Journal of Computer Languages 76 (2023) 101221

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

A new versioning approach for collaboration in blended modeling

Joeri Exelmans [a,*], Jakob Pietron [b], Alexander Raschke [b], Hans Vangheluwe [a], Matthias Tichy [b]

[a] University of Antwerp, Flanders Make, Antwerp, Belgium
[b] Ulm University, Ulm, Germany

ARTICLE INFO

Keywords:
Versioning
Blended modeling
Conflict-tolerant
Operation-based
Collaboration

ABSTRACT

The complexity of modern software-intensive systems and the need for flexibility in their development process forces developers to collaborate using the most appropriate language(s) for each given task, view and component. Blended modeling is the ability to edit a model through multiple concrete syntaxes simultaneously.

To support collaborative blended modeling, we present a variation of operation-based versioning that allows bi-directional propagation of changes between concrete and abstract syntaxes. This allows us to support layout continuity between different versions, and to handle information that is not (yet) available (e.g., layout information) when rendering changes from abstract to concrete syntax. Finally, our approach does not enforce immediate conflict resolution. Rather, different merge options and their consequences can be presented to the users, who may choose to only perform partial conflict resolution, deferring final resolution till later.

In this article, we present the general approach and describe salient parts of an implementation.

1. Introduction

Modern software-intensive systems have reached a complexity that can only be managed by a team of developers working together. The original idea to split up a software system into small independent pieces each manageable by a single person has its limits. Therefore, developers must collaborate with each other, not only because of cross-functional concerns, but also because of the flexibility needed in the development process to quickly respond to changes arising in requirements or regulations.

Consequently, the efficient support of collaborative development plays an increasingly important role. This includes versioning as a central piece of technology for any kind of collaborative work. Even in a synchronous context, each edit operation results in a new version that has to be compared to the (potentially also edited) version of a collaborator. Tools such as git [1] or SVN [2] already support asynchronous versioning in the context of textual languages quite well.

Regarding visual languages, a systematic mapping study on collaborative model-driven software engineering was conducted by Franzago et al. [3] in 2017 and recently updated by David et al. [4] to include the newest developments of this field. In these studies, a set of altogether 102 primary studies were analyzed with respect to the three main aspects "Model Management", "Collaboration", and "Communication". In the answer to their RQ2 (Challenges and Shortcomings) the authors mention "conflicts management" and "model synchronization with change propagation [...]" "to be a limitation" and a "relevant challenge" [3]. The results include the finding that "the majority [...] of approaches supporting synchronous collaboration do not provide any means for model versioning".

Some of the approaches dealing with (visual) modeling languages are EMF compare [5] or the tooling created in the AMOR [6] project. These developments compare and merge at the level of abstract syntax (AS). They treat AS as a graph structure, rather than dealing with its textual serialization. This avoids much of the accidental complexity of textual versioning. However, these approaches only support trivial one-to-one mappings between a single (visual) concrete syntax (CS) and AS. This in turn prevents *blended modeling*.

Blended modeling refers to the possibility to edit a model (in particular, its AS, which in turn is the basis for semantic mapping) through different views (the CSs) [7]. The user can freely decide at any time which CS (s)he wants to use, e.g., to perform a certain task as efficiently as possible. The main difference to projectional editing [8] is that in the latter, each CS is only a projection of the AS and can hence not be directly manipulated. Rather, each action directly manipulates the AS and changes are projected to the CS views. This tight coupling means that only editing operations can be performed that can be directly mapped to valid changes of the AS. Any temporary intermediate steps at the level of CS are not allowed and are prevented by an editing tool. These restrictions typically lead to a limited usability of such tools [9,10].

Instead, blended modeling aims at arbitrarily flexible visual, tabular, textual, etc. editors, each with their own CS meta-model. In order

* Corresponding author.
E-mail addresses: joeri.exelmans@uantwerpen.be (J. Exelmans), jakob.pietron@uni-ulm.de (J. Pietron), alexander.raschke@uni-ulm.de (A. Raschke), hans.vangheluwe@uantwerpen.be (H. Vangheluwe), matthias.tichy@uni-ulm.de (M. Tichy).

https://doi.org/10.1016/j.cola.2023.101221
Received 16 January 2023; Received in revised form 16 May 2023; Accepted 12 June 2023
Available online 17 June 2023

to support these scenarios, the definition of appropriate parsing and rendering functions that translate from a CS to an AS and vice-versa is required. The same editor can be reused for different languages, and makes it worth to optimize its usability. This is especially true for generic editors (e. g., SVG) where the supported language(s) just depend on the provided (graphical) parser(s) that translate the visual objects of the CS into elements of the AS [11]. The presented versioning approach also supports generic syntax but does not rely on it.

The technical challenge of blended modeling is the necessary non-trivial bi-directional synchronization between the AS and the different CSs, especially in the context of collaboration. In the context of versioning of blended models, neither the restriction to CS (as with git and SVN) nor to AS (as is common with visual languages) is helpful. Instead, versions must be tracked, conflicts detected and resolved at both levels. In this paper, we do not focus on blended modeling itself, but on a versioning approach that supports collaboration in a blended modeling context.

1.1. Contribution

This article presents a novel versioning approach that supports blended modeling by keeping the complexities of bi-directional change propagation between different CSs and the AS and the concurrent changes with all its facets such as branching, merging, conflict detection, and -resolution orthogonal. This is achieved by separately versioning CS, AS, and introducing a correspondence model (*Corr*) between them which allows non-trivial mappings from CSs to and from AS, branching and merging (including conflict detection) at the level of CS as well as the level of AS.

In our previous work [12], a high-level overview of our approach was introduced, while leaving many things abstract. For instance, we did not mention the supported types of edit operations (“deltas”), or the precise (*graph*) model data structure that these operations could be performed on. The main contribution of this paper is a set of concrete choices for those things that were left abstract in earlier work, and the resulting implementation of a demonstrator thereof. Finally, the Related Work section has been significantly extended.

The remainder of this paper is structured as follows: in Section 2, we introduce our running example, which exhibits all of the complexities addressed by our approach. The solution is then explained in a bottom-up manner: Section 3 explains why and how we persist changes to graphs, record dependencies and detect conflicts between them. Section 4 explains our notion of versions, and how they order sets of changes into a comprehensive edit history. Section 5 explains how we record evolving correspondence relations between CS and AS elements, resulting from bi-directional synchronizations. Section 6 presents the implementation of an interactive demonstrator of our the running example. Section 7 discusses related work, and Section 8 concludes the paper and discusses future work.

2. Running example

This section introduces a small example of blended modeling used throughout the paper to illustrate different complexities that our versioning approach addresses. It consists of an AS, a visual CS, and a textual CS. The example uses a minimal subset of Statecharts [13]. The AS of our formalism only consists of States. Every State can have at most one parent State. Further, no cycles are allowed in the parent-of relationship. Thus, for our example, we focus only on state hierarchy and omit all other elements of Statecharts such as initial states, transitions, and orthogonal states.

Similarly, the visual and textual CS are also very restricted. The visual CS consists of rounded rectangles (*rountangles* [14]) with geometries. It does not have any notion of a parent relation — instead, the geometries of rountangles between whose corresponding States a parent exists, must be such that the parent rountangle completely

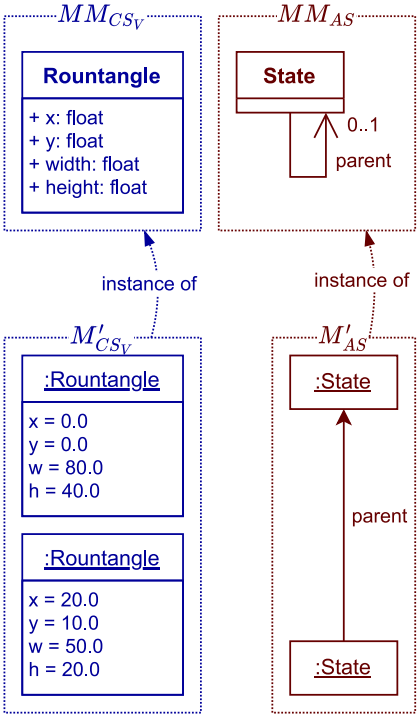

Fig. 1. Running example: meta-models and instances for CS and AS.

surrounds the child. On the textual CS, a state is realized by the token `state` followed by opening and closing curly braces surrounding the definitions of child states, if there are any. The upper half of Fig. 1 shows the meta-models of the AS (in red, on the right) (MM_{AS}) and visual CS (in blue, on the left) (MM_{CS_V}), and the lower half shows respective example instances. We refrain from providing a meta-model for the textual CS as it does not add to our story.

Fig. 2 shows a tiny blended modeling scenario. At the beginning, all models consistently contain exactly one state represented by a rountangle (M_{CS_V}), or by the text ‘‘`state {}`’’ (M_{CS_T}), and in the abstract syntax by a class of type “State” (M_{AS}). Starting from this initial version depicted on the left hand side, two users Alice and Bob[1] perform different edit operations concurrently: In visual CS, Alice adds a new rountangle inside the already existing rountangle, resulting in a new CS model M'_{CS_V}. In the textual CS, Bob removes the state from the initial version, resulting in the new CS model M''_{CS_T}.

The figure also shows the resulting AS models: in the initial state (M_{AS}) just one State object exists. Alice’s change can be *parsed* to produce a new State and parent link in AS, whereas Bob’s change can be parsed to produce a State deletion in AS. Each of these changes can in turn be *rendered* to the other (textual/visual) CS (respectively). Finally, the two new models (M' from Alice and M'' from Bob) have to be merged on each level (visual CS, AS, and textual CS). As one can see on the right hand side of the figure, the CS levels could be naively merged without giving rise to conflicts, but at the AS level a conflict is unavoidable because a parent link is created to a State that is concurrently deleted.

To resolve this conflict, a choice must be made: do we keep the deletion, or do we keep the parent link? Or something else entirely? This is however not the topic of this paper. Instead, we focus on a unified and efficient representation of (possibly concurrent) changes, whether these changes were triggered by user edit operations (on CS),

[1] We distinguish two different users, although it will not make a difference, if both operations are done by the same person without a synchronization in-between.

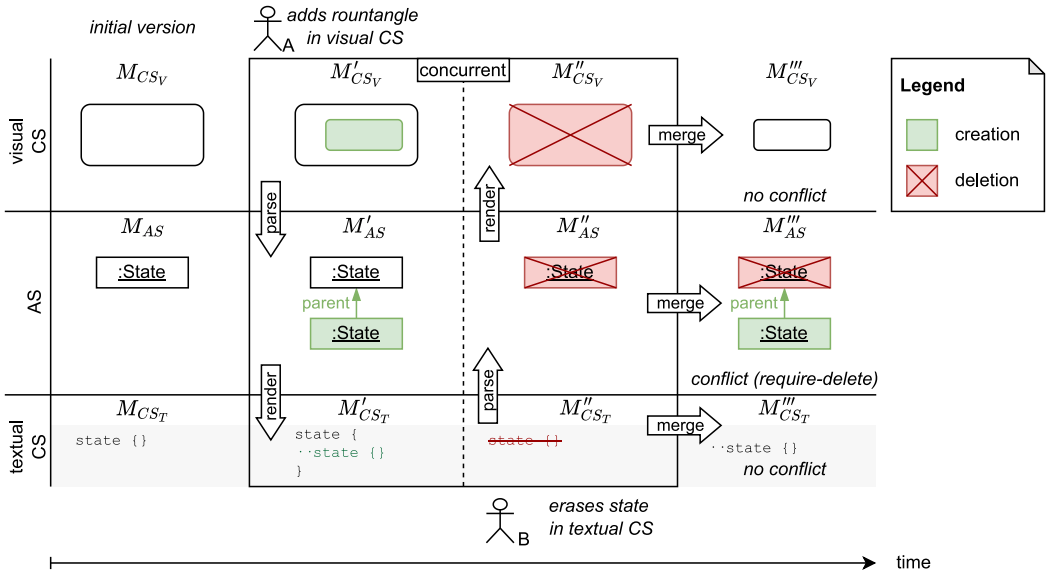

Fig. 2. Running example: Comprehensive scenario.

or by bi-directional synchronizations (between CS and AS). We will show how this allows for intuitive and fast conflict detection, and calculation of alternative conflict resolutions.

In the remainder of the paper, we assume that the operation of Bob (removal of text in M''_{CS_T}) has already been parsed, as if Bob altered the AS directly. This way, our example still covers the essential complexities of blended modeling: concurrency, and bi-directional change propagation (between rountangle geometries and parent links). Bi-directional change propagation between textual CS and AS would be of comparable complexity.

We are aware that this example is quite limited. We have chosen to use Statecharts because they are a widely used formalism and therefore well known. This saves additional effort to understand the examples in the rest of the text. We admit that an additional textual CS meta-model would be interesting in the context of blended modeling, but we wanted to focus on our collaboration approach and the additional complexity would be distracting to the reader.

3. Dependency-aware operation-based versioning

In this section, we first motivate that we use *operation-based* versioning, and then explain our particular implementation in detail: we reduce all model operations to combinations of simple (primitive) graph operations. We first explain what these primitive operations are, and then see how we can combine them into transactions. We see how we can efficiently detect conflicts, both on primitive operations and on transactions of operations.

3.1. Why operation-based?

State-based versioning systems (such as git) record snapshots of the entire model state, for every model version [15]. They have to resort to *diffing* (model differencing) to produce sets of changes relative to a common ancestor version, and then attempt to merge these changes. Operation-based versioning, on the other hand, simply records changes as they happen [15]. The model state for any sequence of recorded changes can be reproduced by *replaying* these changes. We have chosen the operation-based approach, because:

- Diffing is error-prone: it cannot detect higher-level edit operations (*transactions*) and instead reduces everything to primitive operations. [16].
- It scales better with large models and frequent versions [17]
- It can support synchronous collaboration.
- If not only operations, but also dependencies between operations are recorded, conflict detection becomes very simple and efficient to implement (see Section 3.4.1).

Operation-based versioning also comes at a cost: the versioning system needs to be integrated with the (CS) editor (to be notified of edit operations as they happen, typically via a plugin). We do hope however, that in the future, a standard API (similarly to the language server protocol[2] [18]) can reduce this integration cost, and even relieve editors from having to implement features such as undo and synchronous collaboration themselves.

3.2. Graph state meta-model

We have decided upon a very simple graph meta-model. It consists of *nodes*, *values*, and *edges*:

Nodes are created and deleted. They are identified by an immutable globally unique identifier (GUID). Once deleted, the same node (with the same GUID) can never be created again.

Values are immutable numbers, strings, booleans, and the special value *null*. They are not created and deleted — they always already exist. (This is similar to how the graph transformation tool GROOVE [19] treats data values.)

Edges are directed, and connect a source (= a node) to a target (= a node or a value). An edge is identified by its source node and immutable *label* (= a string), meaning that for a given source node and label, a unique edge exists. Initially, all edges have as target *null*. If an edge's target is a node, that node must have been already created, and not yet deleted.

Allowing only one edge for a given source node and label is on purpose: we use edges to model object properties and we want to support *true updates* of their value, meaning that concurrent updates (to different values) should be conflicting. The alternative, representing edge updates by deletions and re-creations of an edge (with an "updated" target), would have to permit multiple edges with the same source and label. This could lead to *inconsistencies* when using edges to model object properties (an object could have multiple values for the

[2] https://langserver.org

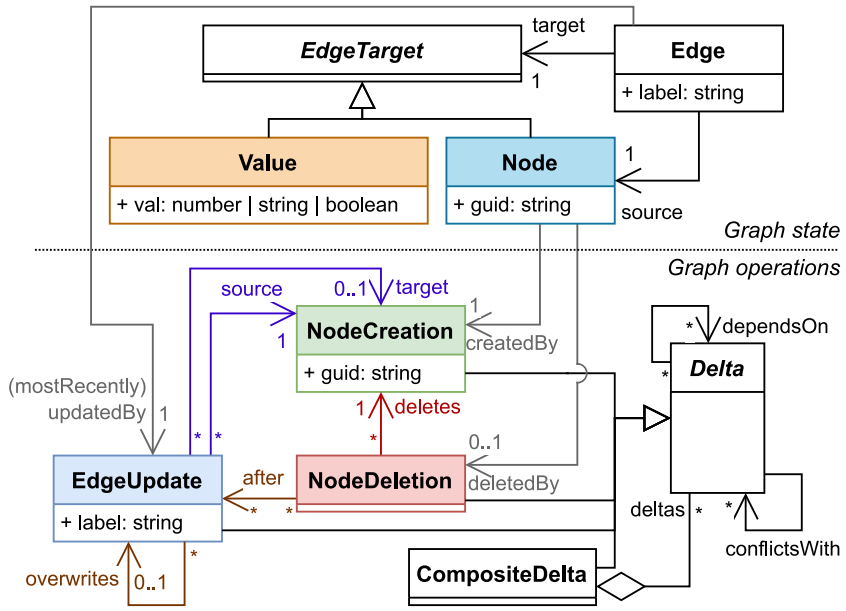

Fig. 3. Meta-model of graph state and graph operations.

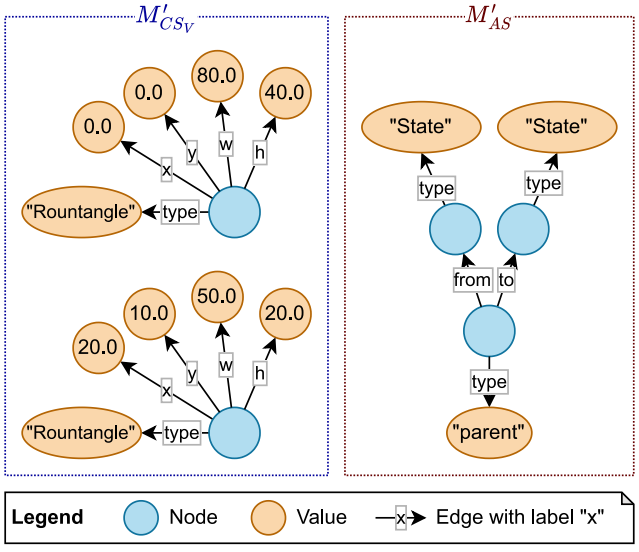

Fig. 4. Running example: M'_{CS_V} and M'_{AS} from Fig. 1 as graph instance.

same property). We prefer dealing with conflicts over inconsistencies, because an inconsistency is an invalid *state* and contains no information on who and what caused it. This is opposed to conflicts, which are invalid pairs of *operations* that intrinsically contain information about the author and cause.

Further, we can store the outgoing edges of every node in a dictionary, indexed by their label. This allows for reasonably efficient property lookup (similar to Python).

Attributes of more complex types (ordered sequences, (multi)sets, dictionaries) can also be represented. For a dictionary, the attribute edge simply points to another node, whose outgoing edges represent the dictionary entries. Similarly, a set can be represented by a node, with an outgoing edge for every element. The labels on the edges can be GUIDs to represent a multiset, or derived from the set elements (e.g., some hash function) to represent an ordinary set. Ordered sequences can be represented in graph form as (doubly) linked lists, or by a dictionary whose keys are array indices. The former will have better support for (concurrent) random updates.

The upper part of Fig. 3 shows our graph meta-model. An instance of such a graph state is shown in Fig. 4. Each object of the object diagram in Fig. 1 is represented as a (blue) node with edges to (orange) values. The graph state edges are labeled with the corresponding attribute names of objects. Links in the object diagram (e. g., the parent relation between in M'_{AS}) are also represented as nodes in the graph state with three edges: the "type" edge defines the type, the "from" and "to" edges define the connected objects and according to the labels the direction of the corresponding object diagram link. review3-comment4Before we look at the different types of deltas, we explicitly state that all *models* can be represented as *graphs* [20], and thus, every type of CS or AS can be mapped onto the primitive graph structure introduced in this section.Even plain text can be represented as a linked list of characters, which is also a graph.

3.3. Deltas, dependencies, and conflicts

We now explain our implementation of operation-based versioning. Central to our approach is the concept of a *delta*. A delta is a *transaction* of changes on a graph (representing a model), meaning the delta is either fully executed, or not at all. A delta can be caused by a user edit operation, or as the result of parsing/rendering a change that happened elsewhere.

Deltas can have *dependencies* on other deltas. Deltas and their dependencies form an append-only, directed acyclic graph (DAG), which we call a *dependency graph*. A delta δ_b depends on another delta δ_a, if δ_a happened before δ_b and if δ_b *requires*, *overwrites*, or *deletes* a result that was produced by δ_a.

A pair of deltas is *concurrent* if one does not (transitively) depend on the other, or vice-versa. Concurrent deltas are either *commutating* or *conflicting*. Commutating deltas can be executed in any order: the effect is always the same. Conflicting deltas are all other cases: either the order of execution matters (e. g., they would overwrite each other), or no meaningful order of execution exists (e.g., one deletes something that is required to exist by the other). Conflicting deltas will always have a common dependency, as we will see in Section 3.4.1. We exploit this property to find conflicts efficiently.

3.4. Primitive and composite deltas

We will introducetwo classes of deltas: *primitive* and *composite*. Primitive deltas are the tiniest operations on instances of our graph

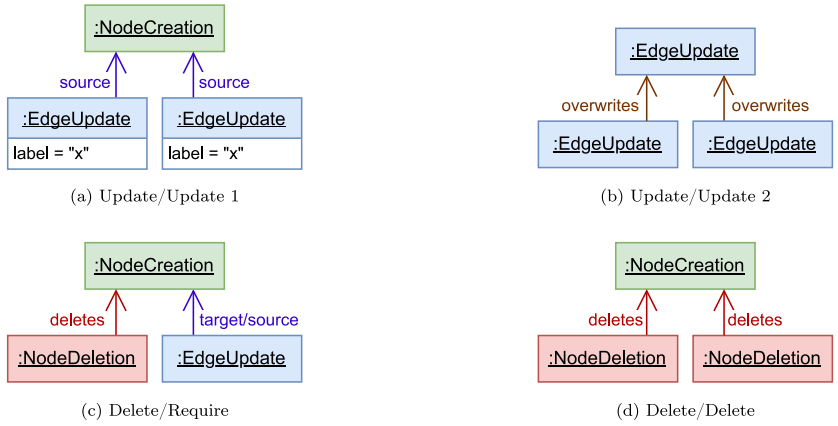

Fig. 5. Conflict types between primitive deltas.

meta-model, from which all other operations are constructed. Only a few types of primitive deltas exist, so we can exhaustively define the dependency and conflict types that can occur between them. Composite deltas are transactions consisting of any number of smaller (primitive or composite) deltas. The dependencies and conflicts that occur between composite deltas are derived from the deltas they consist of.

3.4.1. Primitive deltas

The following three types of *primitive deltas* exist: NodeCreation, NodeDeletion, and EdgeUpdate. Each of these has its own dependency types as it can be seen in the lower part of Fig. 3.

NodeCreation creates a node with a newly generated GUID.

Dependencies: NodeCreation has no dependencies.

EdgeUpdate sets the target of an outgoing edge to another node or a value, or it deletes the edge by setting the target to *null*. If an EdgeUpdate deletes an edge, the edge can be re-created in a follow-up EdgeUpdate.

Dependencies: EdgeUpdate depends on the NodeCreation of the source and new target (if there is one) of the edge and on the previous, most recent EdgeUpdate of the same edge (i.e., same source and label) that it *overwrites*.

NodeDeletion deletes a node. The node must not have any incoming edges.

Dependencies: NodeDeletion depends on the NodeCreation of this node. If there ever was an EdgeUpdate for which this node was the source or target, it depends on a follow-up EdgeUpdate that sets this edge to another target (possibly *null*), to ensure that an edge never points to a deleted target.

For these three primitive deltas, there exist exactly three *conflict types*. Every conflict type involves an overlapping dependency between the conflicting operations:

Update/Update occurs whenever two EdgeUpdates independently (un)set the target of the same edge, see Figs. 5(a) and 5(b).

Delete/Require occurs between an independent pair of EdgeUpdate and a NodeDeletion, when the deleted node is the source or new target of the edge, see Fig. 5(c).

Delete/Delete occurs whenever two NodeDeletions delete the same node, see Fig. 5(d).

Table 1
Conflict types (if any) between all possible combinations of primitive deltas.

Delta Type	NodeCreation	EdgeUpdate	NodeDeletion
NodeDeletion	*never conflicting*	Fig. 5(c)	Fig. 5(d)
EdgeUpdate	*never conflicting*	Figs. 5(a) & 5(b)	
NodeCreation	*never conflicting*		

Table 1 systematically lists all possible conflict types for each pair of primitive delta types. Additionally, Fig. 5 visualizes the conflict types.

Note that for each conflict type, the conflicting deltas have a common dependency. When a new delta δ_b is created, we can efficiently detect newly introduced conflicts by navigating in the inverse direction the incoming dependencies of δ_a, where δ_a is a dependency of δ_b.

In the theory of prime event structures, if δ_b depends on δ_a, and if δ_a and δ_c are conflicting, then δ_b and δ_c are also conflicting [21]. In our approach, we ignore such "derived" conflicts because (1) the number of conflicts would explode, which would be problematic when visualizing them, or keeping track of them in our implementation. (2) The possible *configurations* (defined as left-closed, non-conflicting sets of deltas [21]), remain identical under our definition of "conflict".

3.4.2. Composite deltas

A single user edit operation on a model often consists of many primitive deltas. Likewise, a single parsing/rendering invocation may produce a set of changes consisting of many primitive deltas. These primitive deltas belong together: we are only interested in their collective effect. In other words, we want to treat them as *transactions*. Such a transaction can again be understood as a delta: it may have dependencies on, and conflicts with, other transactions. A *composite delta* is a set of deltas that is treated as a delta itself.

We call the set of all primitive deltas, their dependency and conflict relations the "level 0" (L0) dependency graph. This set can be partitioned into composite deltas, that form a "level 1" dependency graph. A L1-delta δ_b has a dependency on another L1-delta δ_a if at least one of the L0-deltas of δ_b has a dependency on one of the L0-deltas of δ_a, and likewise for conflicts. The partitioning must be such that no dependency cycles occur at L1, and such that a composite never contains internally conflicting deltas. Following the same rules, the set of all L1-deltas could be partitioned again to produce an L2 dependency graph. Higher-level dependency graphs are *abstractions* of lower-levels. The implementation of our running example contains two levels: L0 and L1.

Fig. 6 shows an example graph state after adding a rountangle in the editor. This graph state is the result of executing the created graph state operations (as defined in the meta-model in Fig. 3). Since the creation of a rountangle is an atomic operation and therefore all subsequent

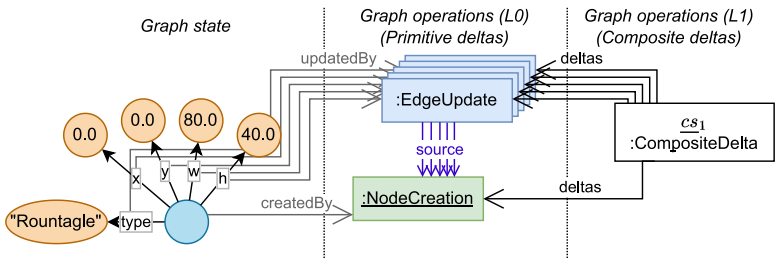

Fig. 6. Running example: fragment of two levels of deltas, and the graph state produced by them.

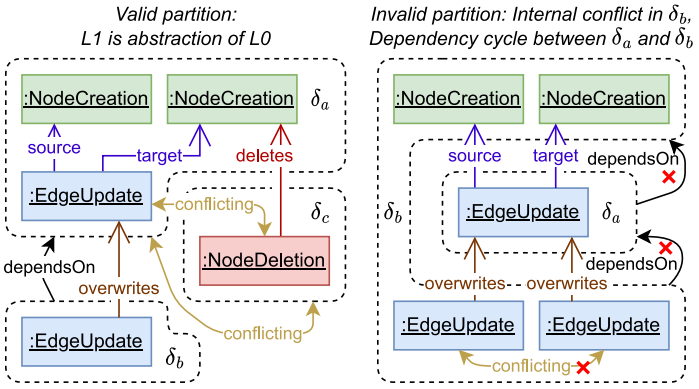

Fig. 7. Higher-level dependency graphs: valid and invalid partitions.

operations depend on the entire set of primitive (L0) deltas, a combined composite (L1) delta cs_1 is also created.

Fig. 7 shows an example of a valid and an invalid partition. In the valid partition, δ_c and δ_a are conflicting because there exists a conflict between the primitive deltas they consist of. Likewise, there exists a dependency from δ_b to δ_a because at least one dependency exists between their primitive deltas. The invalid partition contains two inconsistencies: an internal conflict between the primitive deltas of δ_b and a dependency cycle between δ_a and δ_b.

4. Versions

We have now seen how we can represent user edit operations on a graph as (composite) deltas. The dependencies between these deltas define only a partial ordering on their possible executions, but this does not yet show the full picture of the edit history of a graph. For instance, commutating deltas are unordered with respect to each other, but the user may have executed in a specific order. We therefore introduce another data structure, called the *history graph*.

4.1. History graph

The *history graph* records the order in which deltas were executed. The nodes of the history graph nodes are called *versions*. Conceptually, a *version* is an (unordered) set of deltas. When the deltas in this set are replayed in any order consistent with the dependencies between the deltas, this always results in the same graph state. The set is not allowed to contain conflicting deltas, and it must be *left-closed*. A set S of deltas is *left-closed* if for every delta $\delta \in S$, all dependencies of δ are also in S [21].

A version can have any number of *parent*-links to other versions. A version's parent is the version that came immediately before it. Versions and their parent-links form another DAG (similar to git's commit history).

The history graph is constructed as follows. We always start from an initial empty version (consisting of no deltas). We create a new version (and parent-link to the previous version) after every new edit operation. This allows us to implement *undo/redo* at the level of the versioning system: it is simply the act of going back/forward one version in the history graph. Since every parent-link represents an edit operation (typically a composite delta) that is applied to the parent, we actually store this delta in the parent link. To know the deltas that a version consists of, we simply follow a path of parent links until we reach the initial version.

4.2. Branching

When multiple versions have the same parent, we say that history is branched at the parent. Every child of the parent is called a branch. Multiple branches may be the result of concurrent edit operations (by multiple users), or of undo (going back one or more versions), and making changes again.

4.3. Merging

We define a merge-function, that takes as input $0..*$ versions, and produces as output $1..*$ versions[3]. It first takes the union U of all the deltas of the inputs, and then constructs all *maximal non-conflicting subsets* $S_i \subseteq U$, meaning that, no delta $\delta \in U$ can be added to any S_i without introducing a conflict in S_i. Every S_i is an output version. If no conflict exists in U, then U is simply returned. Otherwise, multiple outputs will be produced, that can be presented to the user as alternatives for resolving a conflict.

5. Synchronization of CS and AS

Until now, we have seen how deltas and versions can be used to record a history of changes to a single graph. This graph could represent

[3] Merging zero versions produces as output the initial (empty) version, which is consistent with set theory, where the union of zero sets is the empty set.

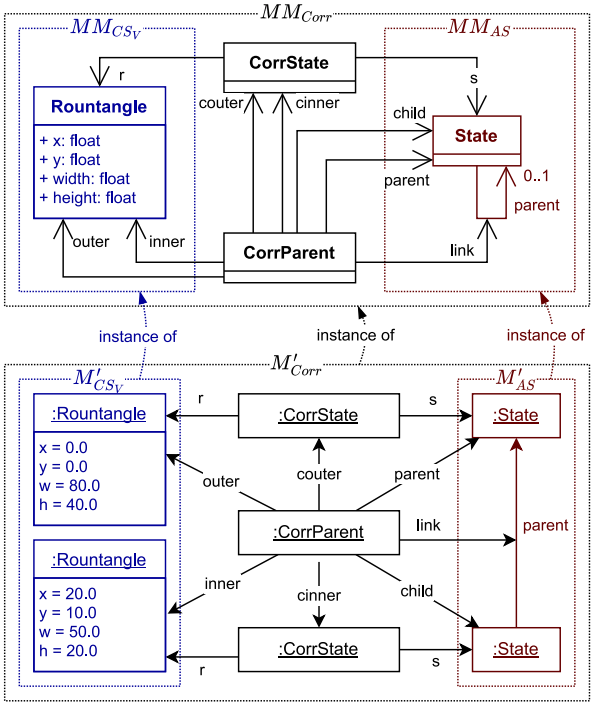

Fig. 8. Running example: meta-models and instances for CS, AS, and correspondence.

the CS or AS of a model. We will now see how we can propagate changes between CS and AS, and record these change propagations as part of the history of the affected graphs.

5.1. Motivation for incremental change propagation

Today, most tools for authoring models (and code) support only non-incremental parsing (and sometimes non-incremental rendering as well). The parser takes as input a snapshot of a CS instance, and produces as output a snapshot of an AS instance (vice versa for the renderer). Incremental parsing and rendering, on the other hand, takes *changes* (to a model) as input, and produces *changes* (to another model) as output.

There are several benefits to incremental parsing and rendering. Firstly, it can be more efficient: for instance, if a set of CS model elements is changed, only those AS model elements that are involved in some sort of correspondence relation with the altered CS model elements should be updated when parsing, and vice-versa for rendering. In other words, we do not have to parse/render a model from scratch after every change. Secondly, it can support *layout continuity* when rendering: by only updating those parts of the CS that have changed, and not generating a new layout from scratch, the user can continue working with the layout that (s)he is familiar with. Layout continuity applies to textual CS as well: whitespace, and the order of model element declarations can be considered "textual layout". Finally, and perhaps most importantly, incremental parsing and rendering integrates naturally with an operation-based versioning system. In our case, we can uniformly persist all changes to CS and AS, whether they represent user edit operations or change propagations between CS and AS, in the form of deltas, as explained in Section 3. This makes merging and conflict detection much easier [15].

5.2. Correspondence model

To support incremental parsing and rendering, we must record which CS elements correspond to which AS elements. For instance, in our running example, we must record which Rountangle corresponds to which State. We persist these fine-grained correspondence links in a *correspondence model*, an idea taken from Triple Graph Grammars [22]. A correspondence model instance contains all the elements from one CS and one AS instance, plus all the necessary correspondence links between them. For each CS and AS meta-model between whom parsing and rendering should occur, a correspondence meta-model (also called correspondence *meta-triple* in the literature [23]), needs to be defined.

The correspondence meta-triple for our running example is shown in Fig. 8 which extends Fig. 1 with MM_{Corr} in the upper half and M'_{Corr} in the lower half. MM_{Corr} contains the entire meta-models of CS and AS, and additionally defines so-called *correspondence classes* that relate CS and AS elements. The CorrState correspondence class relates one Rountangle to one State, and CorrParent relates two Rountangles to a parent link between their respective States, on the condition that the Rountangles are geometrically nested (this geometric constraint is not shown in the figure). The lower half of Fig. 8 shows an instance of a correspondence model (in this case, M'_{Corr} of our running example). This instance contains one CS and one AS instance, and it contains three *correspondence objects*: two instances of CorrState and one instance of CorrParent.

5.3. Co-evolution of CS, AS, correspondence

As CS and AS model instances evolve, so do correspondence relations between their elements. Therefore, the correspondence model is versioned as well. Every correspondence instance version contains one CS instance version, and one AS instance version, and the correspondence relations between them.

New versions of correspondence instances are created as the result of parsing or rendering changes to CS or AS respectively. Fig. 9 shows the signature of the parser and renderer functions. The parser is a function that takes as input a (e.g., composite) CS-delta (+ the resulting new CS version), and the CS, AS and Corr versions to which this change should be applied. It produces as output an AS-delta (+ the resulting new AS version) and a Corr-Delta (+ the resulting new Corr version). The renderer function has a similar signature, with the roles of CS and AS swapped.

In this paper, we only provide the interface that parsers and renderers must conform to. Their (many) possible implementations are out-of-scope.

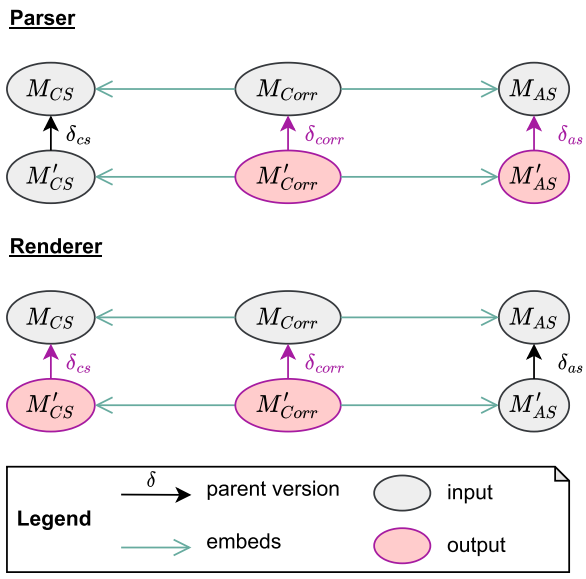

Fig. 9. Inputs and outputs of the parser and renderer functions.

5.4. Full scenario

We now explain how the full scenario of our running example can be represented by all the building blocks explained until now. We quickly summarize: primitive deltas are grouped into transactions that we call composite deltas. Versions and parent links define an execution order on deltas. We separately version CS, AS, and the correspondence model. New versions are produced by user edits (CS, AS), and by the parser and renderer (CS, AS, Corr). Finally, we will see how the merge function can be applied at the level of CS, AS and correspondence.

Initial conditions. At the top of Fig. 11, we see the three meta-models MM_{CS}, MM_{Corr}, and MM_{AS}. Right below them, we see the initial version $\emptyset$, which axiomatically contains no deltas, has an empty graph state, and conforms to all three meta-models. On the next row, we see the versions $M_{CS_V} = \{cs_1\}$, $M_{AS} = \{as_1\}$, and $M_{Corr} = \{corr_1\}$, that we assume to already exist. Fig. 10 is an extension of Fig. 6, and shows in detail the L0-deltas that the L1-deltas cs_1, as_1, and $corr_1$ each consist of. Here, cs_1 creates the outer Rountangle. Likewise, as_1 creates a State object. Finally, $corr_1$ contains all the L0-deltas of cs_1 and as_1, along with L0-deltas for the creation of a CorrState object relating the Rountangle and the State objects.

Note that from the viewpoint of the correspondence model, the Rountangle, the State, and their correspondence links are created in a single, atomic transaction ($corr_1$). In our current implementation, we do not record whether the change to CS or AS came first (i.e., the *cause* of the correspondence relation), because this information is not strictly necessary to support blended modeling. We are aware that this information may have value (e.g., empirical) nonetheless.

User edits. Continuing the scenario of our running example, Alice creating the inner Rountangle in visual concrete syntax results in the new L1-delta cs_2, and produces a new CS version $M'_{CS_V} = \{cs_1, cs_2\}$, whose parent is M_{CS_V} (see Fig. 11). The new delta cs_2 has no dependencies (Fig. 12 shows the dependencies between all L1-deltas of our running example). Bob's (concurrent) deletion of the State is captured in delta as_3, which results in $M''_{AS} = \{as_1, as_3\}$, whose parent is M_{AS}. The new delta as_3 has a delete-dependency on as_1.

Parsing Alice's change. When parsing Alice's change, the parser inputs are (1) M'_{CS_V}, and its parent link to M_{CS_V} with delta cs_2, and (2) M_{Corr}. It produces as output a new L1-delta as_2 (and AS version M'_{AS}) that creates the corresponding State and parent-link to the existing State. This delta depends on as_1, because it creates a link to an object created by as_1. The parser function also produces a new L1-delta $corr_2$ (and Corr version M'_{Corr}, containing the changes to both CS, AS as well as the creation of CorrState. Note that $corr_2$ depends on $corr_1$ (Fig. 12).

The newly created version M'_{AS} is concurrent with M''_{AS}, as shown in Fig. 11. Furthermore, a conflict exists between these concurrent versions, because as_2 and as_3 are conflicting. As explained before, conflicting deltas always have a common dependency. In this case, they both depend on as_1 (State creation). as_2 creates a link to the State, but as_3 deletes it. The consequences of this conflict will be discussed later, when we explain *merging*.

Rendering Bob's change. When rendering Bob's change, the renderer inputs are (1) M''_{AS}, and its parent link to M_{AS} with delta as_3, and (2) M_{Corr}. It produces as output a new CS L1-delta cs_3 (and CS version M''_{CS}) that deletes the outer Rountangle. This delta depends on cs_1 (which created the now-deleted Rountangle). The renderer function also produces a new L1-delta $corr_3$ (and Corr version M''_{Corr}), that in a single transaction deletes the Rountangle, State, and the correspondence link between them. This delta also depends on $corr_1$.

Looking at Fig. 12, one may wonder why $corr_3$ does not "reuse" the deltas cs_3 and as_3. The reason is that, in the correspondence model, the deletion of CorrState and its outgoing links must happen logically before the deletion of the targets of these links (Rountangle and State). Otherwise, the edges representing CorrState's outgoing links would point to deleted nodes, which is forbidden in our graph state meta-model (Section 3.2). To force a correct order of deletions, we must "insert" a dependency into cs_3. We cannot alter cs_3 itself, because that would violate the *append-only* nature of our delta graph data structure. Therefore, in the context of the correspondence instance, we *override* cs_3 with an identical delta cs'_3 with the added dependency on the CorrState deletion. The existing delta cs_3 is still used in the context of the CS instance (which has no notion of CorrState). The same reasoning holds for as_3 and its overriding delta, as'_3.

Merging Alice's and Bob's changes. We now look at two possible applications of the merge-function (defined in Section 4.3). We can (naively) merge Alice's and Bob's changes at the level of CS. The merge function's inputs are then $M'_{CS_V} = \{cs_1, cs_2\}$ and $M''_{CS_V} = \{cs_1, cs_3\}$. The union of the (L1-)deltas of these versions does not contain any conflict, so the output is simply the union of all deltas $M'''_{CS_V} = \{cs_1, cs_2, cs_3\}$. Of course, merging at the level of CS ignores the meaning of the model. It is better to merge at the level of the AS. The merge function's inputs are now $M'_{AS} = \{as_1, as_2\}$ and $M''_{AS} = \{as_1, as_3\}$. Because of the conflict between as_2 and as_3, the maximal non-conflicting subsets of $\{as_1, as_2, as_3\}$ are $\{as_1, as_2\} = M'_{AS}$ and $\{as_1, as_3\} = M''_{AS}$. In other words, the outputs are identical to the inputs, and therefore, a choice must be made whether to accept Alice's changes, or Bob's (which was to be expected).

In general concurrent blended modeling scenarios, we envision always merging at the level of the correspondence model. The rationale is as follows: conflicts in both CS (e.g., disagreement on layout) and AS (e.g., the conflict of our running example) will be present as conflicts in the correspondence model, resulting in the merge-function outputting all alternatives to consider. Each alternative (= a correspondence instance) can be presented to the user, since its CS is simply part of it. Fig. 13 shows our envisioned procedure for a visual and textual CS. Note that because correspondence models are CS-specific, in order to merge at the correspondence level, we must first render all concurrent changes to one CS (in the figure, we have chosen the visual CS).

5.5. Decentralized conflict resolution

Fundamentally, we consider conflict resolution to be decentralized: if collaborators disagree on how a conflict should be resolved (e.g., one user chooses $\{corr_1, corr_2\}$ and another user (concurrently) chooses $\{corr_1, corr_3\}$), then the conflict is re-introduced as soon as

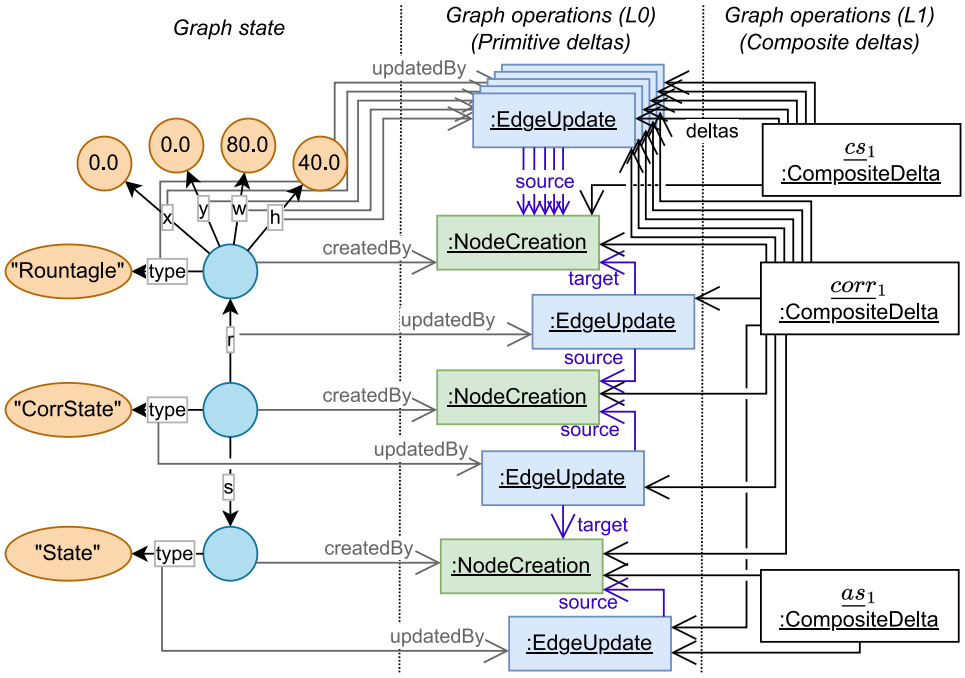

Fig. 10. Running example: L0- and L1-deltas of CS, AS and Corr (extension of Fig. 6).

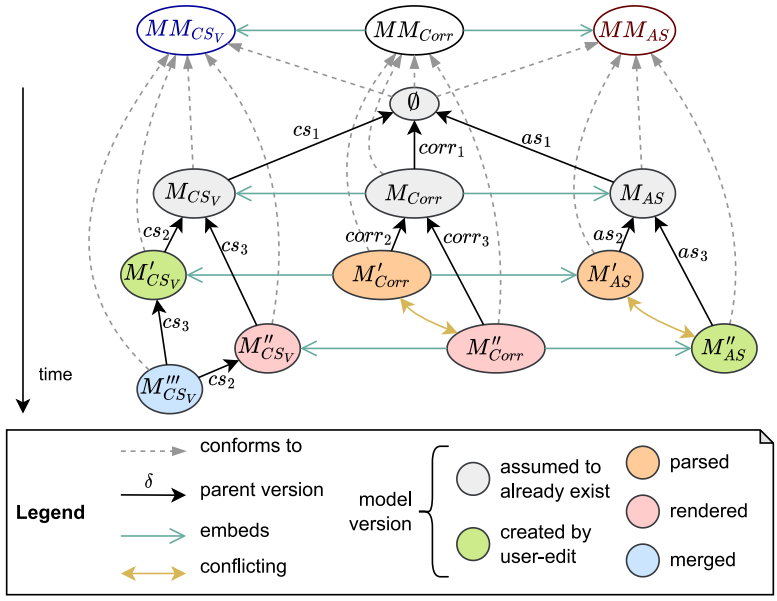

Fig. 11. Running example: versions of CS, AS, and Corr.

their changes are merged once again. Automated conflict resolution (e.g., giving higher priority to a certain type of operation, or to a specific user) can be used as well, as long as all collaborators run the same (deterministic) conflict resolution algorithm. The precise implementation of such an algorithm is out-of-scope.

5.6. Postponed rendering, non-blocking parsing

We have seen that the result of parsing and rendering is simply the creation of new AS/CS (resp.) and correspondence instance versions. The outputs of parsing/rendering are persisted forever, but so are the inputs: this means that parsing and rendering do not need to happen immediately and that they are *non-blocking* procedures.

When rendering is non-deterministic, it benefits from the ability of postponing it. For instance, when a renderer needs to generate some layout, this could be done entirely by (complex) algorithms, but we believe that some level of human interaction will typically give better results. Further, we want to always support fully manual rendering as a fallback,[4] in case no automated renderer is available. Supporting human input is only practical if we can postpone rendering of a CS until that CS is actually opened/requested by a user. Then only at that point

[4] In this case, we can automatically *verify* if the result is correct by parsing it and comparing it to the original AS. This works as long as parsing is deterministic.

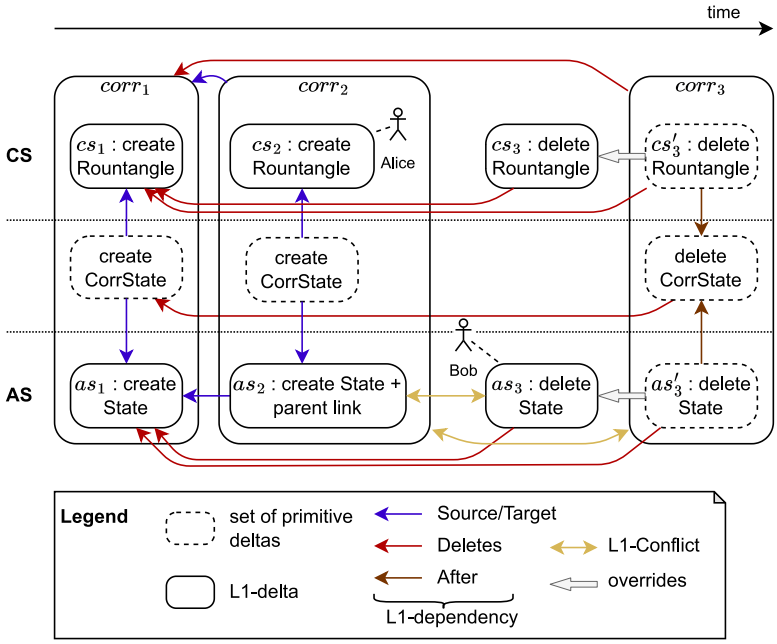

Fig. 12. Running example: L1-deltas, dependencies, and conflicts.

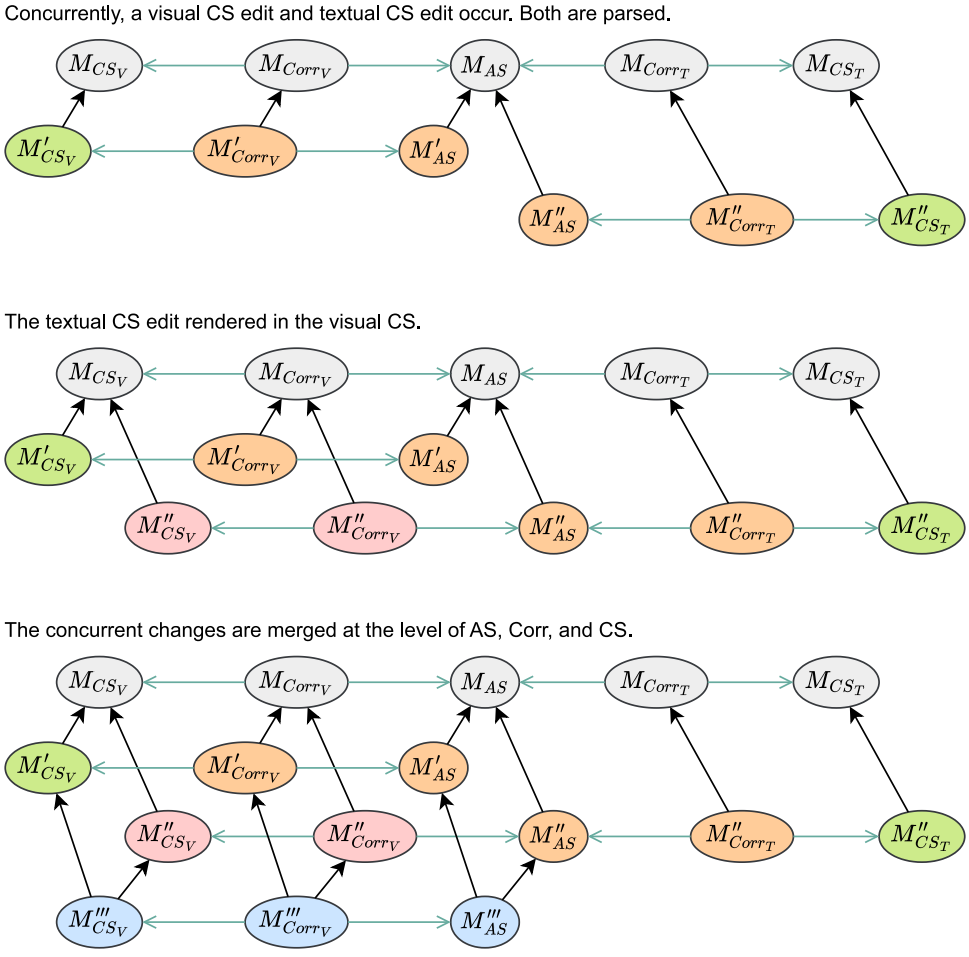

Fig. 13. Example: Merging non-conflicting changes to different CS's.

in time do we have to ask the user for her input (e.g., by letting her choose from some auto-rendered alternatives).

We probably want to parse every change as soon as possible, but parsing still benefits from the non-blocking aspect: when the latency of parsing becomes high, the user can continue making changes to the CS, without having to wait each time for the parsing to complete.

6. Implementation

We developed a proof-of-concept implementation of our novel versioning approach. We created a web-based demonstrator that can be easily tried out interactively directly in the web-browser.[5] [The source code is available online],[6] too.

The "core" concepts, like the different types of primitive deltas, their dependencies and conflicts, versions, and graph state, have been implemented as a (reusable) Typescript library.

Our web-based demo, that uses this library, was created with React. Wherever graphs had to be rendered, we used the D3.js library with an animated spring layout. Following the idea of rapid prototyping, we created this first proof-of-concept from scratch which enabled us to fail and — in turn — learn fast. Using existing modeling frameworks such es EMF, would instead require significantly more effort in refactoring the EMF code-base and shifting the versioning paradigm from state-based to operation-based. Nevertheless, we are convinced that our approach could (theoretically) be integrated into EMF.

6.1. Library

We now explain the most important building blocks from our reusable TypeScript library.

6.1.1. Deltas

Deltas, once created, are immutable objects in memory. They have *content-based IDs*, which users of git may be familiar with: The ID of a delta is computed as the hash of its properties and dependencies. For instance, the ID of an EdgeUpdate is computed from (1) its type ("EdgeUpdate"), (2) the ID of the NodeCreation delta of the source of the edge, (3) the label of the edge, and (4) the ID of the previous EdgeUpdate that is overwritten (if there is any). Apart from serving as GUIDs, a side-effect of using content-based IDs is "free" de-duplication, since identical deltas will have the same ID. If two users would concurrently delete the same node, they would effectively be creating the same delta (not resulting in a conflict).

The in-memory representation of deltas and their dependencies is doubly-linked. This allows navigating dependencies also in their reverse order. This mechanism is used to detect conflicts with other deltas (conflicting deltas always have a common dependency). As soon as a delta is created, we check if it is conflicting with any of the other (already created) deltas, and add this information to both deltas involved in the conflict.

Serialization. When serializing a delta (not yet implemented), we only have to write out (1) its type, (2) its attributes (e.g., the label of the edge in case of an EdgeUpdate), and (3) the IDs of its dependencies. From this, all other information (its ID, its involvement in conflicts, its dependent deltas) can be derived. Hence, serialized deltas are small, that can be efficiently *streamed* over a network, supporting synchronous collaboration.

[5] https://sp2.informatik.uni-ulm.de/onioncollaboration/

[6] https://msdl.uantwerpen.be/git/jexelmans/onioncollab

6.1.2. Versions

Just like deltas, versions are immutable objects in memory, and their IDs are *content-based*. The ID of a version is computed as the bitwise XOR of the IDs of all the deltas that a version consists of. Since the XOR-operator is commutative, this matches our notion that a version is an unordered set of deltas. Also here do we get a form of de-duplication for free: if different users would execute the same commutating operations in a different order, they would effectively arrive at the same version, without even having to merge each other's changes.

Serialization. When serializing an individual version (not yet implemented), we only have to write out its parent links to other version. For every parent link, we write (1) the ID of the delta that is associated with the link, and (2) the ID of the parent version. Just like deltas, serialized versions are small, and can be streamed, supporting synchronous collaboration.

The total size of a stream of deltas and versions (which can be written to disk) will be linear with the total amount of edit operations over time. We envision always storing the full edit history on disk.

6.1.3. Graph state

Our library contains a class GraphState (see Fig. 14), which represents a mutable instance of our graph state meta-model (Section 3.2). The graph state can only be altered (in-place) by executing a delta on it. The class also allows "un-executing" (reverting the effect of) a delta, which allows us to navigate the version history in reverse order ("undo"). The class NodeState represents the mutable state of a node in a GraphState. It can be queried for the current incoming and outgoing edges. It also exposes two methods that help in the creation of deltas for updating an outgoing edge, and for deleting the node. The reason for having these methods, is that the manual creation of the deltas involved in these common operations can be fairly complex (many dependencies to provide). For instance, when deleting a node, the targets of all incoming edges of that node have to be set to *null* with an EdgeUpdate, and the NodeDeletion must depend on all these EdgeUpdates, as well as on the most recent EdgeUpdates of all the outgoing edges, and on the NodeCreation that created the node. These methods only create the deltas; they do not execute them. This is the caller's responsibility (see Fig. 14).

Serialization. It is not strictly necessary to be able to serialize a graph state. For any version, its associated graph state can be generated by replaying all the version's deltas. However, it may be a good idea to serialize also the graph state periodically. This will allow faster random seeking through the version history, analogous to *keyframes* in video codecs. The graph transformation tool GROOVE [19] also uses this technique.

6.2. Demonstrator

We now explain our web-based demonstrator. We first give an overview of its functionality, followed a brief overview of its architecture. Finally, we motivate a number of choices made during its implementation.

6.2.1. Demos

Our demonstrator actually consists of a small number of demos, that are intended to be followed in order. Each demo adds a new feature compared to the previous one. The demos are:

Demo: Graph State In this demo, the user can edit a generic graph (create/delete nodes + edges) directly, and observe the produced deltas and versions. (screenshot: Fig. 15)

The user can navigate between versions with undo/redo, or click on any version to navigate to it immediately. At any point in time, in the delta view, the deltas that make up the currently active version are rendered in **bold**.

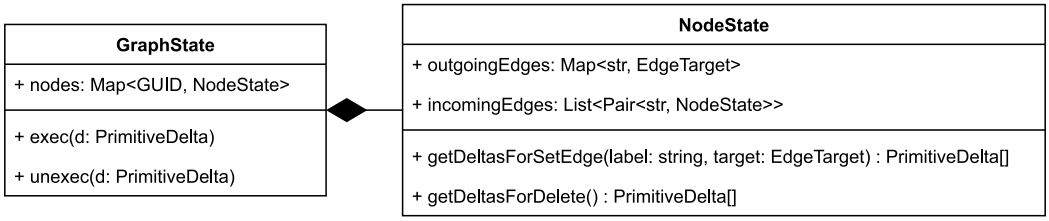

Fig. 14. GraphState and NodeState classes.

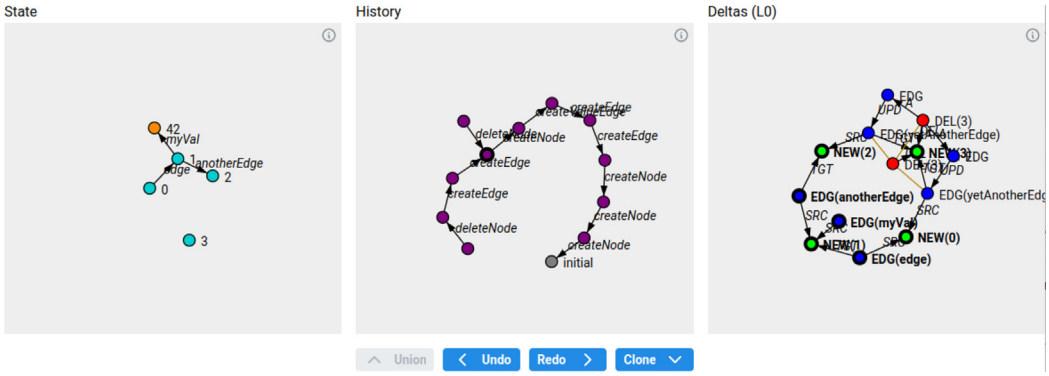

Fig. 15. Screenshot of our "graph state" demo.

Demo: Concrete Syntax This demo extends our first demo by adding a minimalistic rountangle editor, where the user can create, delete, move, and resize Rountangles. The underlying state of the rountangle editor is captured by an evolving graph state instance. The graph state can be observed (but not edited directly, as in the first demo). Every user action in the rountangle editor is captured by a number of primitive deltas (observable in the "Deltas (L0)" view), which are grouped into one composite delta (observable in the "Deltas (L1)" view).

Demo: Correspondence In this demo, there are three graph state instances: CS, AS, and correspondence (matching our running example). The CS instance can be edited via a rountangle editor (introduced in the previous demo). The AS instance can be edited by directly manipulating its graph state (as in the first demo). The graph state of the correspondence instance is read-only. (screenshot: Fig. 16)

Every edit operation on CS/AS can be parsed/rendered, to produce a new version of the correspondence model and AS/CS, respectively. Parsing and rendering can be set to occur automatically after every user edit, or to be manually triggered, via the "Auto" on/off switches at the top. By default, parsing (which is deterministic) happens automatically, while rendering is manually triggered.

Certain AS operations, such as creating a new State, or adding/removing a parent-link, cannot be rendered deterministically: there are infinitely many valid ways to (incrementally) update the CS layout. When non-determinism occurs, our renderer will create/destroy the necessary Rountangles for us, but it will ask the user to update their geometries to her taste. The user's input is only accepted if it is consistent with the AS update (screenshot: Fig. 17).

Demo: Blended Modeling This is our most complex demo. It is similar to the previous demo, but with two CSs, corresponding to a single AS. Currently, both CSs are of the same type (2D rountangles), but in the future, we want to add support for a textual syntax. (screenshot: Fig. 18)

6.2.2. Architecture

Fig. 19 shows the interaction between UI components, deltas, and the (in-memory) graph state. Both CS and AS can be edited by the user. When the user performs an edit operation, this operation does not directly change the (internal) state of the editor. Instead, the action is translated into a new set of primitive deltas combined in a composite delta, and a new version of the CS/AS. The new deltas are applied to the (in-memory) CS/AS graph state, which in turn causes an update in the editor state.

Parsing and rendering can be set to happen automatically after each user action, or to be manually triggered. As we have seen in Section 5.3, parsing and rendering creates new deltas and versions of the correspondence model, in addition to AS/CS. The new correspondence deltas are applied to the graph state of the correspondence model, which is in turn visualized in a graph view component.

6.2.3. Implementation choices

Simulated concurrency. Once the demo has been loaded in the browser, it runs completely local and "offline", and single-threaded. Rather than building a "networked" demo, we choose to *simulate* concurrency, because of two important reasons. The first reason is that simulating concurrency gives us *total control*: Because the global state of our demo is known at all times, it could be saved and restored (not implemented yet), and experiments (potentially involving tweakable network delays) become *repeatable*. The second reason for simulating concurrency is that we wanted to focus our efforts on what is novel, rather than getting involved in the accidental complexities of implementing a network protocol and maintaining a "collaboration server". This also shows that we do not depend on any specific (e.g., client–server) network topology. Finally, the fact that our demo is fully local, allows us to use small, incrementing integers for unique node IDs,[7] which are easier for humans to remember and compare, making it easier to understand what is going on. Despite this, our architecture is designed to allow both synchronous and asynchronous collaboration, and we remain confident that we can achieve this goal.

[7] Our core library allows usage of any unique ID generation function.

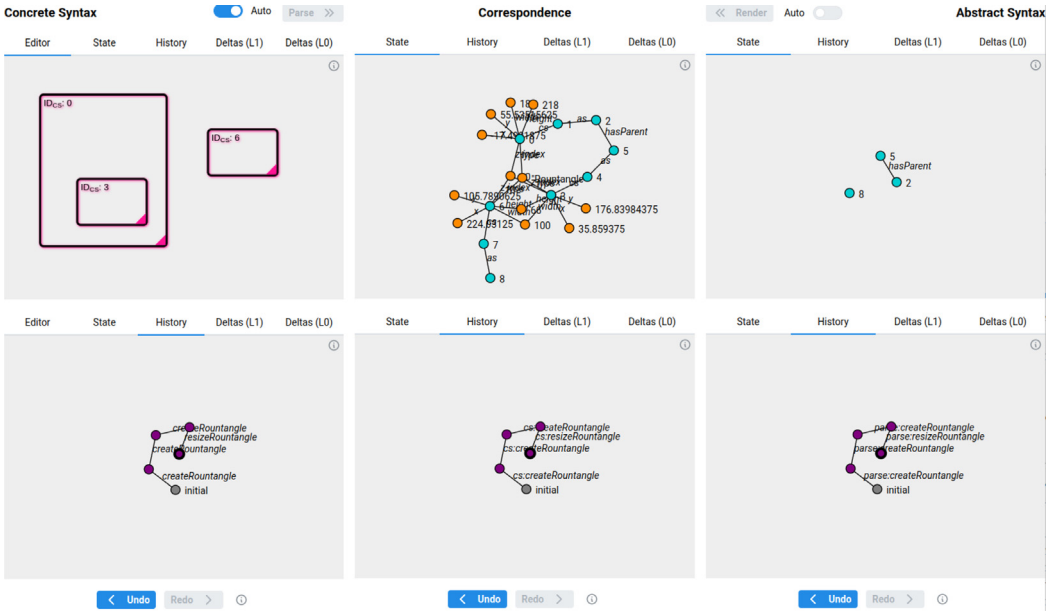

Fig. 16. Screenshot of our "correspondence" demo.

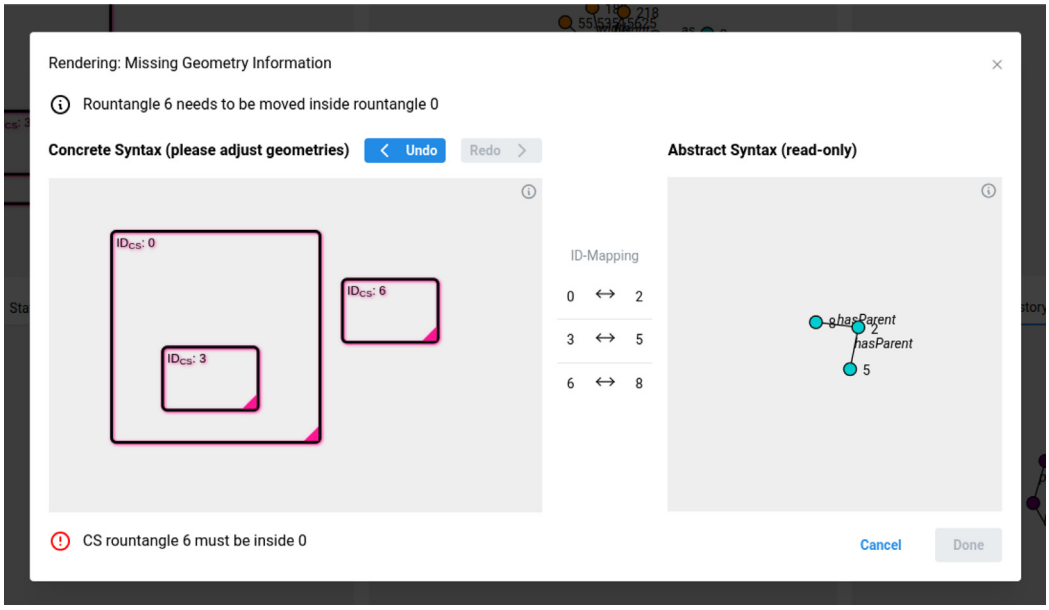

Fig. 17. Screenshot of the manual renderer ("correspondence" and "blended modeling" demos).

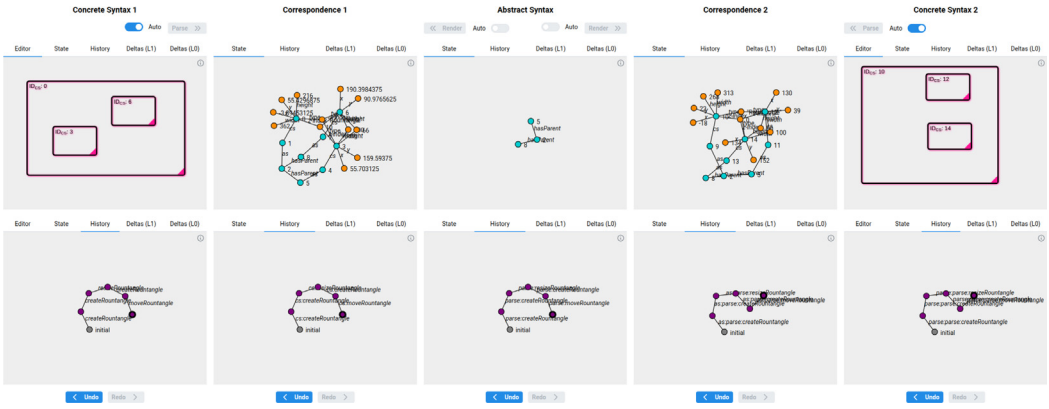

Fig. 18. Screenshot of our "blended modeling" demo.

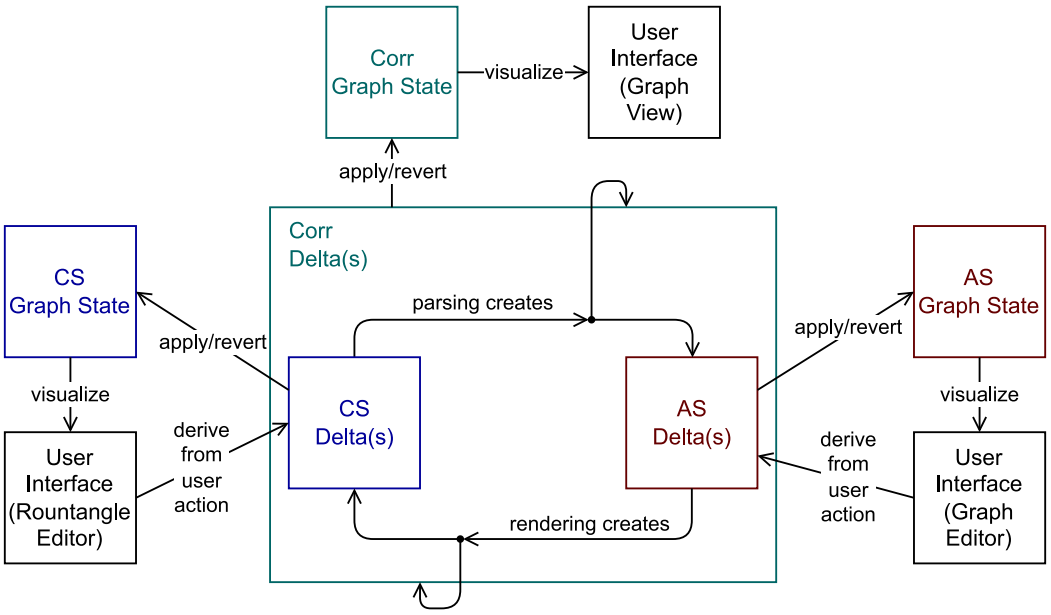

Fig. 19. Overall architecture of our approach.

The most obvious mechanism for simulating concurrency is *undo*: by undoing changes (i.e., going back to an earlier version), and subsequently making new changes, history, which is append-only, is effectively *branched*, resulting in concurrent versions, that can be merged later on. From a logical/causal viewpoint, there is no difference between concurrent versions produced by undo or by actual concurrent edit operations (performed by users unaware of each other's actions). Another mechanism for simulating concurrency is disabling auto-parsing and auto-rendering: For instance, if changes to concrete syntax are immediately parsed to abstract syntax, but changes to abstract syntax are not immediately rendered to other concrete syntaxes, then different concrete syntaxes can be edited "concurrently", causing abstract syntax to be branched (and merged, if desired).

Parser and renderer implementation. In our previous work [12], we decided that the precise implementations of parsers and renderers were out-of-scope. While it is still not the focus of our work, we needed a parser and renderer implementation for our proof-of-concept.

The current implementation of our parser and renderer is ad-hoc, code-based (about 300 lines in TypeScript). Given a set of (CS/AS) deltas, it produces new (AS/CS, resp. + Corr) deltas. During their operation, the parser and renderer work on hidden *clones* of the current *graph state* of CS, AS and Corr, that are updated in-place, while keeping track of the produced deltas. When parsing/rendering has finished, the clones are destroyed, and the produced deltas returned. This is not the most efficient, but sufficient to demonstrate our approach.

Writing our (incremental) parser and renderer in code was tedious work, and we are convinced that this is ultimately not a good solution. We will discuss alternatives in Future Work (Section 8.1).

Minor simplifications. We have made some trivial simplifications (still demonstrating the essential complexity of incremental change propagation) to the graph state representation of CS, Corr and AS, compared to what is presented in Fig. 4. We found that for our ad-hoc parser + renderer implementation, it is sufficient for the correspondence model to only keep track of which CS-Rountangle node corresponds to which AS-State node, allowing us to omit CorrParent links (every State can have at most one parent, after all). This in turn allowed us to omit the intermediary node in AS parent-links, representing them with an edge only. Finally, the parser + renderer assumes that every AS node is a State, and that every Corr node is a CorrState node, so therefore these "type"-edges have been omitted as well.

Merging implementation. While we do have a working implementation of the merge-function described in Section 4, due to timing constraints, we do not yet have a UI for merging versions, and displaying conflict resolution alternatives. However, the user can enact the scenario of our running example and observe conflicts between edits and propagated changes in the Delta (L0 or L1) view.

7. Related work

We have classified the related work into six partially overlapping sets, as shown in Fig. 20. Each set contains some of the tools/approaches/projects that will now be discussed.

Since one of our main motivations is to support collaboration in blended modeling, it is natural to look at existing tools for blended modeling. As part of the BUMBLE project [24], which "aims for a significant improvement of the current state-of-the-art modeling tools", several approaches to blended modeling have been (prototypically) implemented [25–27]. None of them provide versioning capabilities however. In [26], the authors show a kind of synchronous collaboration across different platforms but neglect all the problems that arise, such as versioning, conflict detection, and resolving inconsistencies.

David et al. reviewed commercial and open source tools with blended modeling features in [28]. Unfortunately, very little information is provided about the collaboration capabilities of these tools. For one of these tools, Umple, Badreddin et al. mention in [29] that its versioning is based on the textual syntax provided for all models in this tool. The presented merging approach is similar to SVN [2] diffing and merging.

In [30], Van Tendeloo et al. present a framework for collaborative model development that supports blended modeling. They propose a clear separation of separately versioned CS and AS, each corresponding to their own meta-model. However, Van Tendeloo's framework does not specify how versioning should be implemented, and how or when the synchronization of CS and AS should occur. Our work attempts to fill this gap.

If we broaden the view to general model versioning approaches, several publications can be found that inspired our presented approach. In [16], Kehrer et al. present an approach that extracts semantic differences from low-level state-based differences in model versions. The authors state that "logging-based [i.e., operation-based (author's note)] approaches require closed environments and do not work with

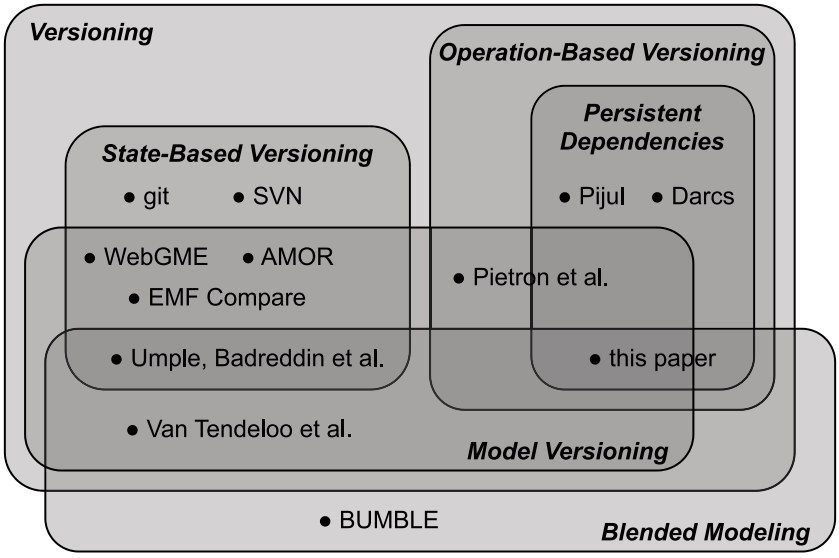

Fig. 20. Classification of related work.

independently created models". This is true for our approach as we require an editing environment that can record all user operations, and not only the resulting state. However, we motivate in Section 3.1 that this is only an issue of standardization. Interestingly, in a recent literature study on collaborative model-driven software engineering [4], it is mentioned that the "support for semantic techniques, however, has increased from 4% to 17%. Especially in approaches emphasizing collaboration across disparate domains, semantic techniques [...] are crucial".[8] This statement is in particular applicable in the context of blended modeling.

Barrett et al. draw a similar conclusion [31]. In their work on model merging, the authors suggest exposing different element matching strategies to the user. Their presented merge process allows for combining different matching strategies and is able to deal with conflicts across different matching strategies. Our proposed solution to this matching problem is the introduction of a correspondence model that keeps the matching information between the CS and AS elements. This is not a matching between different versions of the same model, but a matching between different types of models, however these two cases can be treated similarly. The authors identify the possibility of postponing merge decisions — as supported by our approach — as an advantage because "[...] the later merge decisions will have to be made, [...] the more context there will be to make them".[31]

Many of the versioning approaches implemented in different tools use or at least simulate existing generic text-based versioning tools. Only a few tool providers present their internal approach to versioning. Maroti et al. [32] present the tool WebGME, a further development of GME that supports online collaboration by versioning and automatic branching if a conflict occurs. They use hashes as identifiers for the different nodes in their graph-like data structure representing (a set of) model objects, similar to how git [1] operates. In our operation-based approach, the graph structure contains the edit operations that led to a specific model. This information allows us to handle conflicts in a more fine-grained manner because we know how the model(s) evolved. We additionally expect our approach to scale better, since no costly diffing needs to be performed.

The work of Pietron et al. [33] presents an operation-based versioning system supporting synchronous and asynchronous collaboration. Compared to the approach in this paper, their work focuses mainly on the AS and does not support multiple CSs and their synchronization. Some CS-related operations, such as updating the layout of an element, are supported but lack a clear distinction from AS-related operations.

DesignSpace [34] is a centralized, operation-based versioning system and model repository, intended to bring heterogeneous artefacts from third party tools into a single unified graph structure, via the use of tool adapters. DesignSpace supports both synchronous and asynchronous collaboration, and it has demonstrated fast incremental global consistency checking (i.e., on *state*) in large repositories [35]. Its "core" overlaps functionally with our versioning approach, but it is not mentioned how concurrent operations, merging and conflict detection are implemented.

Regarding specific research in the context of model versioning, a comprehensive overview of this topic is provided by the AMOR project [6]. Brosch et al. [36] give various (overlapping) definitions of conflicts and/or inconsistencies. Mens [37] goes into more detail on different types of inconsistencies such as syntactic, structural, and semantic conflicts. A precise formal definition of conflicts based on graph theory can be found in Taentzer et al. [38]. Their terms "state-based conflict" and "operation-based conflict" they introduce should not be confused with state-based vs. operation-based versioning.

In the work of Barrett et al. [31] the authors also provide definitions of the terms "inconsistency", "inconformity", and "conflict". The given definitions are consistent with ours. Due to the separate treatment of AS and CS in our approach, the case "inconformity" is handled by the parsing function, which always results in a conform but possibly partial AS model.

A taxonomy of conflicts together with a visualization of the different conflicts (conflict diagram) is presented by Brosch et al. in [39,40]. In this work, conflicts are either "overlapping changes" (competing changes, similar to our "conflicts") or (constraint) "violations". The basis of their approach is a tight coupling between AS and CS, but they allow extensions with language-specific features. They define graph transformation rules for conflict detection and resolution, but they only consider elements of the AS and generate only (abstract) elements of the conflict diagram. They mention to deal with layout problems arising during the rendering process of the conflict diagram by simple heuristics preserving simple layout continuity. Layout conflicts in the original model are handled by only preserving the merging user's layout. More complex layout problems are not considered (and even cannot be considered due to the restriction to the level of AS).

In our work, CS and AS are explicitly distinguished and are only loosely coupled via a correspondence model. This enables us to detect and handle CS and AS conflicts in a uniform manner, and to directly

[8] Here, the term "semantic techniques" refers to operation-based versioning.

support blended modeling, which is not the case with the results of the AMOR project. Wieland et al. [41] propose "conflict-tolerant merging of models". This is supported — by design — in our approach, including the information about "how this conflict was resolved and who was responsible for the resolution decision" [41] by persisting all (conflicting) deltas together with meta-information about the author of the delta.

Badreddin et al. also conclude that separate treatment of CS and AS is beneficial. They write in [29]: "For effective versioning and merging, separate consideration of model entities from diagram layout directives is necessary to reduce the number of potential merging conflicts".

Finally, Pijul [42] is a text-based versioning system that can be used as a drop-in replacement for git. Pijul uses diffing in order to detect "patches" (similar to deltas in our approach), but it actually stores these patches (instead of snapshots), and it even persists dependencies between them. Further, a conflict is a symmetric relation between patches. Because Pijul only supports operations on text and filesystem directories, dependencies at the abstract syntax level (e.g., a module importing another module) cannot be detected. As a remedy, Pijul supports manually adding dependencies at the file level. Pijul improves its predecessor Darcs [43] by a conflict-tolerant data structure that allows to keep conflicts until they are merged manually. This is comparable to the conflicting deltas (see Section 3.3). In contrast, our approach persists the (evolving) AS tree/graph, and a module import is represented by an edge to that module's node, automatically resulting in a dependency.

8. Conclusion

We presented an approach for optimistic versioning of CS, AS, and a fine-grained arbitrary correspondences between them, in an operation-based manner. We start from primitive graph operations and exhaustively describe the types of dependencies and conflicts that can occur between them. By composing these primitive operations into transactions, we can treat corresponding changes to CS and AS as atomic, and merge, detect and resolve conflicts at this level. CS and AS model versions are simply sets of deltas. Correspondence model versions are also sets of deltas, but at a higher level. Parsing and rendering is simply the act of producing new deltas and new model versions. As a consequence, parsing and rendering are non-blocking operations, which is especially useful if one wants to support postponed (semi-) manual rendering, which we think is vital in blended modeling.

Although the building blocks of our approach (i.e., correspondence models, operation-based versioning, dependencies and conflicts between operations, transactions of operations) are not novel, our proposed combination is.

8.1. Limitations and future work

We see several directions for this research to continue which we want to summarize in this following.

We consider any kind of bias in conflict resolution (e.g., preferring certain types of operations, or preferring certain users) to be *external* from an architectural point of view, and we think that it should be based on empirical research. Our solution should lend itself well to empirical research, because we follow a "just record everything" philosophy. In the long term, a biased conflict resolution suggestion mechanism could become self-adapting, based on historical data.

The creation of the correspondence meta-model, and writing the parser/renderer in a generic programming language (like we did for our demo) is tedious work. It is therefore worth exploring their automated generation from specifications in more appropriate modeling languages. We can think of some examples:

- The CS ↔ AS mappings of many visual topological DSLs are trivial (CS = AS + icons + layout). In these cases, they could be generated from a very simple specification.
- For textual CS ↔ AS mappings, a grammar-like specification may be more natural.
- For more complex CS ↔ AS mappings, declarative descriptions (e.g., based on TGGs) already exist and could be used to generate operational CS ↔ AS mappings compatible with our approach.

Further, it would be an interesting challenge to extend such specifications to support also human input (like the renderer in our demo), to deal with non-determinism.

Related to this is the *reuse* of parser/renderer logic for certain CS features. For instance, geometric "insideness" is a CS feature that is shared by many visual languages, such as Statecharts, Venn Diagrams, ROOM [44], DEVS and SysML. We can enable reuse by having more than one correspondence layer between CS and AS, as demonstrated by CouchEdit [11]. By introducing a "CS + insideness" layer between CS and AS, where geometric "insideness" is resolved to abstract "insideness" links, the concrete syntax "insideness" feature could be *reused* across languages. We would then have two correspondence models: one between CS and CS + insideness, and one between CS + insideness and AS. The parser and renderer rules for the former (CS ↔ CS + insideness) would be reusable, and the parser rules for the latter (CS + insideness ↔ AS) would be trivial (one-to-one mapping).

We could also extend our work to bi-directional synchronization in general. In our running example, every CS instance always contained at least as much information as its corresponding AS instance, but this is not the case for bi-directional synchronization in general. In the most complex case, there is only *partial overlap* between the information contained by either instance. This would lead to interesting problems.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article

Acknowledgments

Author J. Exelmans is an SB Ph.D. fellow at FWO (1S70622N). Author J. Pietron is partly funded by the project *GENIAL!*, which is partly funded by the German Federal Ministry of Education and Research (BMBF) within the research programme ICT 2020 (reference number: 16ES0875).

References

[1] git Version Control System, 2022, URL https://git-scm.com/. (Accessed 12 August 2022).

[2] Apache Subversion, 2022, URL https://subversion.apache.org/. (Accessed 12 August 2022).

[3] Mirco Franzago, Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Collaborative model-driven software engineering: A classification framework and a research map, IEEE Trans. Softw. Eng. 44 (12) (2018) 1146–1175, http://dx.doi.org/10.1109/TSE.2017.2755039.

[4] Istvan David, Kousar Aslam, Sogol Faridmoayer, Ivano Malavolta, Eugene Syriani, Patricia Lago, Collaborative Model-Driven Software Engineering: A Systematic Update, in: 2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems, MODELS, IEEE, 2021, pp. 273–284, http://dx.doi.org/10.1109/MODELS50736.2021.00035.

[5] EMF Compare, 2022, URL https://www.eclipse.org/emf/compare/. (Accessed 12 August 2022).

[6] AMOR — Adaptable Model Versioning, 2009, URL http://modelversioning.org/. (Accessed 12 August 2022).

[7] Federico Ciccozzi, Matthias Tichy, Hans Vangheluwe, Danny Weyns, Blended modelling - what, why and how, in: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C, IEEE, 2019, pp. 425–430, http://dx.doi.org/10.1109/MODELS-C.2019.00068.

[8] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, Janet Siegmund, Efficiency of projectional editing: A controlled experiment, in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE, ACM, New York, NY, USA, 2016, pp. 763–774, http://dx.doi.org/10.1145/2950290.2950315.

[9] Bashar Nuseibeh, Steve Easterbrook, Alessandra Russo, Making inconsistency respectable in software development, J. Syst. Softw. 58 (2) (2001) 171–180, http://dx.doi.org/10.1016/S0164-1212(01)00036-X.

[10] Esther Guerra, Juan de Lara, On the quest for flexible modelling, in: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS '18, ACM, New York, NY, USA, 2018, pp. 23–33, http://dx.doi.org/10.1145/3239372.3239376.

[11] Leander Nachreiner, Alexander Raschke, Michael Stegmaier, Matthias Tichy, CouchEdit: A relaxed conformance editing approach, in: Esther Guerra, Ludovico Iovino (Eds.), ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings, MODELS '20, ACM, 2020, pp. 43:1–43:5, http://dx.doi.org/10.1145/3417990.3421401.

[12] Joeri Exelmans, Jakob Pietron, Alexander Raschke, Hans Vangheluwe, Matthias Tichy, Optimistic versioning for conflict-tolerant collaborative blended modeling, in: Catherine Dubois, Julien Cohen (Eds.), STAF 2022 Workshop Proceedings: 10th International Workshop on Bidirectional Transformations (BX 2022), 2nd International Workshop on Foundations and Practice of Visual Modeling (FPVM 2022) and 2nd International Workshop on MDE for Smart IoT Systems, MeSS 2022 (Co-Located with Software Technologies: Applications and Foundations Federation of Conferences (STAF 2022)), Nantes, France, July 5-8, 2022, in: CEUR Workshop Proceedings, vol. 3250, CEUR-WS.org, 2022, URL http://ceur-ws.org/Vol-3250/fpvmpaper1.pdf.

[13] David Harel, Statecharts: A visual formalism for complex systems, Sci. Comput. Program. 8 (3) (1987) 231–274, http://dx.doi.org/10.1016/0167-6423(87)90035-9.

[14] David Harel, On visual formalisms, Commun. ACM 31 (5) (1988) 514–530, http://dx.doi.org/10.1145/42411.42414.

[15] Ernst Lippe, Norbert van Oosterom, Operation-based merging, SIGSOFT Softw. Eng. Notes 17 (5) (1992) 78–87, http://dx.doi.org/10.1145/142882.143753.

[16] Timo Kehrer, Udo Kelter, Gabriele Taentzer, A rule-based approach to the semantic lifting of model differences in the context of model versioning, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, 2011, pp. 163–172, http://dx.doi.org/10.1109/ASE.2011.6100050.

[17] Maximilian Koegel, Jonas Helming, Stephan Seyboth, Operation-based conflict detection and resolution, in: 2009 ICSE Workshop on Comparison and Versioning of Software Models, 2009, pp. 43–48, http://dx.doi.org/10.1109/CVSM.2009.5071721.

[18] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, Jordi Cabot, Towards a language server protocol infrastructure for graphical modeling, in: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 370–380, http://dx.doi.org/10.1145/3239372.3239383.

[19] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, Maria Zimakova, Modelling and analysis using GROOVE, Int. J. Softw. Tools Technol. Transf. 14 (1) (2012) 15–40, http://dx.doi.org/10.1007/s10009-011-0186-x.

[20] Jean Bézivin, Model driven engineering: An emerging technical space, in: Ralf Lämmel, João Saraiva, Joost Visser (Eds.), Generative and Transformational Techniques in Software Engineering, International Summer School, Braga, Portugal, July 4-8, 2005. Revised Papers, GTTSE 2005, in: Lecture Notes in Computer Science, vol. 4143, Springer, 2005, pp. 36–64, http://dx.doi.org/10.1007/11877028_2.

[21] Glynn Winskel, An introduction to event structures, in: J.W. de Bakker, Willem P. de Roever, Grzegorz Rozenberg (Eds.), Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, the Netherlands, May 30 - June 3, 1988, Proceedings, in: Lecture Notes in Computer Science, vol. 354, Springer, 1988, pp. 364–397, http://dx.doi.org/10.1007/BFb0013026.

[22] Holger Giese, Robert Wagner, Incremental Model Synchronization with Triple Graph Grammars, vol. 4199, 2006, pp. 543–557, http://dx.doi.org/10.1007/11880240_38.

[23] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, Manuel Wimmer, Explicit transformation modeling, in: Sudipto Ghosh (Ed.), Models in Software Engineering, Workshops and Symposia At MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 6002, Springer, 2009, pp. 240–255, http://dx.doi.org/10.1007/978-3-642-12261-3_23.

[24] BUMBLE — Blended modelling for enhanced software and systems engineering, 2019, URL https://blended-modeling.github.io/. (Accessed 12 August 2022).

[25] Malvina Latifaj, Federico Ciccozzi, Mattias Mohlin, Ernesto Posse, Towards automated support for blended modelling of UML-RT embedded software architectures, in: Companion Proceedings of the 15th European Conference on Software Architecture, ECSA-C 2021, in: CEUR Workshop Proceedings, vol. 2978, 2021, URL http://ceur-ws.org/Vol-2978/industry-paper90.pdf.

[26] Malvina Latifaj, Hilal Taha, Federico Ciccozzi, Antonio Cicchetti, Cross-platform blended modelling with JetBrains MPS and eclipse modeling framework, in: Shahram Latifi (Ed.), 19th International Conference on Information Technology-New Generations, ITNG 2022, Springer International Publishing, Cham, 2022, pp. 3–10, http://dx.doi.org/10.1007/978-3-030-97652-1_1.

[27] Lorenzo Addazi, Federico Ciccozzi, Blended graphical and textual modelling for UML profiles: A proof-of-concept implementation and experiment, J. Syst. Softw. 175 (2021) 110912, http://dx.doi.org/10.1016/j.jss.2021.110912.

[28] Istvan David, Malvina Latifaj, Jakob Pietron, Weixing Zhang, Federico Ciccozzi, Ivano Malavolta, Alexander Raschke, Jan-Philipp Steghöfer, Regina Hebig, Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study, Softw. Syst. Model. (2022) http://dx.doi.org/10.1007/s10270-022-01010-3.

[29] Omar Bahy Badreddin, Timothy C. Lethbridge, Andrew Forward, A novel approach to versioning and merging model and code uniformly, in: Luís Ferreira Pires, Slimane Hammoudi, Joaquim Filipe, Rui César das Neves (Eds.), Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014, MODELSWARD 2014, SciTePress, 2014, pp. 254–263, http://dx.doi.org/10.5220/0004699802540263.

[30] Yentl Van Tendeloo, Hans Vangheluwe, Unifying model- and screen sharing, in: IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE, 2018, pp. 127–132, http://dx.doi.org/10.1109/WETICE.2018.00031.

[31] Stephen C. Barrett, Greg Butler, Patrice Chalin, Mirador: A synthesis of model matching strategies, in: Proceedings of the 1st International Workshop on Model Comparison in Practice, Association for Computing Machinery, New York, NY, USA, 2010, pp. 2–10, http://dx.doi.org/10.1145/1826147.1826151.

[32] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamer Levendovszky, Ákos Lédeczi, Next generation (meta)modeling: Web- and cloud-based collaborative tool infrastructure, in: Daniel Balasubramanian, Christophe Jacquet, Pieter Van Gorp, Sahar Kokaly, Tamás Mészáros (Eds.), Proceedings of the 8th Workshop on Multi-Paradigm Modeling, in: CEUR Workshop Proceedings, vol. 1237, CEUR-WS.org, 2014, pp. 41–60, URL http://ceur-ws.org/Vol-1237/paper5.pdf.

[33] Jakob Pietron, Fabian Füg, Matthias Tichy, An operation-based versioning approach for synchronous and asynchronous collaboration in graphical modeling tools, in: Ludovico Iovino, Lars Michael Kristensen (Eds.), 2021 Workshop Proceedings, STAF, in: CEUR Workshop Proceedings, vol. 2999, CEUR-WS.org, 2021, pp. 88–89, URL http://ceur-ws.org/Vol-2999/fpvmdata4mdepaper3.pdf.

[34] Andreas Demuth, Markus Riedl-Ehrenleitner, Alexander Nöhrer, Peter Hehenberger, Klaus Zeman, Alexander Egyed, Designspace: an Infrastructure for Multi-user/Multi-tool Engineering, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015, ACM, 2015, pp. 1486–1491,.

[35] Michael Alexander Tröls, Luciano Marchezan, Atif Mashkoor, Alexander Egyed, Instant and global consistency checking during collaborative engineering, Softw. Syst. Model. (2022) 21 (6) 2489–2515, http://dx.doi.org/10.1007/s10270-022-00984-4.

[36] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, An introduction to model versioning, in: Marco Bernardo, Vittorio Cortellessa, Alfonso Pierantonio (Eds.), in: Formal Methods for Model-Driven Engineering, vol. LNCS 7320, Springer, Berlin, Heidelberg, 2012, pp. 336–398, http://dx.doi.org/10.1007/978-3-642-30982-3_10.

[37] T. Mens, A state-of-the-art survey on software merging, IEEE Trans. Softw. Eng. 28 (5) (2002) 449–462, http://dx.doi.org/10.1109/TSE.2002.1000449.

[38] Gabriele Taentzer, Claudia Ermel, Philip Langer, Manuel Wimmer, Conflict detection for model versioning based on graph modifications, in: Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, Andy Schürr (Eds.), Graph Transformations - 5th International Conference, Enschede, the Netherlands, September 27–October 2, 2010. Proceedings, ICGT 2010, in: Lecture Notes in Computer Science,

vol. 6372, Springer, 2010, pp. 171–186, http://dx.doi.org/10.1007/978-3-642-15928-2_12.

[39] Petra Brosch, Martina Seidl, Manuel Wimmer, Gerti Kappel, Conflict visualization for evolving UML models, J. Object Technol. 11 (3) (2012) 2: 1–30, http://dx.doi.org/10.5381/jot.2012.11.3.a2.

[40] Petra Brosch, Conflict Resolution in Model Versioning (Ph.D. dissertation), Vienna University of Technology, Vienna, 2012, URL https://publik.tuwien.ac.at/files/PubDat_208975.pdf.

[41] Konrad Wieland, Philip Langer, Martina Seidl, Manuel Wimmer, Gerti Kappel, Turning Conflicts into Collaboration, Comput. Support. Coop. Work (CSCW) 22 (2–3) (2013) 181–240, http://dx.doi.org/10.1007/s10606-012-9172-4.

[42] Pijul distributed version control system, 2016-2022, URL https://pijul.org/. (Accessed 18 August 2022).

[43] David Roundy, Darcs: Distributed version management in haskell, in: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell, Haskell '05, Association for Computing Machinery, New York, NY, USA, 2005, pp. 1–4, http://dx.doi.org/10.1145/1088348.1088349.

[44] Bran Selic, Real-time object-oriented modeling, IFAC Proc. Vol. 29 (5) (1996) 1–6, http://dx.doi.org/10.1016/S1474-6670(17)46346-4.