Brewer E (2012) CAP twelve years later: How the "rules" have changed. *Computer* 45(2):23–29
10. Buzan T (2006) *The ultimate book of mind maps: unlock your creativity, boost your memory, change your life*. HarperCollins UK
11. Corley J, Syriani E, Ergin H, Van Mierlo S (2016) *Modern Software Engineering Methodologies for Mobile and Cloud Environments*, IGI Global, chap *Cloud-based Multi-View Modeling Environments*, pp 120–139. 7
12. Davey BA, Priestley HA (2002) *Introduction to Lattices and Order*, Second Edition. Cambridge University Press
13. David I, Aslam K, Faridmoayer S, Malavolta I, Syriani E, Lago P (2021) Collaborative Model-Driven Software Engineering: A Systematic Update. In: *Model Driven Engineering Languages and Systems*, ACM, pp 273–284
14. David I, Aslam K, Malavolta I, Lago P (2022) Collaborative Model-Driven Software Engineering: Practices and Needs in Industry. To appear.
15. David I, Latifaj M, Pietron J, Zhang W, Ciccozzi F, Malavolta I, Raschke A, Steghöfer JP, Hebig R (2022) Blended Modeling in Commercial and Open-source Model-Driven Software Engineering Tools: A Systematic Study. *Software & Systems Modeling* To appear.

¹⁰ <http://www.ntp.org/>

16. de Lara J, Guerra E (2010) Deep Meta-modelling with MetaDepth. In: *Objects, Models, Components, Patterns*, Springer, LNCS, vol 6141, pp 1–20
17. de Lara J, Guerra E, Cuadrado JS (2014) When and How to Use Multilevel Modelling. *ACM Trans Softw Eng Methodol* 24(2):12:1–12:46
18. De Porre K, Myter F, Troyer CD, Scholliers C, Meuter WD, Boix EG (2019) Putting Order in Strong Eventual Consistency. In: *International Federated Conference on Distributed Computing Techniques*, Springer, Lecture Notes in Computer Science, pp 36–56
19. Debreceni C, Bergmann G, Búr M, Ráth I, Varró D (2017) The MONDO collaboration framework: secure collaborative modeling over existing version control systems. In: *Foundations of Software Engineering*, ACM, pp 984–988
20. Derntl M, Nicolaescu P, Erdtmann S, Klamma R, Jarke M (2015) Near Real-Time Collaborative Conceptual Modeling on the Web. In: *Conceptual Modeling*, Springer, LNCS, vol 9381, pp 344–357
21. Du J, Iorgulescu C, Roy A, Zwaenepoel W (2014) GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In: *Symposium on Cloud Computing*, ACM, pp 4:1–4:13
22. Fowler M, Scott K (2000) *UML distilled - A brief guide to the Standard Object Modeling Language* (2. ed.). Addison-Wesley-Longman
23. Franzago M, Ruscio DD, Malavolta I, Muccini H (2018) Collaborative Model-Driven Software Engineering: A Classification Framework and a Research Map. *IEEE Trans Software Eng* 44(12):1146–1175
24. Gallardo J, Bravo C, Redondo MA (2012) A model-driven development method for collaborative modeling tools. *J Netw Comput Appl* 35(3):1086–1105
25. Grosch P, Krafft R, Wölki M, Bieniusa A (2020) Autocouch: a JSON CRDT framework. In: *Workshop on Principles and Practice of Consistency for Distributed Data*, ACM, pp 6:1–6:7
26. Halpern JY, Shoham Y (1991) A propositional modal logic of time intervals. *J ACM* 38(4):935–962
27. Herbsleb JD (2007) Global Software Engineering: The Future of Socio-technical Coordination. In: *International Conference on Software Engineering*, IEEE, pp 188–198
28. Izquierdo JLC, Cabot J (2016) Collaboro: a collaborative (meta) modeling tool. *PeerJ Comput Sci* 2:e84
29. Johnson PR, Thomas R (1975) Maintenance of duplicate databases. RFC 677:1–10
30. Kelly S (2017) Collaborative modelling with version control. In: *Software Technologies: Applications and Foundations*, Springer, LNCS, pp 20–29
31. Lamport L (1978) Time, Clocks, and the Ordering of Events in a Distributed System. *Commun ACM* 21(7):558–565
32. Le Noir J, Delande O, Exertier D, da Silva MAA, Blanc X (2011) Operation based model representation: Experiences on inconsistency detection. In: *Modelling Foundations and Applications – 7th European Conference*, Springer, LNCS, vol 6698, pp 85–96
33. Maróti M, Kecskés T, Kereskényi R, Broll B, Völgyesi P, Jurácz L, Levendovszky T, Lédeczi Á (2014) Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure. In: *Multi-Paradigm Modeling*, CEUR-WS, pp 41–60
34. Meiklejohn C, Van Roy P (2015) Lasp: a language for distributed, coordination-free programming. In: *Principles and Practice of Declarative Prog.*, ACM, pp 184–195
35. Michaux J, Blanc X, Shapiro M, Sutra P (2011) A semantically rich approach for collaborative model edition. In: *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC)*, ACM, pp 1470–1475
36. Mistrik I, Grundy J, van der Hoek A, Whitehead J (2010) Collaborative Software Engineering: Challenges and Prospects. In: *Collaborative Software Engineering*, Springer, pp 389–403
37. Mosterman PJ, Vangheluwe H (2004) Computer Automated Multi-Paradigm Modeling: An Introduction. *Simulation* 80(9):433–450
38. Muccini H, Bosch J, van der Hoek A (2018) Collaborative Modeling in Software Engineering. *IEEE Software* 35(6):20–24
39. Preguiça NM, Marquès JM, Shapiro M, Letia M (2009) A Commutative Replicated Data Type for Cooperative Editing. In: *International Conference on Distributed Computing Systems*, IEEE, pp 395–403
40. Reineke J, Tripakis S (2014) Basic Problems in Multi-View Modeling. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, LNCS, pp 217–232
41. Roh H, Jeon M, Kim J, Lee J (2011) Replicated abstract data types: Building blocks for collaborative applications. *J Parallel Distributed Comput* 71(3):354–368
42. Saini R, Mussbacher G (2021) Towards Conflict-Free Collaborative Modelling using VS Code Extensions. In: *Model-Driven Engineering Languages and Systems: Companion Proceedings*, ACM, pp 35–44

43. Saito Y, Shapiro M (2005) Optimistic replication. *ACM Comput Surv* 37(1):42–81
44. Schmidt DC (2006) Model-Driven Engineering. *IEEE Computer* 39(2):25–31
45. Shapiro M, Pregoça N, Baquero C, Zawirski M (2011) A comprehensive study of convergent and commutative replicated data types. Tech. rep., Inria-Centre Paris-Rocquencourt; INRIA
46. Shapiro M, Pregoça NM, Baquero C, Zawirski M (2011) Conflict-Free Replicated Data Types. In: *Stabilization, Safety, and Security of Distributed Systems - 13th Int. Symposium*, Springer, Lecture Notes in Computer Science, vol 6976, pp 386–400
47. Singhal M, Kshemkalyani AD (1992) An efficient implementation of vector clocks. *Inf Process Lett* 43(1):47–52
48. Steinberg D, Budinsky F, Merks E, Paternostro M (2008) EMF: Eclipse Modeling Framework. Pearson Education
49. Sun C, Jia X, Zhang Y, Yang Y, Chen D (1998) Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Trans Comput-Hum Interact* 5(1):63–108
50. Sun D, Sun C (2006) Operation Context and Context-based Operational Transformation. In: *Conference on Computer Supported Cooperative Work*, ACM, pp 279–288
51. Sun D, Sun C, Ng A, Cai W (2020) Real Differences between OT and CRDT in Correctness and Complexity for Consistency Maintenance in Co-Editors. *Proc ACM Hum-Comput Interact* 4
52. Syriani E, Vangheluwe H, Mannadiar R, Hansen C, Mierlo SV, Ergin H (2013) AToMPM: A Web-based Modeling Environment. In: *Model-Driven Engineering Languages and Systems*, CEUR-WS.org, vol 1115, pp 21–25
53. Taentzer G, Ermel C, Langer P, Wimmer M (2010) Conflict detection for model versioning based on graph modifications. In: *Graph Transformations – 5th International Conference, ICGT*, Springer, LNCS, vol 6372, pp 171–186
54. Thomas RH (1979) A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)* 4(2):180–209
55. Van Mierlo S, Barroca B, Vangheluwe H, Syriani E, Kühne T (2014) Multi-level modelling in the modelverse. In: *Workshop on Multi-Level Modelling*, CEUR-WS, pp 83–92
56. Van Tendeloo Y, Vangheluwe H (2017) The Modelverse: A tool for Multi-Paradigm Modelling and simulation. In: *Winter Simulation Conference*, IEEE, pp 944–955
57. Van Tendeloo Y, Vangheluwe H (2018) Unifying model- and screen sharing. In: *Enabling Technologies: Infrastructure for Collaborative Enterprises*, IEEE, pp 127–132
58. Vanherpen K, Denil J, David I, Meulenaere PD, Mosterman PJ, Törngren M, Qamar A, Vangheluwe H (2016) Ontological reasoning for consistency in the design of cyber-physical systems. In: *Int. Workshop on Cyber-Physical Production Systems*, IEEE, pp 1–8
59. Vogels W (2009) Eventually Consistent. *Communications of the ACM* 52(1):40–44
60. Whitehead J (2007) Collaboration in Software Engineering: A Roadmap. In: *International Conference on Software Engineering*, IEEE, pp 214–225
61. Wüest D, Seyff N, Glinz M (2012) FlexiSketch: A Mobile Sketching Tool for Software Modeling. In: *Mobile Computing, Applications, and Services*, Springer, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 110, pp 225–244