Optimistic Versioning for Conflict-tolerant Collaborative Blended Modeling

Joeri Exelmans[1], Jakob Pietron[2], Alexander Raschke[2], Hans Vangheluwe[1] and Matthias Tichy[2]

[1] *Department of Computer Science, University of Antwerp – Flanders Make, Antwerp, Belgium*
[2] *Institute of Software Engineering and Programming Languages, Ulm University, Ulm, Germany*

Abstract

Optimistic versioning is a key component in supporting collaborative workflows. Text-based versioning has been widely adopted for versioning code, but in model-driven engineering, dealing with visual concrete syntaxes, new methods are required. In the case of blended modeling, a mixture of both textual and visual syntaxes, concurrently editable and synchronizable, introduces additional challenges.

We propose a type of operation-based versioning to record not only user edits, but also bi-directional change propagations between concrete and abstract syntax. This way we can support blended modeling with layout continuity, and flexible handling of missing information (e. g., layout information) when rendering changes from abstract to concrete syntax. In addition, the proposed versioning approach enables collaborative conflict resolution by allowing partial conflict resolution, thus deferring a final resolution.

Keywords

versioning, blended modeling, conflict-tolerant, operation-based

1. Introduction

Model-driven engineering (MDE) has become widely accepted as prime enabler for the creation of increasingly complex software-intensive systems. In addition to graphical models, various model representations such as tabular or textual ones are typically used. The flexible use of different representations (concrete syntax (CS)) for one and the same model (abstract syntax (AS)) is also called *blended modeling*. The ability to switch between different representations allows the user to choose the one that is most useful and efficient for the current task [1].

Complex systems are developed in teams in both asynchronous and synchronous collaborative environments [2]. Synchronous collaboration has gained importance since the Corona pandemic, where developers who normally develop models in the same room, e. g. on whiteboards, were forced to use (online) tools for collaboration [3].

In addition to the complexity of concurrency in (a)synchronous collaboration (i.e. branching, merging, and dealing with conflicts), an orthogonal problem of blended modeling is the challenge of (multi-)CS and AS synchronization. For instance, there may be concurrency on the same

FPVM 2022: 2nd International Workshop on Foundations and Practice of Visual Modeling, July 4–8, 2022, Nantes, France
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)
0000-0002-6916-5140 (J. Exelmans); 0000-0001-8308-6636 (J. Pietron); 0000-0002-6088-8393 (A. Raschke); 0000-0003-2079-6643 (H. Vangheluwe); 0000-0002-9067-3748 (M. Tichy)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3250/fpvmpaper1.pdf

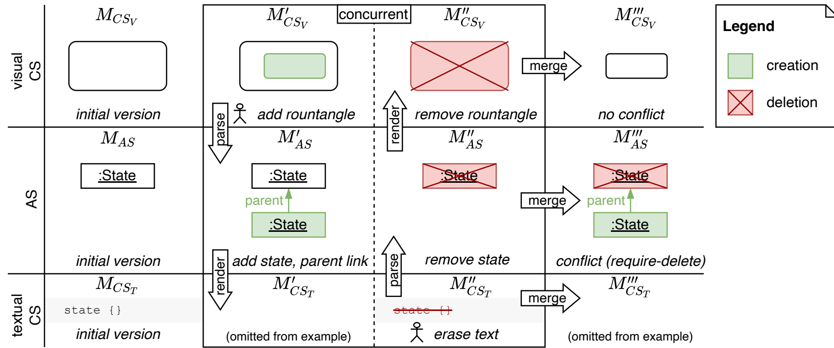

Figure 1: Running example: Blended modeling scenario with concurrent user edits

or on different CSs that are being synchronized (see Figure 1 and Section 2). Another issue of blended modeling is that of missing information when switching between CS representations. For example, if an element is added in textual syntax, the position of the new corresponding element in a graphical syntax remains unknown.

Versioning systems for "code" (e.g. git [4], SVN [5]) are only suited for recording textual CS. Model versioning systems attempt to overcome this limitation by recording, comparing and merging instead at the level of the AS. Only a single (visual) CS is assumed to exist, with a 1:1 mapping between CS and AS elements. This hinders support for blended modeling.

To overcome this limitation, we propose to combine operation-based versioning with incremental bi-directional change propagation to enable (a) arbitrary mappings between CS and AS, with traceability between CS and AS, (b) blended modeling with layout continuity and flexible handling of missing (layout) information, and (c) reuse of CS editing environments for different languages. Additionally, as part of our operation-based versioning approach, we propose a new way to persist merge conflicts, to allow recording of the steps taken in conflict resolution.

Our approach should not be confused with projectional editing [6], where CS operations directly impact the AS without using a parser. On the contrary, we allow arbitrary CS/AS mappings, increasing the flexibility for the modeler and ultimately the usability of the modeling environment [7, 8].

The remainder of the paper is structured as follows. In Section 2, we introduce a running example that is used to illustrate our proposed solution presented in Section 3. Section 4 discusses related work and Section 5 concludes with an outlook on future work.

2. Running Example

To illustrate our approach, we introduce a (visual) CS and AS for a very limited subset of the Statecharts formalism. On the CS side, we have 2D drawings of rountangles (rounded rectangles) with geometries. On the AS side, we can have State objects, between which a "parent" association exists (every State can have 1 parent). The correspondence relation between CS and AS is as follows: There exists a one-to-one mapping between rountangles and State objects, and whenever a rountangle is geometrically inside another, a parent link between their

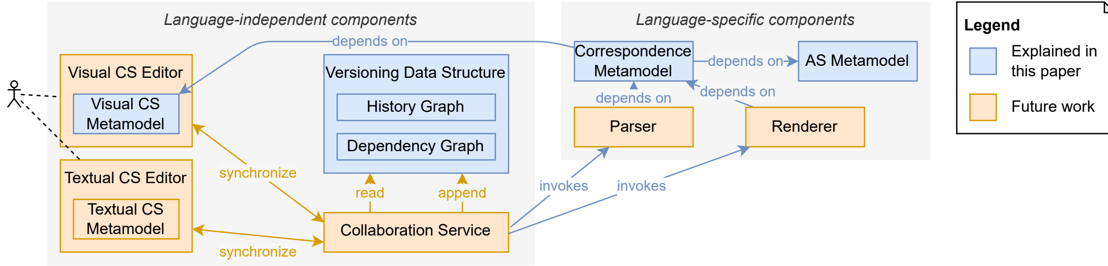

Figure 2: “Big picture”: A possible architecture

corresponding State objects must exist. Additionally, we assume that there is a textual CS, that we do not describe here as it does not add additional insight.

Figure 1 shows evolving CS and AS models. Initially, we have a version M_{CS_V} of a visual CS model with a single rountangle, M_{AS} of the corresponding AS model with a single State, and M_{CS_T} the corresponding textual CS model. Then, concurrently, a change happens to the visual and textual CS models. In the visual model, an inner rountangle is added (producing M'_{CS_V}) and, concurrently, in the textual model, some text is erased (producing M''_{CS_T}). Both changes could be propagated (parsed) to the AS, and subsequently rendered to the other CS. How do we represent these changes, and how do we merge them? We intuitively understand that there will be a merge conflict, at least at the level of the AS: a State object is being deleted, while at the same time, it is the target of a newly created parent link.

There may be many meaningful ways to resolve such a conflict, but this is not our focus here. In this paper the main focus is on recording (concurrent) changes and (concurrent) change propagations, on and between CS and AS.

Note that in the remainder of this paper, we assume that text removal in M''_{CS_T} has already been parsed to produce M''_{AS}. This way, we can focus on the visual CS and the AS, which are sufficient to explain bi-directional change propagation (the main building block of blended modeling), combined with concurrency.

3. Solution

We introduce a set of components that can become part of a collaboration architecture. Their role in a possible architecture is shown in Figure 2.

We explicitly distinguish between language-specific and language-independent components. In our approach, CS and AS are separate, evolving models, each conforming to their own metamodel, as proposed by Van Tendeloo [9]. AS metamodels will always be language-specific, e.g. specific to Statecharts. CS metamodels (and their editors) will often be language-independent. For instance, a metamodel for vector graphics drawings may serve as a CS meta model for both Statecharts and Petri Nets. Users only interact directly with a model through a CS. Obviously, the definition of a mapping between CS and AS will be language-specific. In our approach, this mapping consists of a *correspondence metamodel*, and a *parser* and *renderer* function. We will explain these concepts, and specify an interface for parser and renderer functions.

The language-independent component *collaboration service* is left underspecified. Among its

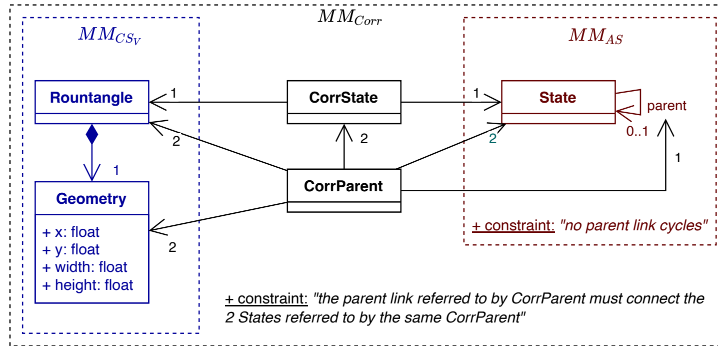

Figure 3: Running example: Concrete syntax, abstract syntax and correspondence metamodels

many tasks is synchronization of CS editors with evolving CS models, (networked) synchronization between collaborators, and synchronization between CS and AS models. For the latter, it invokes the parser and renderer functions. This service uses the *versioning data structure* to record the edit history of models. We will explain this component, and why it is especially well suited for blended modeling.

The components presented in this paper are deployment-independent. For instance, the versioning data structure could be deployed centrally or decentrally.

3.1. Incremental parsing and rendering

A change to a CS may cause corresponding changes to the AS. Change propagation from CS to AS is called *parsing*. Subsequently, this change to the AS may cause changes to other CSs. Change propagation from AS to CS is called *rendering*[1].

Parsing and rendering must happen incrementally for several reasons: The first reason is performance (not having to parse/render from scratch after every change). The second reason is *layout continuity* when rendering a visual update (not regenerating a new layout from scratch, which would be confusing to users familiar with an existing layout). The third reason is that in a multi-user collaboration scenario, propagating concurrent changes must cause new concurrent changes (instead of new concurrent models), that can be merged accurately and efficiently.

3.2. Correspondence model

The user only interacts with a model through a CS, so any information (e. g., inconsistencies) from the AS or semantic level must also be visualized in the CS. CS and AS can relate to each other in nontrivial ways, and inferring this information a posteriori would be complex, and non-deterministic (more than one solution). We therefore persist traceability information between CS and AS elements, and keep it up-to-date, every time changes are propagated.

We persist traceability information between one CS and one AS model in a *correspondence model*, an idea taken from triple graph grammars (TGGs) [10]. For any pair of CS and AS metamodels that can be synchronized, a correspondence metamodel must be defined. This

[1]Van Tendeloo uses the terms "comprehension" and "perceptualization" instead of "parsing" and "rendering", resp. We find "parsing" and "rendering" more intuitive.

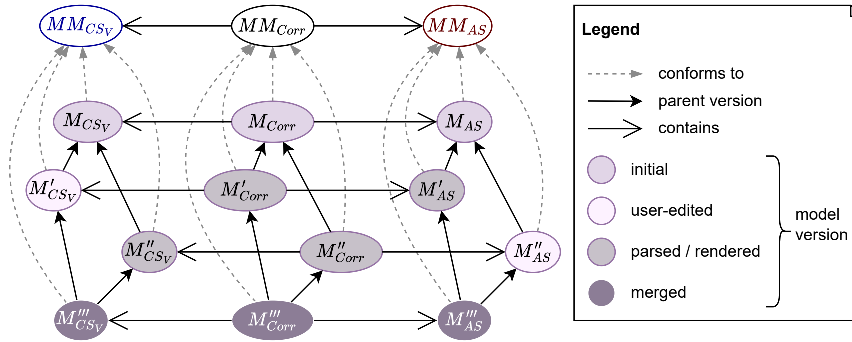

Figure 4: Running example: Edit history of CS, AS and Correspondence models

metamodel contains all types of the CS and AS metamodels that it relates to, and also the different correspondence object types, that associate CS to AS types. At the instance level, a correspondence model includes one CS and one AS model (it acts as an "overlay" on a (CS, AS) model pair), and a set of correspondence objects between their elements.

Running example: Figure 3 shows the metamodels (MM) for "the AS of the CS" (MM_{CS_V}, in blue), the AS (MM_{AS}, in red) and the correspondence (MM_{Corr}). The correspondence MM includes the elements of the CS and AS MMs, and defines two correspondence object types (*CorrState* and *CorrParent*). *CorrState* relates one CS rountangle to one AS State, and *CorrParent* relates two CS rountangles with geometrical enclosure to a parent link between their corresponding AS States.

Note that although we adopt some ideas from TGGs, we do not require change propagation rules to be specified as TGGs - our approach is neutral with respect to rule specification, and rules may even be specified imperatively.

3.3. Persistence of (propagated) changes

Instances of CS, AS, and correspondence models evolve. We rely on an underlying versioning system (explained in Section 3.4) to persist every change in each of these models (operation-based versioning [11]). A change may be the effect of an edit operation, or a propagated change (ultimately traceable to an edit operation). Changes can be replayed, so in essence, after every change, a permanent and immutable version is created.

Running example: Figure 4 shows the three metamodels from Figure 3 at the top. Below, we see a history graph of the model versions from Figure 1, and their conformance to the meta-level. We saw that M'_{CS_V} and M''_{AS} were produced by concurrent user edits. In our history graph, we simply record these model versions, and the relation to their parent version (M_{CS_V} and M_{AS}, respectively). Parsing and rendering only produces new model versions. Parsing M'_{CS_V} produces M'_{AS} and M'_{Corr}, and rendering M''_{AS} produces M''_{CS_V} and M''_{Corr}. All pairs (M'_x, M''_x) are concurrent, and when merged, produce model M'''_x (with $x \in \{CS_V, AS, Corr\}$).

We will now explain our versioning approach in more detail. We will see what precisely makes up a model version, how we persist changes, how we relate changes to CS and AS to each other, and how we detect conflicts.

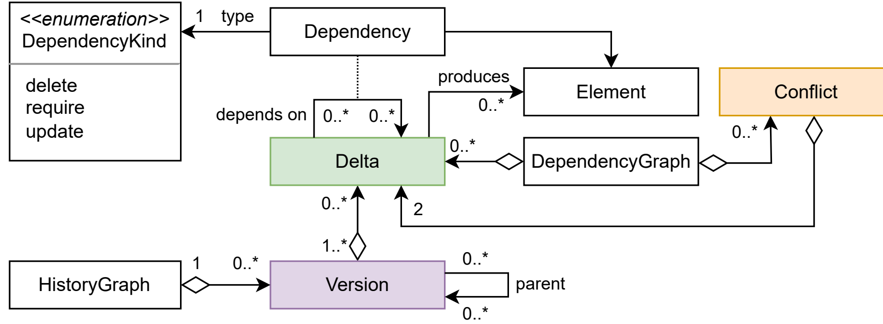

Figure 5: Metamodel of the versioning data structure

3.4. Versioning Data Structure

In this section, we present a new and application-agnostic versioning approach. Our approach can be classified as *operation-based* according to Brosch et al. [11], meaning that we persist changes between versions, instead of persisting snapshots of versions. Snapshots can be reconstructed by *replaying* changes. By persisting changes, we do not have to perform *diffing*, a complex and error-prone process. Persisting changes also enables persisting traceability information between propagated changes and user edits, which is crucial in order to support incremental parsing and rendering.

Our versioning approach consists of two data structures, called *History Graph* and *Delta Graph*. We will first explain the *Delta Graph*, which consists of deltas and dependency links between them.

Deltas. A delta (colored in green) records a change to a model. A delta may be caused by a user edit, or a propagated change (model synchronization). We only record the *effect* of the change (i. e., atomic CRUD operations on model elements), as opposed to the *cause* (e.g. the edit command). Rcording the cause as well may have advantages for edit history comprehensibility and analysis [12, 13], but this is not our focus. Deltas are transactional, meaning that they are either fully applied to the model, or not at all.

Dependencies. Deltas can depend on the effect of other deltas. We persist these *dependencies* between deltas in a directed, acyclic and append-only *Dependency Graph* in order to efficiently detect conflicts between deltas, as we will discuss later. Dependencies can be of three different types: *update*, *require*, and *delete*.

Figure 6 shows the *Dependency Graphs* for CS, AS, and correspondence of our running example. Delta cs_1 results in the creation of the outer rountangle. Further, there is delta cs_2, that creates the inner rountangle. The two deltas do not depend on each other and can, therefore, be applied to the model in any order, producing M'_{CS_V}. However, delta cs_3 deletes the outer rountangle, and in consequence, cs_3 has a *delete* dependency on cs_1. If there were a delta cs_x (not part of our running example - only for illustrative purposes) that resizes the outer rountangle, it would have an *update* dependency on cs_1. The third type, *require*, occurs when a delta requires the existence of an element created by an other delta without updating the required element itself, e.g., when connecting an edge to an element, as in as_2.

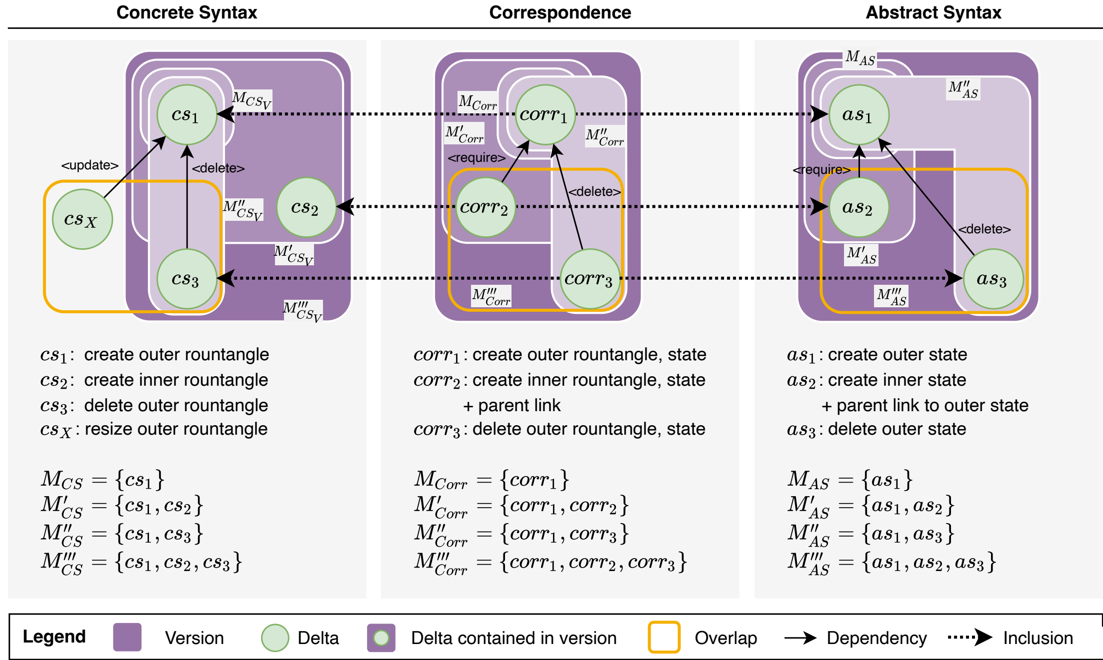

Figure 6: Running example: The three dependency graphs of CS, $Corr$, and AS consisting of deltas. Their effect on the model is listed below. Additionally, the purple layers represent the different versions.

Conflicts. Two deltas that share a dependency on the same model element are *conflicting* if at least one of the deltas alters that model element. *Conflict* types are *update-update*, *update-delete*, and *require-delete* [11].

Continuing our running example, deltas cs_1 and cs_2 are independent and not conflicting, because they do not depend on a common delta. In consequence, they can both be part of the model state without any *Conflict* as it is the case in M'_{CS_V}. However, cs_x (update geometry of outer rountangle) and cs_3 (delete outer rountangle) depend concurrently on the same model element cs_1 and are, consequently, in an *update-delete* conflict as indicated in orange color in Figure 6 on the left.

While the *Dependency Graph* imposes a partial order on the execution of deltas to obtain a valid state, it does not contain any information about the actual order in which deltas were added (or omitted, explained later) by different collaborators, altering the model state. Moreover, we want to be able to point to specific model *versions*. We therefore introduce the *History Graph*, which contains versions and their (partial) order.

Versions. A version (colored in various shades of purple), is a (possibly empty) set of deltas, that when replayed, produces a model state. In a version, all dependencies of all contained deltas must also be contained, a property called *left-closedness* in the theory of Event Structures [14]. The order of versions is persisted in a *History Graph*, which is also directed, acyclic and append-only. Versions refer to their predecessor(s) by *parent* links, whose semantics are identical to parent links in git.

Figure 4 of our running example already showed a *History Graph*, with branching into

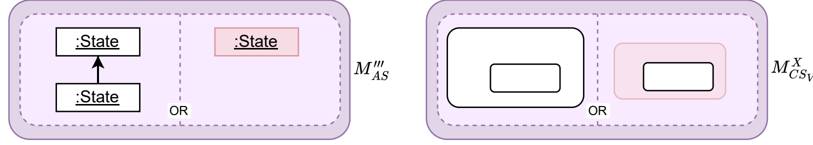

Figure 7: Running example: Instances of versions and M'''_{AS} and $M^X_{CS_V}$ in a superposition each.

concurrent versions (M'_x and M''_x) and merging (M'''_x). Figure 6 visualizes the different versions as sets of deltas.

The *difference* between any two versions A and B is simply a set of *added* deltas ($B \setminus A$) and a set of *removed* deltas ($A \setminus B$). Versions can be trivially merged by taking the union of their deltas, which preserves left-closedness.

Superpositions. Versions are allowed to contain conflicting deltas. This way, conflicting states are not just temporary, in-memory phenomena during the merge process, but persistent, which has two advantages: (1) We can record not just the occurence, but also steps taken in the resolution of the conflict (as a sequence of versions). This may be valueable information. (2) Conflicts are non-blocking, and the modeler can continue working on non-conflicting parts of the model.

We use the term *superposition* for versions containing conflict(s) (reference to quantum physics), because the version can be interpreted as containing all possible conflict resolutions. A conflict (= a pair of deltas) is resolved by excluding from the next version at least one of the conflicting deltas (and its dependants). Possibly a new delta is introduced, replacing both of the conflicting deltas.

In Figure 6, the version M'''_{AS} is in a superposition, because it contains deltas as_2 and as_3 in a *require-delete* conflict. The possible (non-conflicting) versions contained in the superposition are $\{as_1\} \times \{as_2, as_3, as_{n_1}, as_{n_2}, ...\}$, where as_{n_i} is a new (e.g. manually added) delta that replaces both conflicting deltas.

Conflicts in versions can be visualized by presenting the effects of non-conflicting deltas in a side-by-side view. Figure 7 shows M''_{AS} and $M^X_{CS_V}$ in their the superposition.

3.5. Parsing and rendering interface

We now define an interface for the parsing and rendering behavior. Parsing and rendering does not alter models in-place, and their only effect is that they produce new model versions. We specify their interfaces as *pure functions* (i.e. functions without side-effects, that only have read access to their inputs) that return the newly produced model versions. Purity keeps us honest, forcing us to be explicit about all inputs, and guaranteeing that results are repeatable, which has benefits in distributed environments: For instance, in live collaboration on the same CS model, all users could parse the model locally (to reduce latency), while guaranteeing that their results are identical.

In order to parse incrementally, the parsing and rendering functions need access to the most recent correspondence model, and to the respective CS or AS changes. As output, a new correspondence model is produced, which includes a new AS or CS model, respectively.

Running example: In our example, the inputs and outputs are as follows:

$$parse(M_{Corr}, d(M_{CS_V}, M'_{CS_V})) = (M'_{Corr}, M'_{AS})$$
$$render(M_{Corr}, d(M_{AS}, M''_{AS}), *) = (M''_{Corr}, M''_{CS_V})$$

where d is the difference (i.e. added, removed elements) between two model versions, which we can easily compute from our deltas. In the rendering function, the asterisk $*$ denotes the possibility of having additional parameters, which we will motivate in the next section.

The types of parsing errors that can occur are specific to both CS and AS, so we feel that they should be defined on the level of the correspondence metamodel. When a parsing error occurs, an object describing the error should be created in the returned correspondence model.

3.6. Dealing with missing information

Rendering is usually non-deterministic in the sense that an AS model can be mapped correctly onto many CS models, due to missing information (e.g. about layout). An automated rendering *function* can however only produce a single result, which may not always match the user's intention (which ultimately remains unknown to the computer). Therefore, we believe we must support human interaction in rendering. We see several complementary ways to support this.

First, the rendering function could have additional input parameters, such as a random seed or a layout heuristic to optimize, that tweak the result. We could further present a number of pre-rendered solutions to choose from, based on frequently chosen parameter values. Furthermore, after rendering, the user can make manual improvements in the CS model. Perhaps a feature in the CS editor to "freeze" the AS (guaranteeing that the user cannot accidentally alter the AS while altering the CS) could be beneficial here. This would be easy to implement since we know when a CS change causes a AS change. Finally, by relying on versioning to record *everything*, changes do not have to be rendered (or parsed) immediately. For instance, in a scenario where a Statechart is being edited through a textual syntax, (interactively) rendering the visual syntax may be postponed until the visual syntax is actually opened. In the end, empirical study should point out preferred workflows. Our work can become a basis for such study.

4. Related Work

For space reasons, we limit ourselves in the following to model versioning approaches. A comprehensive overview of this topic by Brosch et al. can be found in [11]. Various (overlapping) definitions of conflicts and/or inconsistencies are given in the literature. Mens [15] distinguishes syntactic, structural, and semantic conflicts, but also calls all of them inconsistencies. Taentzer et al. provide in [16] a precise formal definition of conflicts based on graph theory. They introduce the terms state-based and operation-based conflict (not to be mixed up with state-based vs. operation-based versioning approaches). Our definition of conflict matches the latter.

Brosch et al. present in [17, 18] a taxonomy of conflicts together with a visualization of the different conflicts (conflict diagram). In this taxonomy, conflicts are either "overlapping changes" (competing changes, similar to our "conflicts") or (constraint) "violations". The overall approach is based on a tight coupling between AS and CS, but is extensible with language-specific features.

In the graph transformation rules applied for conflict detection and resolution, only the elements of the AS are considered and also only (abstract) elements of the conflict diagram are generated. Layout problems arising during the rendering process of the conflict diagram are partially solved using simple heuristics preserving layout continuity. Layout conflicts in the original model are only handled in the simplest way by just preserving the layout of the merging user. More complex layout problems are not considered (and even cannot due to the restriction to the level of AS).

In contrast to this approach, we explicitly distinguish between CS and AS and thus, are able to detect and handle conflicts more precisely. In addition, due to our loose coupling of CS and AS via a correspondence model, we directly support blended modeling environments which is not the case in the AMOR project [19]. The proposed "conflict-tolerant merging of models" [13] by Wieland et al. is enabled by design in our approach including the information about "how this conflict was resolved and who was responsible for the resolution decision" [13]. The persistence of all (conflicting) deltas together with meta-information of the delta author fulfills this requirement.

In [9], van Tendeloo et al. present a more flexible framework for collaborative model development with a clear separation of CS and AS, each corresponding to their own metamodel. Our underlying idea of a flexible, language-independent modeling environment that supports the indeterminacy needed for blended modeling is based on this work [20]. The problem with van Tendeloo's framework is that it leaves open when and how the synchronization of CS and AS can and should be done. Our approach attempts to bridge this gap.

The work of Pietron et al. [12] presents an operation-based versioning system propagating user-performed edit operations. Compared to the approach in the course of this paper, their work focuses mainly on the AS and does not support multiple CS 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.

5. Conclusion

We presented a set of components and interfaces for collaborative modeling environments supporting CS reuse and blended modeling through loose CS/AS coupling, and bi-directional synchronization. By explicitly versioning CS, AS, and their correspondence, we can distinguish between conflicts on each of these levels, and synchronizations can happen asynchronously or even be postponed, which is especially useful when dealing with missing information while rendering. By allowing versions with unresolved conflicts to be persisted, we support deferred resolution of merge conflicts in a collaborative way, as suggested by Wieland et al. [13].

Currently, we are working on a formal, yet abstract description of a conflict and inconsistency detection algorithm and its demonstration in a prototypical web-based implementation of such a modeling environment. Another future research direction is the consideration of multiple layers between CS and AS with increasing abstraction levels (insideness, connectedness relation, etc.), to allow even more CS reuse between languages.

Acknowledgments

Author J. Exelmans is an SB PhD 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] F. Ciccozzi, M. Tichy, H. Vangheluwe, D. Weyns, Blended Modelling - What, Why and How, in: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), 2019, pp. 425–430. doi:10.1109/MODELS-C.2019.00068.

[2] S. Abrahão, F. Bourdeleau, B. H. C. Cheng, S. Kokaly, R. F. Paige, H. Störrle, J. Whittle, User experience for model-driven engineering: Challenges and future directions, in: 20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, IEEE, 2017, pp. 229–236. doi:10.1109/MODELS.2017.5.

[3] I. David, K. Aslam, S. Faridmoayer, I. Malavolta, E. Syriani, P. Lago, Collaborative model-driven software engineering: A systematic update, in: 24th International Conference on Model Driven Engineering Languages and Systems, MODELS 2021, IEEE, 2021, pp. 273–284. doi:10.1109/MODELS50736.2021.00035.

[4] git Version Control System, 2022. URL: https://git-scm.com/, last visited: 10/05/2022.

[5] Apache Subversion, 2022. URL: https://subversion.apache.org/, last visited: 10/05/2022.

[6] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, J. 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, p. 763–774. doi:10.1145/2950290.2950315.

[7] B. Nuseibeh, S. Easterbrook, A. Russo, Making inconsistency respectable in software development, Journal of Systems and Software 58 (2001) 171 – 180. doi:10.1016/S0164-1212(01)00036-X.

[8] E. Guerra, J. 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), MODELS '18, ACM, New York, NY, USA, 2018, pp. 23–33. doi:10.1145/3239372.3239376.

[9] Y. V. Tendeloo, H. Vangheluwe, Unifying model- and screen sharing, IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE) (2018) 127–132. doi:10.1109/WETICE.2018.00031.

[10] A. Schürr, Specification of graph translators with triple graph grammars, in: E. W. Mayr, G. Schmidt, G. Tinhofer (Eds.), Graph-Theoretic Concepts in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 151–163. doi:10.1007/3-540-59071-4_45.

[11] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, M. Wimmer, An Introduction to Model Versioning, in: M. Bernardo, V. Cortellessa, A. Pierantonio (Eds.), Formal Methods

for Model-Driven Engineering, volume LNCS 7320, Springer, Berlin, Heidelberg, 2012, pp. 336–398. doi:10.1007/978-3-642-30982-3_10.
[12] J. Pietron, F. Füg, M. Tichy, An operation-based versioning approach for synchronous and asynchronous collaboration in graphical modeling tools, in: L. Iovino, L. M. Kristensen (Eds.), STAF 2021 Workshop Proceedings, volume 2999 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021, pp. 88–89. URL: http://ceur-ws.org/Vol-2999/fpvmdata4mdepaper3.pdf.
[13] K. Wieland, P. Langer, M. Seidl, M. Wimmer, G. Kappel, Turning Conflicts into Collaboration, Computer Supported Cooperative Work (CSCW) 22 (2013) 181–240. doi:10.1007/s10606-012-9172-4.
[14] G. Winskel, An introduction to event structures, in: J. W. de Bakker, W. P. de Roever, G. 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, volume 354 of *Lecture Notes in Computer Science*, Springer, 1988, pp. 364–397. doi:10.1007/BFb0013026.
[15] T. Mens, A state-of-the-art survey on software merging, IEEE Transactions on Software Engineering 28 (2002) 449–462. doi:10.1109/TSE.2002.1000449.
[16] G. Taentzer, C. Ermel, P. Langer, M. Wimmer, Conflict Detection for Model Versioning Based on Graph Modifications, in: H. Ehrig, A. Rensink, G. Rozenberg, A. Schürr (Eds.), Graph Transformations, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2010, pp. 171–186. doi:10/dcjxkr.
[17] P. Brosch, M. Seidl, M. Wimmer, G. Kappel, Conflict visualization for evolving UML models, J. Object Technol. 11 (2012) 2: 1–30. doi:10.5381/jot.2012.11.3.a2.
[18] P. Brosch, Conflict Resolution in Model Versioning, Ph.D. thesis, Vienna University of Technology, Vienna, 2012. URL: https://publik.tuwien.ac.at/files/PubDat_208975.pdf.
[19] AMOR — Adaptable Model Versioning, 2009. URL: http://modelversioning.org/, last visited: 10/05/2022.
[20] L. Nachreiner, A. Raschke, M. Stegmaier, M. Tichy, CouchEdit: A Relaxed Conformance Editing Approach, in: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS-C), ACM, 2020, pp. 1–5. doi:10.1145/3417990.3421401.