An Operation-based Versioning Approach For Synchronous and Asynchronous Collaboration in Graphical Modeling Tools

Jakob Pietron[1], Fabian Füg[1] and Matthias Tichy[1]

[1]*Institute of Software Engineering and Programming Languages, Ulm University, Ulm, Germany*

Abstract

In the domain of Model-driven Engineering (MDE), modeling of software and technical systems is often a collaborative and interactive activity performed by several people. However, existing tools do not offer sufficient collaboration features as reported by studies conducted with industrial practitioners. In this paper, we introduce a new operation-based approach enabling both synchronous and asynchronous collaboration in graphical modeling tools. The presented approach is capable of conflict detection, resolving, branching, and merging. Furthermore, we demonstrate how a seamless transition between both collaboration modes can be ensured. We define user performed edit operations, such as adding a new block or changing a property's value, as first-class citizens. Edit operations do not have to be atomic and can result in multiple atomic operations which are finally applied to the local model. Both kinds of operations are persisted in a sequential history and they are also distributed to other connected clients through a central server, which ensures a global unified history across all clients. Due to the sequential history consisting of operations and the change information they contain, we can apply a conflict detection method that is able to narrow down conflicts to the minimal possible set of individual conflicting operations for manual resolution while automatically merging the remaining changes.

Keywords

collaboration, synchronous, asynchronous, graphical, modeling, conflict detection, conflict resolution

1. Introduction

Due to the ever increasing size of modern software and systems, the need for collaboration features of software developing tools has grown continuously. Especially modeling is a creative and interactive but mostly collaborative work. This raises the need for suitable collaboration features in modeling tools, such as versioning and change propagation, but also conflict detection and resolving should be handled appropriately. Studies conducted in industry targeting modeling tools report problems related to insufficient or even missing collaboration features [1, 2, 3].

In general, collaboration can be classified by means of the classification framework by Franzago et al. [4] into *synchronous* or *asynchronous* collaboration. A further classification addresses the way of how differences between versions are described or identified. Versioning approaches can be either *state-based* or *operation-based* [5]. State-based comparison identifies differences between two versions by matching similar elements and identifying contradictions.

Operation-based comparison, instead, records operations which are applied to the model to be versioned and performs a conflict detection based on the operations.

Besides technical challenges, collaboration in model-driven software development creates new challenges in terms of user experience. Especially if modelers collaboratively create a model, it becomes important for them to trace changes made by others and understand how a model has evolved over time. We argue that in a graphical modeling tool recording every edit operation a user performs is a suitable way to record a user's modeling intention. Instead of calculating state differences or a possible sequence of edit operations a posteriori, such as state-based approaches do, we assume that having access to the list of performed edit operations improves the comprehensibility of evolving models for users and, consequently, the user experience, too.

In this paper, we introduce an operation-based approach that supports synchronous and asynchronous collaboration as well as a seamless transition between the two operation modes based on user-made edit operations, e.g., creating a new block or adding a property. These edit operations are utilized for versioning purposes by storing them in a model history (instead of persisting the whole state of the model) and also for collaboration purposes by distributing them through a central server to other connected clients. The central server detects conflicts based on the received edit operations. We demonstrate the applicability of our approach by extending an existing tool named Iris [6]. Iris is a graphical modeling tool which is developed to create roadmaps for the automotive domain by modeling technical systems and identifying their future needs.

The remainder of this paper is as follows: In the following Section 2 we discuss other collaboration approaches. In Section 3, we explain the architecture of our approach and in Section 4 we present the way we store operations for versioning purposes. How we address both collaboration modes and a fluent transition between both is explained in Section 5. Finally, in Section 6, we summarize our approach and outline future work.

2. Related Work

Graphical models are often stored in a serialized and textual file. For example, XMI [7] is a popular format in the MDE domain. The most basic approach to version these serialized models is using a version control system (VCS) such as Git [8]. However, understanding changes or even handling conflicts at this textual level is not a way that can be understood by human users. Conflicts must be lifted to a higher level a user can understand which is the graphical syntax of the model. For this purpose, EMF Compare [9, 10] is a popular tool. It is a state-based approach and instantiates each model version, calculates the differences at the abstract syntax level, and visualizes them even on concrete syntax level when customized. The usual combination of a classical VCS and EMF Compare only supports asynchronous collaboration. A drawback of EMF Compare — as for any other state-based approach — is that it is only capable of calculating deltas between different model states and not the actual performed changes which were applied to the model. SiLift [11, 12] goes a step further by addressing this drawback. It does not only calculate the differences between different versions of a model, but also tries to identify a sequence of model transformations, which must be previously specified by a user, that could transform one version to the other version. The a posteriori calculated result is a possible sequence of

transformations, but not necessarily the actual sequence.

Yohannis et al. [13] developed CBP that combines state-based with operation-based model persistence. Changes to the model are recorded in a XML-like textual syntax and appended to a model's XMI file. The approach relies on a classical VCS, hence, it supports only asynchronous collaboration. However, Yohannis et al. demonstrated that their operation-based approach is faster in terms of version comparison than the state-based comparison of EMF Compare [14].

There also exist other approaches that can be classified as *operation-based*. CoOBra [15] supports both synchronous and asynchronous collaboration but is not capable of branching as known from other VCS. Furthermore, it ensures no globally consistent order of the performed operations. Also in synchronous mode, conflicting changes are discarded.

Another operation-based approach, Kotelett [16, 17], defines a special *Difference Language (DL)* to represent deltas between two versions. Kotelett supports both synchronous and asynchronous collaboration but no conflict detection in both modes. While so called *micro changes* are exchanged during synchronous collaboration, after persisting a model, these micro changes are dropped and the state-based delta between two model versions is calculated.

We conclude that various approaches enabling either asynchronous or synchronous collaboration for graphical modeling environments exist. Some of them also support both modes but suffer from missing conflict detection in synchronous mode. Further, some approaches do not persist a model's detailed and globally ordered editing history.

3. Architectural Blueprint

Our approach treats edit operations as first class citizens. This requires a technique to store and process operations instead of storing a full state of a model. We also need an application architecture that is *event-driven*, or in our terms driven by user initiated edit operations. Over the course of this chapter, we explain the architecture of our approach as well as the versioning concepts.

To enable synchronous collaboration our architecture is a client-server architecture and is outlined in Figure 1. It follows the *flux* pattern as introduced by Facebook [18], which is a state-of-the-art pattern for web applications that reflects our requirement of handling user triggered edit operations as first-class citizen. Adopting *flux* leads to an unidirectional data flow of edit operations through our application. Composite operations are triggered by a user within
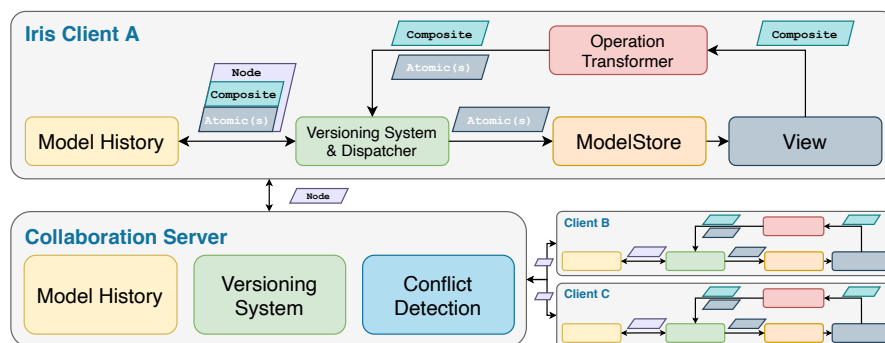

Figure 1: Overview of the client-server architecture of our collaboration approach. Edit operations are used for both versioning and collaboration purposes.

the graphical *view*. A central *dispatcher* is responsible of delegating *composite* and derived *atomic operations* to corresponding parts of the application. The *model store* holds the currently displayed version of the model and can update the model by applying atomic operations to the model received from the dispatcher. Finally, the store notifies the view that it can update itself because the model has changed. To support versioning, we extended the dispatcher to not only delegate the operations to the model store to update the current model, but also to the *model history* which stores all operations applied to the model in the past for versioning purposes as we describe in more detail in Section 4.

In contrast to the client application, the collaboration server only offers three features. First, the server can handle several simultaneous modeling sessions. For every modeling session, the server broadcasts received operations from one connected client to all other connected clients. Second, the server stores for each modeling session all operations within its *model history*. The server is neither capable of applying these operations nor does it hold the state corresponding to a version of the model. Third, the server is responsible for detecting conflicts and ensuring an eventual consistent and equal model history across all clients.

Our approach introduces several classes to represent model elements and operation types. Figure 2 shows the generic metamodel of our approach and its application in Iris. Every *Element* but also every *EditOperation* is identified by a globally unique ID [19]. A *Model* can consist of several abstract *Elements* which can contain further *Elements*. All *Elements* can have several attributes identified by a name. Concrete *Elements* needs to be defined by the adopting application. An *EditOperation* has exactly one *target* element and depending on its *type* also a *parent* element. For instance, if an *EditOperation* is of type *Create*, it has the new element as *target* and the element in which the new element is to be created as *parent*.

In our approach, operations should represent the user's intention in the form of those operations the user can trigger within the graphical modeling tool. From this it follows that they may not be always atomic and we need to distinguish between two kinds of operations. *CompositeOperations* correspond to the actual user operation and do not have to be atomic. They can be decomposed into one or many *AtomicOperations*. In Iris, this is achieved by an *OperationTransformer*. For example, in Figure 3, a screenshot of Iris shows the two implemented types of *Elements*: *Block* and *Property*. When a user wants to delete *BlockA* this corresponds to the *CompositeOperation delete*(*BlockA*). However, this operation is decomposed to the *AtomicOperations delete*(*Prop1*), *delete*(*Prop2*), and *delete*(*BlockA*).
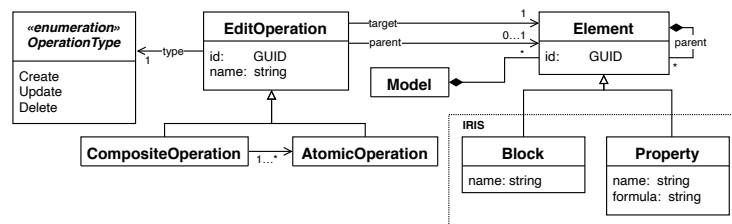
Figure 2: Generic metamodel of our approach. Contents of dotted square are specific to graphical modeling tool Iris
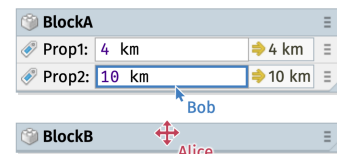
Figure 3: User awareness during synchronous collaboration aims to avoid conflicts.

4. Versioning and Data Persistence

A main idea of our concept is to persist user-triggered operations instead of the whole model state which classifies our approach regarding Brosch et al. [5] as an *operation-based* versioning approach. While state-based approaches mostly utilize external VCS to version a serialized state of the model, our approach, instead, uses edit operations for data persistence and versioning. In the following, we introduce the versioning and persistence technique of our approach in detail.

A common way to modify an application's state is to manipulate its state directly, e.g., by overwriting an old value with a new one. The drawback of this approach is that old model states or versions are no longer available and no information about the operation that changed that value is available. Instead, our versioning approach is based on operations. A composite operation o_k deterministically triggers an model's valid state v_j to be transformed to an also valid successor state v_k what can be expressed by $v_j \xrightarrow{o_k} v_k$ where v_j is predecessor of v_k ($v_j \prec v_k$). To rebuild an arbitrary past version of the model v_x, all operations $\preceq o_x$ have to be sequentially reapplied to an initial (empty) state. That implies that operations are persisted in a causally correct order. In the literature this pattern is also referred to as Event Sourcing (ES) [20].

However, a sequential list does not offer the possibility to store arbitrary competing versions or branches which are predecessors of a common ancestor or model version. Therefore, we implemented a data structure based on a directed acyclic graph (DAG). This enables us to store multiple competing versions and consequently support branching and merging, as illustrated in Figure 4. This comes into particular account when users collaborate asynchronously and switching to synchronous collaboration mode. Furthermore, since the graph is implemented as an acyclic one, we can ensure that no older version can be based on a newer one.

The implemented DAG consists of multiple *Nodes*, in general, and *EditNodes* and *MergeNodes*, in particular, as illustrated by the metamodel in Figure 5. Each node of the graph has a globally unique ID that also serves as identifier of the corresponding model version. Furthermore, a node has a reference *prev* to its predecessor node. In addition, an *EditNode*, highlighted in orange color, holds a *CompositeOperation* and the derived *AtomicOperation(s)*. A node needs to persist both types of operations: in terms of conflict detection, as described later, *AtomicOperations* would be sufficient, but, to reach our goal of representing the history of a model through edit intention of users, we also include the *CompositeOperation*.

A branch can be created by two nodes referring to the same predecessor node, e.g., in Figure 4 both nodes B and D point to node A as common ancestor. This can happen explicitly by a
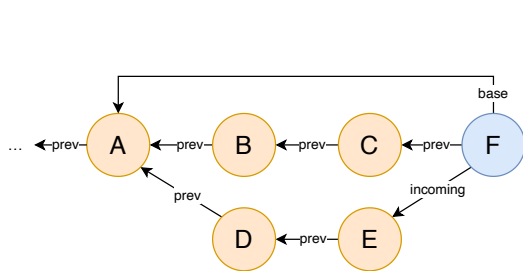
Figure 4: DAG-based data structure consisting of *EditNodes* (orange) and *MergeNodes* (blue) to enable versioning.
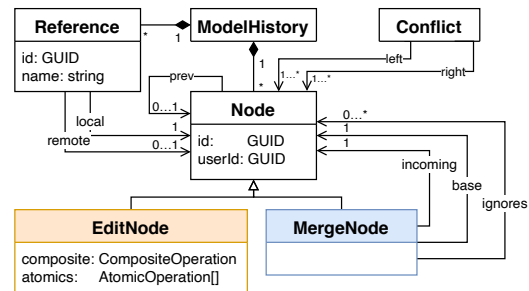
Figure 5: Metamodel of underlying data structure which implements a directed acyclic graph.

user who creates a new branch or implicitly by working in asynchronous mode while other users also working in synchronous mode. To merge two branches or competing model versions, *MergeNodes*, highlighted in blue color, are needed. They also enable conflict resolution as we describe in the next section. To handle multiple branches, the DAG can hold multiple *References* each pointing to a *Branch*.

MergeNodes have two more pointers besides the reference to their predecessor. First, a reference to the *incoming* branch by pointing to its latest node and, second, a reference to the last common ancestor of both branches *prev* and *incoming*. For instance, in Figure 4, node *A* is the last common ancestor of the branches merged by node *F*.

As introduced previously, to restore a specific version we reapply a sequential list of operations to the model. Therefore, we require a topological order to derive a linear list of nodes from the graph. Since an *EditNode* only refers to its predecessor, the ordering for this kind of nodes is obvious. For instance, in Figure 4, node *C* is the successor of *B*, which in turn is the successor of *A*. To recreate v_c, we sequentially reapply the operations of nodes A, B, C. Accordingly for v_e we need to reapply nodes A, D, E. In case of a *MergeNode*, we define the topological order as follows: first, we apply all nodes $\preceq$ than *base*, second, the sequence from *base* to *prev*, and, third, the sequence from *base* to *incoming*. Consequently, to recreate the latest model version v_f, shown in Figure 4, and assuming that there is no conflict between both branches, this would result in the sequential list of A, B, C, D, E. Since *F* is a *MergeNode* and holds no operations itself, it is not part of that list. Regarding the defined topological order, our approach utilizes a depth-first search (DFS) to derive an ordered sequence from the graph.

4.1. Conflict Detection

Since we implemented an operation-based versioning approach, also the conflict detection must be able to perform a conflict detection between two competing versions based on the recorded edit operations. We use the approach by Yohannis [13] to efficiently compare two models based on operations. The approach iterates the two competing sequences of nodes following a common predecessor node. For every *Element* the presented algorithm logs the nodes which have an effect to that element. If both competing sequences change and require the same element, this would result in a conflict.

In our implementation, the conflict detection is performed at the level of *AtomicOperations* but conflicts are generated at the level of *CompositeOperations*, because they represent the edit operation performed by a user and, thus, cannot be only partially part of a conflict. Furthermore, if a *CompositeOperation* is part of more than one conflict, those conflicts are combined into a single conflict.

4.2. Conflict Resolution

For the occurring conflicts, a user has the ability to choose for every conflict one or the other side. Assuming, in Figure 4, there are two conflicts: *B* and *E* as well as *C* and *D*. Now, the user could choose *B* instead of *E* but *D* instead of *C*. The user's choice is encoded within a *MergeNode* in a set named *ignores*. While iterating the graph to derive a sequence of nodes to recreate a specific version, nodes ignored from a *MergeNode* are removed from the list. According to the

previous example, node $F.ignores = \{E, C\}$ and, from this, the sequence to build F is A, B, D.

This is, so far, a very basic conflict resolution. However, we assume that it is quite handy, because only a minimal subset of composite operations of two competing versions is included in a conflict and not the whole version itself. Therefore, due to the fine-grained resolution of conflict detection, most competing operations are not in conflict and can be merged or rebased automatically. Additionally, even if a user decided to exclude a node from the merged version, that node or its corresponding model version is not deleted. It can easily be restored by reloading that specific version. In the future, we plan to help a user to apply ignored operations again to a merged version with a technique called micro-cherry picking as we describe in [21].
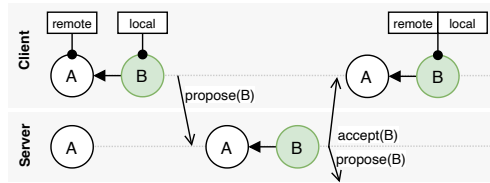
5. Collaboration

An advantage of the introduced approach is the ability to support both asynchronous and synchronous collaboration with a seamless transition from one to the other mode. In the following, we describe how the different modes operate and especially how conflicts are handled. Due to the introduced DAG-based data structure, the presented approach is capable of determining which version is based upon which previous version also in case of multiple competing versions. A basic decision of our approach is that the central server is the single source of truth. Clients can propose nodes to the server. The server can accept or reject nodes. Nodes accepted by the server can not be modified or reordered anymore, whereas updates sent from the server can reorder or withdraw unaccepted nodes at the client in case of conflicting changes. In Figure 5, the *Reference* a client uses for each branch to manage its current synchronization status is shown. On the one hand, a *Reference* points to the latest *local* node and, on the other hand, optionally, to the latest node the server has accepted. Consequently, it is possible that there are one or many nodes, which are still unaccepted. It holds that $node_{remote} \preceq node_{local}$. In general, the approach requires a reliable FIFO connection between a client and the server to ensure the right order of nodes, detect packet loss, and connection loss. For instance, we implemented this features using websockets which, in addition to be based on TCP, offer bidirectional communication by default and produce less overhead compared to plain HTTP-based communication.

In the following, we, first, explain asynchronous collaboration, second, synchronous collaboration, and, third, describe how the transition from one to the other mode is working.

5.1. Synchronous

In a synchronous collaboration setting it is important to avoid any visual input delays between a user triggered an operation and the consequences of the edit operation become visible to the user. Hence, delaying the results of an edit operation until the resulting node is proposed to and accepted by the server may increases this time span to a level that is no longer user-friendly. Nielsen [22, p. 135] summarizes that *"0.1 second is about the limit for having the user feel that the system is reacting instantaneously [...]"*. Therefore, we decided to apply an edit operation immediately at the client, perform the conflict detection a posteriori at the server, and implemented a propose / accept protocol. Initially performed benchmarks show that the generated overhead by our approach during synchronous collaboration is around 5 ms. Assuming a typical round-trip-time to be between 10 ms and 500 ms [23], the developed

approach seems promising. Because of the a posteriori conflict detection during synchronous collaboration, there are three different cases to consider:

Figure 6: Propagation of nodes in synchronous collaboration mode without any conflicts.

No interleaved updates. This can be described as the most basic case and is shown in Figure 6. A client proposes a new node B as direct successor of A to the server ($B.prev = A$). The client updates its *local* reference to B while the client's *remote* reference remains at A. B is now in an unaccepted state but will be shown to the user without any delay. When the server receives B, it checks whether $B.prev$ points to the server's last known and accepted node of this branch. Since there are no concurrent changes made to the model in this example, the server evaluates $B.prev = A$ to *true* and can, therefore, send an *accept*(B) to the proposing client and a *propose*(B) to all other clients within the same modeling session, if any. Finally, after receiving that message, the client updates its *remote* reference to B.
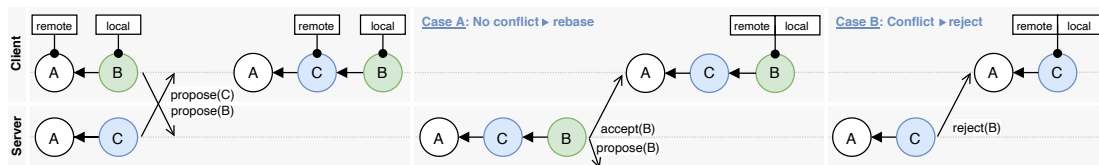
Interleaved updates without conflict. Due to network latency it can occur that the server has a new node C accepted while a client that still did not received the corresponding *propose*(C) added another node B. Both nodes C and B now point to the same predecessor node A. This example is illustrated in Figure 7 at the left.

A client always places a newly received node behind the last node that was accepted by the server, rebases all unaccepted nodes, and updates the *remote* reference accordingly. In the given example, C will placed between A and B. This can temporarily lead to an invalid model state because until now C and B might have a conflict.

The server is able to detect the concurrent changes because of $B.prev$ points to A which is not its latest known node C. Now, the conflict detection compares B and C. In the case illustrated in the middle of Figure 7, there is no conflict between B and C. Hence, the server rebases B and sends an *accept*(B) to the client which updates its *remote* reference correspondingly.

Interleaved updates with conflict. In case of a conflict between B and C, there are different ways to handle that conflict. We decided to reject the unaccepted conflicting node. This can be seen in Figure 7 at the right. The server withdraws node B and notifies the client about it with a *reject*(B) message. Following this, the client also withdraws the rejected node B and updates its *local* reference if it pointed to the rejected node before.

Instead of rejecting a conflicting node during synchronous collaboration, it would be possible to automatically branch conflicts or open a merge view every time. However, we decided against

Figure 7: Synchronous collaboration with interleaved updates without (A) and with (B) conflicts.

for the following reasons: we assume that this would interrupt the workflow more than simply rejecting a node. Moreover, due to the fine grained level of conflict detection, most interleaved updates do not conflict and can simply be rebased. Additionally, we implemented several user awareness features in the synchronous mode to prevent concurrent and conflicting updates. E.g., in Figure 3, a modeling session of Alice and Bob can be seen. Their current cursor in the form of a *ghost cursor* and their currently selected element are visualized to the other in real-time.

5.2. Asynchronous.

If the client is offline, it operates in the asynchronous collaboration mode. Equivalently to the synchronous mode, the client can append new nodes to its local model store and updates its *local* reference accordingly. Because the connection to the server is missing, new edit operations or nodes, respectively, are not sent to the server.

5.3. Transition.

The transition from *online* to *offline* is straight forward: due to the required TCP connection, the client is able to detect a connection loss and switch to the asynchronous mode. If the client is able to reestablish a connection, several synchronizations steps corresponding to the client's internal state are followed, see Figure 8. To address a new connection failure, all exchanged information during the synchronization are only stored temporarily and discarded, if necessary. In that case the synchronization must be started again. We describe each step in the following and explain how edit operations made by others during the synchronization are handled.

Wait for History. After the client has (re)established the connection, client and server must communicate the branches and their latest versions currently known to each other. Therefore, the server initiates the synchronization process with a *syncStart* message. This message contains all known references to branches including their latest known node each. The client answers also with a *syncStart* message containing its information. Both client and server store temporarily the received known references and latest nodes by their opposite. It should be noted that during the synchronization of a single client, other clients may perform concurrent edit operations. Those are received by the syncing client but withheld until further notice.

Sync. Now, the client and the server need to exchange nodes, the other party does not have in its local model history. Therefore, in the state *Sync*, the client and server send *requestNodes* messages answer with *responseNode* messages containing the corresponding nodes. Received nodes are stored within a temporary store.

Wait for SyncFin. After the server has received all requested nodes it sends sends a *syncFin* message to the client. The client responds with a *syncFin*, too, and switches to state *Wait for*
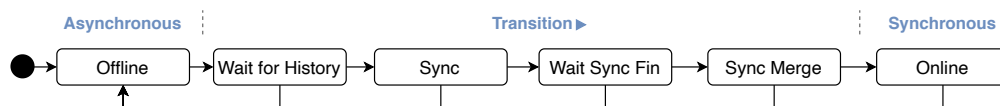
Figure 8: Client's steps of synchronization to switch form asynchronous to synchronous collaboration.

SyncFin. All nodes previously unknown to the other side have now been exchanged.

While the client remains in *Wait for SyncFin*, the server adds all received nodes from the temporary store to its model store. It iterates all branches and performs the conflict detection to decide how competing versions of a branch have to be handled. Since other connected clients may have proposed their edit operations to the server while exchanging nodes with the syncing client, the conflict detection includes those concurrent changes. There are four different cases:

1. *Fast Forward.* This case is given, if only the client has made changes to a specific branch. In case of the branch is already known to server, the server updates its reference to the latest node received by the client related to that branch. If the branch is not known to the server, it is a new branch and the server creates a reference accordingly.

2. *Update client.* This case if given, if only the server holds changes made to a specific branch. The server does not need to update its reference to that branch.

3. *Rebaseable.* This case is given, if both the client and the server made changes to a branch, but these changes are not in conflict. Consequently, the server rebases the sequence of nodes proposed by the client to the latest node of the server and updates its reference to the latest node of the client.

4. *Manual Merge Required.* In this case, in contrast to *Rebaseable*, the changes are in conflict.

The server sends the decisions made regarding each branch along with all its updated references to the client through a *syncSummary* message. All other connected clients within the same modeling session, if any, receive a *syncInfo* message containing all new nodes and the updated references.

Sync Merge.   When the client receives the *syncSummary* message, all withheld messages and nodes temporary stored so far, are appended to the local model history, because, as described before, the server conflict detection includes all these nodes. According to the decisions the server made, the client updates its references, rebases its changes, or requests the user to resolve the detected conflicts. If the user has to resolve a conflict, the decision is packed into a merge node as described in Section 4 and sent to the server. The client, finally, is in state *online* or synchronous collaboration mode, respectively.

6. Conclusion

We presented our operation-based collaboration and versioning approach supporting both synchronous and asynchronous collaboration as well as enabling a seamless transition between both modes. Therefore, we introduced a DAG-based data structure and explained how a specific version of the model can be recreated and competing branches or versions can be reconciled, again. Furthermore, we outlined how fine-grained conflicts are detected and handled.

In the future, on the one hand, we plan to investigate the formal correctness of our proposed transition process between the operation modes, and, on the other hand, we aim to develop visualizing mechanism based on our versioning approach to present the past operations and their impact to the model. Moreover, we will address ways for users to efficiently navigate through a growing history of an evolving model. We assume, that especially visual modeling languages and tools will benefit from this.

Acknowledgments

References

[1] G. Liebel, N. Marko, M. Tichy, A. Leitner, J. Hansson, Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice, Software and Systems Modeling 17 (2018) 91–113. URL: https://doi.org/10.1007/s10270-016-0523-3. doi:10.1007/s10270-016-0523-3.

[2] A. Cicchetti, F. Ciccozzi, J. Carlson, Software evolution management: Industrial practices, in: Proceedings of the 10th Workshop on Models and Evolution co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 2, 2016, volume 1706 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2016, pp. 8–13. URL: http://ceur-ws.org/Vol-1706/paper2.pdf.

[3] R. Jongeling, J. Carlson, A. Cicchetti, Impediments to introducing continuous integration for model-based development in industry, in: 45th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2019, Kallithea-Chalkidiki, Greece, August 28-30, 2019, IEEE, 2019, pp. 434–441. URL: https://doi.org/10.1109/SEAA.2019.00071. doi:10.1109/SEAA.2019.00071.

[4] M. Franzago, D. D. Ruscio, I. Malavolta, H. Muccini, Collaborative model-driven software engineering: A classification framework and a research map, IEEE Trans. Software Eng. 44 (2018) 1146–1175. URL: https://doi.org/10.1109/TSE.2017.2755039. doi:10.1109/TSE.2017.2755039.

[5] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, M. Wimmer, An introduction to model versioning, in: Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures, volume 7320 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 336–398. URL: https://doi.org/10.1007/978-3-642-30982-3_10. doi:10.1007/978-3-642-30982-3_10.

[6] A. Breckel, J. Pietron, K. Juhnke, M. Tichy, A domain-specific language and interactive user interface for model-driven engineering of technology roadmaps, in: 46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020, Portoroz, Slovenia, August 26-28, 2020, IEEE, 2020, pp. 162–170. URL: https://doi.org/10.1109/SEAA51224.2020.00035. doi:10.1109/SEAA51224.2020.00035.

[7] The Object Management Group (OMG), XML Metadata Interchange (XMI) Specification Version 2.5.1, 2015. URL: https://www.omg.org/spec/XMI/.

[8] Git, 2020. URL: https://git-scm.com/, accessed on 2020-07-06.

[9] C. Brun, A. Pierantonio, Model differences in the eclipse modeling framework, UPGRADE, The European Journal for the Informatics Professional 9 (2008) 29–34.

[10] EMF Compare | Home, 2020. URL: https://www.eclipse.org/emf/compare/overview.html, accessed on 2020-07-10.

[11] T. Kehrer, U. Kelter, M. Ohrndorf, T. Sollbach, Understanding model evolution through semantically lifting model differences with silift, in: 28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012, IEEE Computer Society, 2012, pp. 638–641. URL: https://doi.org/10.1109/ICSM.2012.6405342. doi:10.1109/ICSM.2012.6405342.

[12] SiLift: Semantic Lifting of Model Differences - SiLift - Projekte - Praktische Informatik, 2017. URL: http://pi.informatik.uni-siegen.de/Projekte/SiLift/index.php, accessed on 2020-07-10.

[13] A. Yohannis, Change-Based Model Differencing and Conflict Detection, University of York, 2020. URL: http://etheses.whiterose.ac.uk/26921/.

[14] A. Yohannis, R. H. Rodriguez, F. Polack, D. S. Kolovos, Towards efficient comparison of change-based models, J. Object Technol. 18 (2019) 7:1–21. URL: https://doi.org/10.5381/jot.2019.18.2.a7. doi:10.5381/jot.2019.18.2.a7.

[15] C. Schneider, CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten, Ph.D. thesis, University of Kassel, 2007. URL: https://nbn-resolving.org/urn:nbn:de:hebis:34-2007121319874.

[16] M. Appeldorn, D. Kuryazov, A. Winter, Delta-driven collaborative modeling, in: Proceedings of MODELS 2018 Workshops, co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018, volume 2245 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2018, pp. 293–302. URL: http://ceur-ws.org/Vol-2245/commitmde_paper_4.pdf.

[17] D. Kuryazov, A. Winter, R. H. Reussner, Collaborative modeling enabled by version control, in: Modellierung 2018, 21.-23. Februar 2018, Braunschweig, Germany, volume P-280 of *LNI*, Gesellschaft für Informatik e.V., 2018, pp. 183–198. URL: https://dl.gi.de/20.500.12116/14938.

[18] Facebok, Flux | Application architecture for building user interfaces, 2021. URL: https://facebook.github.io/flux/.

[19] P. J. Leach, M. Mealling, R. Salz, A Universally Unique IDentifier (UUID) URN Namespace, 2005. URL: https://tools.ietf.org/html/rfc4122, library Catalog: tools.ietf.org.

[20] M. Fowler, Event Sourcing, 2005. URL: https://martinfowler.com/eaaDev/EventSourcing.html, accessed on 2020-07-12.

[21] J. Pietron, Enhancing collaborative modeling, in: MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings, ACM, 2020, pp. 30:1–30:6. URL: https://doi.org/10.1145/3417990.3419490. doi:10.1145/3417990.3419490.

[22] J. Nielsen, Usability engineering, Academic Press, 1993.

[23] Verizon, Monthly IP Latency Data, 2021. URL: https://enterprise.verizon.com/terms/latency/, accessed on 2021-05-02.