Guided Merging of Sequence Diagrams*

Magdalena Widl[1], Armin Biere[3], Petra Brosch[2], Uwe Egly[1],
Marijn Heule[4], Gerti Kappel[2], Martina Seidl[3], and Hans Tompits[1]

[1] Institute for Information Systems, Vienna University of Technology, Austria
{uwe,tompits,widl}@kr.tuwien.ac.at
[2] Business Informatics Group, Vienna University of Technology, Austria
lastname@big.tuwien.ac.at
[3] Institute for Formal Models and Verification, Johannes Kepler University, Austria
firstname.lastname@jku.at
[4] Department of Computer Science, University of Texas, Austin, United States
firstname@cs.utexas.edu

Abstract. The employment of *optimistic model versioning systems* allows multiple developers of a team to work independently on their local copies of a software model. The merging process towards one consolidated version obviously turns out to be challenging when performed without any tool support. Recently, several sophisticated approaches for model merging have been presented. However, even for multi-view modeling languages like UML, which distribute the information on the system under development over different diagrams, diagrams of different views are merged independently of each other. Hence, inconsistencies between different views are likely to be introduced into the merged model. We suggest to solve this problem by exploiting information stored in one view as constraint for the computation of a consolidated version of another view. More specifically, we demonstrate how state machines can guide the integration of parallel changes performed on a sequence diagram. We give a concise formal description of this problem and suggest a translation to propositional logic.

1 Introduction

At least since Brooks' 1987 publication on software engineering, awareness has been brought to the collective consent that software is inherently complex [7]. According to Brooks, this complexity can be split into *essential complexity* introduced by the problem domain itself, and *accidental complexity* emerging from inadequate representations of the problem domain. Essential complexity is enclosed in the very nature of software, and is thus hardly reducible. To mitigate accidental complexity, software engineering practice is shifting from code-centric development to a model-driven engineering (MDE) [4] paradigm, which is based on multi-view modeling languages like the Unified Modeling Language (UML). In MDE, software models are not only employed as informal design

* This work was supported by by the Vienna Science and Technology Fund (WWTF) through project ICT10-018, by the fFORTE WIT Program of TU Wien and the Austrian Federal Ministry of Science and Research, by the Austrian Science Fund (FWF) under grants P21698, J3159-N23, and S11409-N23, and by DARPA contract number N66001-10-2-4087.

sketches, but serve as full development artifacts used for automatic code generation. UML introduces different views on the system under development in order to make the complexity of large systems manageable. These views are represented as different diagrams, each highlighting a certain aspect of the system, while abstracting from others. For example, the internal behavior of objects is shown in state machines whereas interactions between objects are specified by sequence diagrams.

However, not only the sofware itself, but also the process of building software is inherently complex. Already 40 years ago [19], software engineering was defined as the multi-person construction of multi-version software. The combination of multiple persons and multiple versions of software is thus, in addition to the complexity of the software itself, another important source of complexity. Consequently, tools supporting team work and change management emerged [12], in particular, *version control systems* (VCS). Two different versioning paradigms are distinguished. On the one hand, *pessimistic versioning systems* grant exclusive access to a resource by locking this resource for all but one developer, with the consequence that no conflicts are possible, but also all but developer are interrupted in their work. On the other hand, *optimistic versioning systems* manage parallel modifications of a software artifact by comparing and merging independently evolved versions with a common ancestor. In the rest of this paper, we consider optimistic versioning. Initially, VCS were applied only to textual artifacts such as source code, but with the increasing importance of software models in the software engineering process, the need to version control also the modeling artifacts became evident. However, due to the graph-based nature of models, existing VCS, which have been successfully employed for source code, are only of limited value for model versioning. Thus, dedicated model versioning systems based on different algorithms, are necessary. Several approaches have been presented recently [9], for both single-view and for multi-view modeling languages like UML. As far as version control for multi-view models is concerned, however, current approaches merge each diagram individually and ignore valuable information spread across different diagrams. By ignoring this information, false conflicts can be reported or unsatisfactory merge results returned, e.g., inconsistencies between different views of the software model are introduced.

In this paper, we suggest to consider constraints imposed by information distributed over some diagrams when merging others. In particular, we show how two versions of a sequence diagram can be consistently merged by taking the behavior expressed by state machines into account. Since the merged version is not unique in general, the goal is to precalculate a set of consistent merges to support the modeler to integrate the modifications. Given a multi-view modeling language with several concepts as found in UML, we give a formal specification of the merging problem, which allows for a direct encoding of the merging problem to a formalism for which tool support is available. We chose propositional logic as host language to represent the merging problem of sequence diagrams in terms of a satisfiability problem (SAT), because the required constraints are directly transferable to SAT and powerful tools, so-called SAT solvers, are available.

This paper is structured as follows. In Section 2, we discuss an example illustrating the problems occurring during the merge of sequence diagrams. We review related work in Section 3. In Section 4, we give an overview of the modeling language concepts considered in this paper in terms of a graphical metamodel, which we then transform

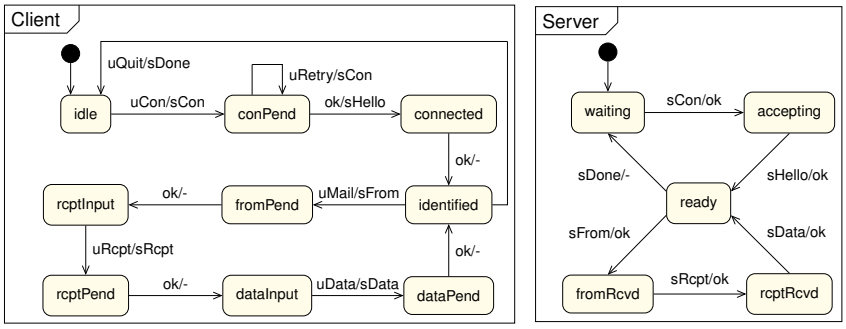

Fig. 1: State machines of an email client and an email server.

to a formal representation. On this basis, we give a concise definition of the sequence diagram merging problem in Section 5. The translation of this problem to the satisfiability problem of propositional logic is explained in Section 6. We present our implementation and a first evaluation in Section 7. Finally, we conclude with an outlook to future work.

2 A Motivating Example

The following example on an email protocol is to motivate the approach developed in this paper. Fig. 1 shows state machines of an email-client and an email-server, respectively, implementing a simplified variant of the *Simple Mail Transfer Protocol* (SMTP). For the sake of readability, only basic sending functionality is realized and no error handling is included. The initial state of each state machine is indicated by an incoming arrow from a black circle. States are connected to each other by transitions. Each transition carries a label that consists of two parts, separated by a "/". The string on the left indicates a *trigger*, whose receipt in the source state of the transition causes the state machine to change its state to the target state of the transition. The string on the right of each transition indicates a set of *effects*, which are symbols that are sent when the transition is executed and which may again trigger state transitions in the same or other state machines. For example, the state machine Client starts in state Idle and waits until it receives uCon, which causes its transition to state conPend. During the execution of the transition, it sends the trigger sCon, which is received by the state machine Server, and causes its transition from state waiting to accepting. During the execution of this transition, Server triggers ok, which is again received by Client, triggering the transition to state connected, and so on.

A valid model might only partially specify the system under consideration. For example, the first transition on Client is triggered by a user, for which no state machine is defined. In this case, an unconstrained state machine is assumed. Such a state machine contains only one state from which any symbol is received and sent.

Communication scenarios between users, clients, and servers are modeled by *sequence diagrams* showing sequences of exchanged messages. Sequence diagrams describe interactions where the interaction partners, the *lifelines*, are instances of state machines. Fig. 2 shows the three sequence diagrams SD^o, SD^α, and SD^β. The lifelines are represented by the rectangles labeled with u:User, c:Client, and s:Server and vertical

dashed lines. Each label contains the name of the lifeline on the left of the colon (e.g., c) and the name of the state machine instantiated by the lifeline on the right of the colon (e.g., Client A message is represented as arrow connecting two lifelines and contains a symbol which can be found as effect on some transition of the state machine of the sender as trigger on some transition of the state machine of the receiver. A sequence diagram is consistent with the state machines that are instantiated by its lifelines, if for each lifeline the sequence of received messages is a path in the state machine. For example, for the uppermost diagram in Fig. 2, SD^o, the sequence of received messages for lifeline c:Client causes triggers uCon $\rightarrow$ ok $\rightarrow$ ok. This sequence is also found as a path in the corresponding state machine, namely connecting states idle $\rightarrow$ conPend $\rightarrow$ connected $\rightarrow$ identified. A similar argument holds for s:Server. Since for u:User we assume an unconstrained state machine, no restriction is imposed on the messages, and the order of the messages received by u:User. So this lifeline is also consistent.

Consider the following evolution scenario. Starting from the sequence diagram SD^o of Fig. 2, which shows the authentication process of an email protocol, two modelers, Alice and Bob, independently perform some modifications. Alice extends the scenario with a logout message resulting in the revised sequence diagram SD^α, while Bob adds the communication necessary to send an email, manifesting in revision SD^β. Trying to merge the modifications of both modelers without any additional information, it is not automatically decidable in which order the added messages from both revisions should be arranged. Hence, we have a *merge conflict*.

It thus has to be decided manually how the changes are integrated. Several syntactically correct merges of the sequence diagram are possible, namely all possible permutations of the two concatenated sequences that preserve the relative order of the messages. However, many of these options turn out to be inconsistent with the state machines. When taking the state machines into account, then only one merged version is possible: Alice's modifications have to be appended after Bob's changes, because otherwise the sequence diagram would model a scenario which is not allowed by the state machines.

3 Related Work

The requirements of model versioning systems strongly diverge from the requirements of traditional versioning systems for text-based artifacts like source code [1–3]. In consequence, several approaches to conflict detection algorithms and model merging strategies have been presented over the last few years (cf. [9] for a detailed survey). These approaches are either realized on the generic metamodeling level resulting in language independent solutions, or use language specific information in order to yield better support for a specific modeling language. Our approach falls into the latter category. However, we are not aware of any approach dealing with the merge problem of sequence diagrams where the sequence diagrams have to be kept consistent with the state machine view. Westfechtel [26] discusses the merge of ordered features in EMF models by aggregating elements into linearly ordered clusters. The order within a cluster is determined either randomly or by a user. Since the merge is performed on the metamodel level to keep the approach generic, the information available within the model cannot be

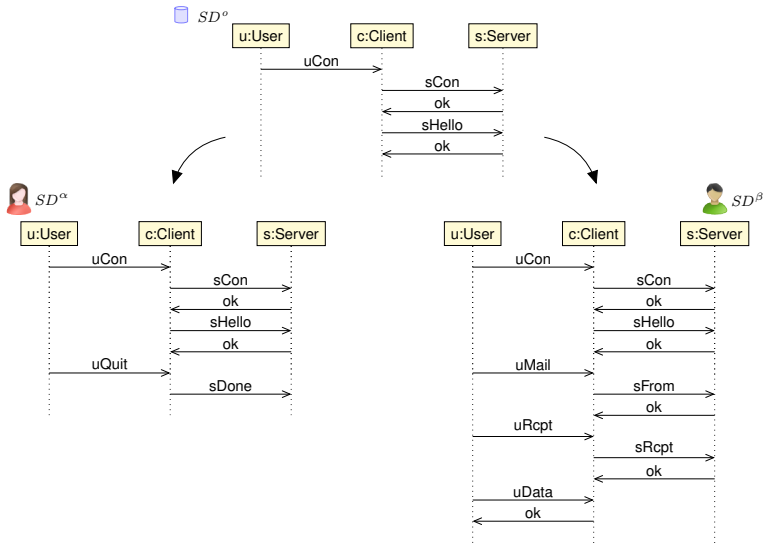

Fig. 2: Evolution of a sequence diagram.

used for merging. Gerth et al. [16] provide dedicated merge support for business process models ensuring a consistent outcome. They formalize process models as process terms and utilize a term rewriting system to detect and interactively resolve merge conflicts. However, there is no support to calculate all valid merge solutions. Cichetti et al. [11] propose to define conflict patterns which might be tailored towards the application on sequence diagrams. Such a conflict pattern might be equiped with a reconciliation strategy for resolving the conflict. In [17] an approach for merging two state machines is presented. This approach exploits syntactical as well as semantical information provided by the models in order to compare variants and perform consistency checks.

Outside the research of model versioning several approaches have been presented to verify the conformance between different views of a model and to eliminate inconsistencies. Diskin et al. [13] present a framework based on category theory for consistency checking of views. Therefore, they first integrate the relevant parts of the metamodels into one global metamodel such that all instance models become instance models thereof. These instance models may then be checked for inconsistencies. Van Der Straeten et al. [24] use the SAT-based constraint solver Kodkod to detect and resolve inconsistencies between class and sequence diagrams. Egyed [14] proposes to identify inconsistencies in an incremental manner. Sabetzadeh et al. [21] present an approach to check consistency between a set of different, but overlapping models. Therefore, they merge this set of models to one model. Tsiolakis [23] suggests to collect constraints distributed over views like the class diagram or state machines and integrate them in the sequence diagram in terms of state invariants yielding pre- and postconditions for individual messages.

In the context of model versioning, these approaches may be used to check if the merged version introduces inconsistencies, i.e., to perform quality control on the merge result. In [8], we proposed to use model checking to validate the merged version of an evolving sequence diagram. No support for the merging process itself is provided.

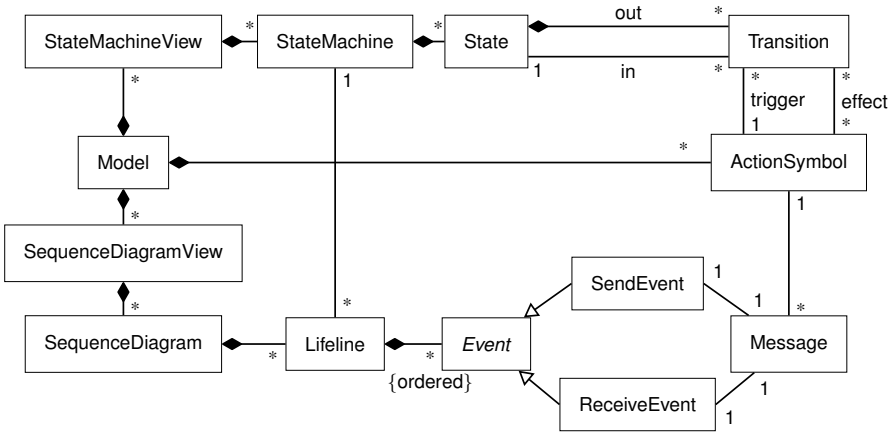

Fig. 3: Metamodel of the *tMVML*.

4 The Modeling Language *tMVML*

In order to give a concise definition of the sequence diagram merging problem, an explicit statement on the modeling language concepts is essential. Therefore, we define the modeling language *tMVML*, the *Tiny Multi-View Modeling Language* in terms of a metamodel. The concepts and terminology of *tMVML* are inspired by the Unified Modeling Language (UML) of the OMG [18]. The compactness of the metamodel allows not only a focused presentation of our approach, but also a direct technical realization discussed in Section 7. Concepts to describe the static structure of a system as found in a class diagram and specification facilities for behavior as offered by the activity diagram are not relevant for this work and are omitted. However, interfaces to other views and advanced concepts of state machines and sequence diagrams not discussed in this paper are planned for later versions of *tMVML*.

In this section, we first define the language concepts handled by our approach in terms of a metamodel. With the same motivation for the works on the formalization of UML [15], we then present a formalization of the concepts of *tMVML* suitable for our purposes. This formalization enables us to precisely define the sequence diagram merging problem in the context of model versioning.

4.1 The *tMVML* Metamodel

The implementation of the *tMVML* metamodel is available at our project website[5]. We consider the excerpt relevant for this work, depicted in Fig. 3. The metamodel contains a root class Model which contains two classes representing views, namely SequenceDiagramView and StateMachineView, and the class ActionSymbol. Instances of ActionSymbol realize the communication between different state machines and are used to describe possible communication sequences in sequence diagrams. For better readability we typeset instances of metaclasses in standard lowercase font using the same

[5] `http://www.modelevolution.org/prototypes/sdmerge`

name as the metaclass, e.g., in order to to refer to an instance of the metaclass State we simply write "state".

A state machine describes the internal behavior of a lifeline. It consists of a set of states, a transition relation between states, and a set of action symbols. Each transition contains one action symbol. The receipt of such a symbol triggers the transition, and a set of action symbols that are sent when the transition is executed. The sent symbols may trigger transitions in other state machines.

Our definition of a sequence diagram is inspired by the UML sequence diagram [18]. It consists of a set of lifelines which communicate with each other via messages. Each message contains a send and a receive event assigned to a lifeline. Events are totally ordered with respect to their lifeline. The order of events imposes an order on the messages attached to them. Each message is assigned an action symbol. In the state machine modeling the behaviour of the lifeline which receives a certain message, this action symbol triggers all transitions which carry the same action symbol as trigger. In the following, we formalize the metamodel of *tMVML* which is required for a concise specification of the sequence diagram merging problem. In this context, we give concrete examples on the usage of the different language elements.

4.2 Formalization of the *tMVML* Metamodel

Let $\mathcal{L}_{\mathcal{A}}$ be the language describing *tMVML* models defined over the alphabet $\mathcal{A} = (\mathcal{A}_S, \mathcal{A}_A, \mathcal{A}_L, \mathcal{A}_M, \mathcal{A}_E)$ where $\mathcal{A}_S$ denotes a set of states, $\mathcal{A}_A$ denotes a set of action symbols, $\mathcal{A}_L$ denotes a set of lifelines, $\mathcal{A}_M$ denotes a set of messages, and $\mathcal{A}_E$ denotes a set of events. Out of the three root elements of *tMVML*, ActionSymbol is an element of $\mathcal{A}_A$ and the elements composing the classes StateMachineView and SequenceDiagramView, sets of state machines respectively sequence diagrams, along with their components and associations, are defined in the following. Besides the language concepts and their interplay, we introduce and define important properties of a sequence diagram, namely *well-formedness*, *time consistency*, *lifeline conformance*, and *correctness*, required to formulated the sequence diagram merging problem.

By $\mathcal{P}(X)$ we refer to the power set of a set X and for any tuple $Y = (y_1, \ldots, y_n)$, by $\pi_i(Y) = y_i$ with $1 \leq i \leq n$, we refer to the projection to the i-th element. We continue to typeset instances of metaclasses in standard lowercase font.

Definition 1 (State machine). *Given the alphabet $\mathcal{A}$, a state machine is a quadruple $(S, A^{tr}, A^{\mathit{eff}}, T)$, where*

- *$S \subseteq \mathcal{A}_S$ is a set of states,*
- *$A^{tr}, A^{\mathit{eff}} \subseteq \mathcal{A}_A$ are sets of action symbols, and*
- *$T \subseteq (S \times A^{tr} \times \mathcal{P}(A^{\mathit{eff}}) \times S)$ is a relation representing the transitions between states.*

A state machine consists of a set of states, two alphabets, and transitions between states. For a transition $t \in T$ with $t = (s, a, A, s')$, s is the source state of the transition, s' the target state, a an action symbol that, when received, triggers the execution of the transition, and A a set of action symbols that are sent to other state machines when the transition is executed. The state machine Server of Fig. 1

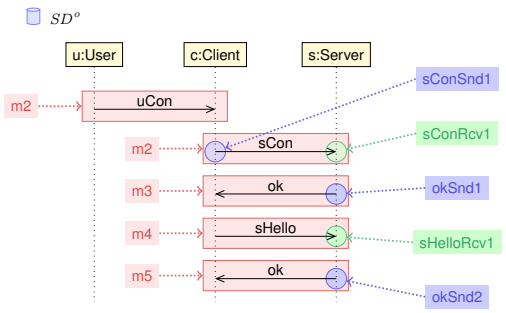

Fig. 4: Sequence diagram SD^o from Fig. 2 with additional labels.

contains states $S = \{\mathsf{waiting},\mathsf{accepting},\mathsf{ready},\ \mathsf{fromRcvd},\mathsf{rcptRvd}\}$, triggers $A^{tr} = \{\mathsf{sCon},\mathsf{sDone},\mathsf{sFrom},\ \mathsf{sRcpt},\mathsf{sData},\mathsf{sHello}\}$, and effect $A^{\mathit{eff}} = \{\mathsf{ok}\}$. Examples for transitions are $(\mathsf{waiting},\mathsf{sCon},\{\mathsf{ok}\},\mathsf{accepting})$ and $(\mathsf{ready},\mathsf{sDone},\{\},\mathsf{waiting})$. The formalization of the communication between state machines is not relevant for this paper and is therefore omitted. It can be found in [8].

Definition 2 (Sequence diagram). *Given the alphabet $\mathcal{A}$ and a set $\mathcal{SM}$ of state machines, a sequence diagram is a quadruple $(L, M, \mathsf{lprop}, \mathsf{msg})$, where*

- $L \subseteq \mathcal{A}_L$ *is a set of lifelines,*
- $M \subseteq \mathcal{A}_M$ *is a set of messages,*
- $\mathsf{lprop} : L \to (\mathcal{SM} \times \mathcal{P}(\mathcal{A}_E) \times \mathcal{P}(\mathcal{A}_E) \times \mathcal{P}(\mathcal{A}_E \times \mathcal{A}_E))$ *describes lifelines,*
- $\mathsf{msg} : M \to (\mathcal{A}_A \times \bigcup_{l \in L} \pi_2(\mathsf{lprop}(l)) \times \bigcup_{l \in L} \pi_3(\mathsf{lprop}(l)))$ *describes messages.*

For a lifeline l with $\mathsf{lprop}(l) = (SM, E^{snd}, E^{rcv}, >)$, SM is the state machine associated to l, E^{snd} and E^{rcv} are sets of send and receive events handeled by l, and the relation $>$ describes the $\{ordered\}$ constraint of the association between the classes lifeline and event in the *tMVML* metamodel. In the following, we assume that

- E^{snd} and E^{rcv} are disjoint,
- the relation $>$ is transitive and antisymmetric,
- for all $(e_1, e_2) \in >$, it holds that $e_1, e_2 \in E^{snd} \cup E^{rcv}$, and
- for two lifelines, the sets of send and receive events are pairwise disjoint.

For a message m with $\mathsf{msg}(m) = (a, s, r)$, a is the action symbol, s the send event, and r the receive event associated to m.

Fig. 4 shows the sequence diagram SD^o of Fig. 2 enriched with some information available in the abstract syntax specification of the metamodel and in the formal description of Definition 2. As usual for sequence diagrams, much information is omitted in the concrete syntax as in Fig. 2 to avoid an information overflow. In the sequence diagram SD^o, the set L contains the lifelines u, c, and s. Lifelines c and s are instances of the state machines Client and Server of Fig. 1. The state machine for u, User, is not shown. Each message is depicted as arrow between two lifelines. Each arrowhead represents a receive event and each arrowtail a send event and

it is possible that sender and receiver lifeline are identical. In Fig. 4, lifeline s handles the four events sConRcv1, okSnd1, sHelloRcv1, and okSnd2, hence $\mathsf{lprop}(\mathsf{s}) = (\mathsf{Server}, \{\mathsf{okSnd1}, \mathsf{okSnd2}\}, \{\mathsf{sConRcv1}, \mathsf{sHelloRcv1}\}, >)$ with $\mathsf{sConRcv1} > \mathsf{okSnd1} > \mathsf{sHelloRcv1} > \mathsf{okSnd2}$. Further, $\mathsf{msg}(\mathsf{m2}) = (\mathsf{sCon}, \mathsf{sConSnd1}, \mathsf{sConRcv1})$.

For ease of presentation, we additionally use the following functions to refer to elements of the sequence diagram: Given the alphabet $\mathcal{A}$ and a sequence diagram $SD = (L, M, \mathsf{lprop}, \mathsf{msg})$, let $\mathsf{lprop}(l) = (SM_l, E_l^{snd}, E_l^{rcv}, >_l)$ for each $l \in L$, and $E = \bigcup_{l \in L}(E_l^{snd} \cup E_l^{rcv})$. Then we have:

- $\mathsf{act} : M \to \mathcal{A}_A$, $\mathsf{snd} : M \to \mathcal{A}_E$, and $\mathsf{rcv} : M \to \mathcal{A}_E$, such that $\mathsf{act}(m) = \pi_1(\mathsf{msg}(m))$, $\mathsf{snd}(m) = \pi_2(\mathsf{msg}(m))$, and $\mathsf{rcv}(m) = \pi_3(\mathsf{msg}(m))$, i.e. the action symbol, send event and receive event of a message.
- $\mathsf{symb} : E \to \mathcal{A}_A$ such that $\mathsf{symb}(e) = a$ iff
  - $e \in E_l^{snd}$ for some $l \in L$ and there exists an $m \in M$ with $\mathsf{act}(m) = a$ and $\mathsf{snd}(m) = e$, or
  - $e \in E_l^{rcv}$ for some $l \in L$ and there exists an $m \in M$ with $\mathsf{act}(m) = a$ and $\mathsf{rcv}(m) = e$,

  i.e., the action symbol of the message an event is associated to. Note that each function value is unique due to the pairwise disjointness of sets of events on lifelines and distinctness of events on messages as described in Definition 2.
- $\mathsf{life} : E \to L$ such that $\mathsf{life}(e) = l$ iff $e \in \pi_2(\mathsf{lprop}(l)) \cup \pi_3(\mathsf{lprop}(l))$. Note that each function value is unique due to the pairwise disjointness of sets of events on lifelines as described in Definition 2.

We further define properties of sequence diagrams required to specify correct merge results: First, the *well-formedness* of a sequence diagram enforces an order on the events with respect to a lifeline.

Definition 3 (Well-Formedness). *A sequence diagram* $(L, M, \mathsf{lprop}, \mathsf{msg})$ *is* well-formed *iff for each* $l \in L$ *the relation* $\pi_4(\mathsf{lprop}(l))$ *is total.*

This total order over events per lifeline imposes an order over the messages of a sequence diagram, which we need for the second property: A sequence diagram is *time consistent* when any message m is not received after a message n if m has been sent before n on the same lifeline, or in other words, messages cannot overtake one another. We first define the *message ordering* relation over a sequence diagram, which describes an order of a sequence diagram's messages according to the order of its events. This relation is then used to define time consistency.

Definition 4 (Message Ordering). *Given a well-formed sequence diagram of the form* $(L, M, \mathsf{lprop}, \mathsf{msg})$*, the* message ordering *relation* $\succ \subseteq M \times M$ *contains a pair* (m, n) *iff for* $\mathsf{msg}(m) = (a_m, s_m, r_m)$, $\mathsf{msg}(n) = (a_n, s_n, r_n)$, $l = \mathsf{life}(s_m)$, $>_l = \pi_4(\mathsf{lprop}(l))$, $k = \mathsf{life}(r_m)$, *and* $>_k = \pi_4(\mathsf{lprop}(k))$ *it holds that*

- $\mathsf{life}(s_n) = l$ *and* $s_m >_l s_n$,
- $\mathsf{life}(r_n) = l$ *and* $s_m >_l r_n$,
- $\mathsf{life}(s_n) = k$ *and* $s_m >_k s_n$, *or*
- $\mathsf{life}(r_n) = k$ *and* $s_m >_k r_n$.

In SD^o of Fig. 2, the message order is given by $\mathsf{m1} \succ \mathsf{m2} \succ \mathsf{m3} \succ \mathsf{m4} \succ \mathsf{m5}$.

Definition 5 (Time Consistency). *A well-formed sequence diagram is called time consistent iff the transitive closure of its message ordering relation $\succ$ is antisymmetric.*

The third property is called *lifeline conformance* and concerns the lifelines of a sequence diagram and the state machines modeling their behaviour. Roughly, a lifeline l is conformant with the state machine SM defined in $\pi_1(\mathsf{lprop}(l))$, if the sequence of action symbols of the messages received by l occurs as path in SM.

Definition 6 (Lifeline Conformance). *Let*

- $SD = (L, M, \mathsf{lprop}, \mathsf{msg})$ *be a well-formed, time consistent sequence diagram,*
- $l \in L$ *be a lifeline with* $\mathsf{lprop}(l) = (SM, E^{snd}, E^{rcv}, >_l)$*,*
- $SM = (S, A^{tr}, A^{\mathit{eff}}, T)$ *be a state machine modelling the behaviour of* l*, and*
- $m = (e_1, \ldots, e_n)$ *be the sequence of events where for all* i, j *with* $1 \leq i, j \leq n$ *it holds that* $e_i, e_j \in E^{rcv}$ *and* $e_i >_l e_j$ *iff* $i > j$*.*

Then, the lifeline l *is* conformant *to* SM *iff there exists a sequence of transitions* $(s_1, a_1, A_1, s_2), (s_2, a_2, A_2, s_3), \ldots, (s_n, a_n, A_n, s_{n+1})$ *such that* $a_i = \mathsf{symb}(e_i)$*.*

In Fig. 4, consider lifeline s of sequence diagram SD^o. The state machine defined for s is Server, shown in Fig. 1. The sequence e for s is $(\mathsf{sConRcv1}, \mathsf{sHelloRcv1})$. The sequence resulting from the action symbols connected to these events, $(\mathsf{sCon}, \mathsf{sHello})$, can be found as path in Server, namely connecting the states waiting to accepting and accepting to ready. The lifeline s is therefore conformant with its state machine. If the action symbol of $\mathsf{m4}$ was sData instead of sHello, then s would not be conformant, as from the only state that can be reached by a transition triggered by sCon there is no outgoing transition triggered by sData. Note that effects are not considered because they do not change the state of their sender.

Finally, a sequence diagram is *correct*, if it has the three discussed properties.

Definition 7 (Correctness of a Sequence Diagram). *A sequence diagram SD is* correct *iff*

1. *SD is well-formed;*
2. *SD is time consistent;*
3. *all lifelines of SD are conformant to their state machine.*

This concludes the specification of the relevant language concepts and their properties. We provided a formal description, which will be necessary to define the sequence diagram merging problem in the next section.

5 Problem Definition

In the context of optimistic model versioning, two versions of a concurrently evolved model, the revisions, have to be combined into one consolidated version. We consider the problem of merging two *revisions* of a sequence diagram into a consolidated, correct sequence diagram using information from the original sequence diagram and the associated state machines.

Definition 8 (Revision). *A sequence diagram* $SD^\alpha = (L^\alpha, M^\alpha, \mathsf{lprop}^\alpha, \mathsf{msg}^\alpha)$ *is a* revision *of a correct sequence diagram* $SD^o = (L^o, M^o, \mathsf{lprop}^o, \mathsf{msg}^o)$ *iff*

- $L^o \subseteq L^\alpha$, $M^o \subseteq M^\alpha$,
- *for each* $l \in L^o$ *it holds that* $\mathsf{lprop}^\alpha(l) = \mathsf{lprop}^o(l)$,
- *for each* $m \in M^o$ *it holds that* $\mathsf{msg}^\alpha(m) = \mathsf{msg}^o(m)$, *and*
- SD^α *is correct.*

In Fig. 2, the sequence diagrams SD^α and SD^β are revision of sequence diagram SD^o. Please note that we consider only additions in this work. Deletions and updates have to be treated respectively.

In the following, we use the position functionpos defined over messages for the integration of two revisions of a sequence diagram. Given a correct sequence diagram $S = (L, M, \mathsf{lprop}, \mathsf{msg})$, $\mathsf{pos} : M \to \{1, \ldots, |M|\}$ such that for all $m, n \in M$ it holds that $\mathsf{pos}(m) = \mathsf{pos}(n)$ iff $m = n$ and $\mathsf{pos}(m) > \mathsf{pos}(n)$ iff $m \succ n$.

A *consolidated version* of a sequence diagram and two revisions is a correct sequence diagram that contains the messages and lifelines of the original sequence diagram and all added messages and lifelines from the revisions. The relative order of messages of the original diagram and the revisions is maintained. We define the function allow, which returns for each message a set of positions at which the message can be placed such that the relative order is kept.

Definition 9 (Allowed Positions). *Given three correct sequence diagrams* $SD^x = (L^x, M^x, \mathsf{lprop}^x, \mathsf{msg}^x)$, *for* $x \in \{o, \alpha, \beta\}$, SD^α *and* SD^β *being revisions of* SD^o, *and the position function* $\mathsf{pos}^x : M^x \to \{1, \ldots, |M^x|\}$ *with* $x \in \{o, \alpha, \beta\}$ *for the respective sequence diagram, let* $M = M^o \cup M^\alpha \cup M^\beta$ *and* $I = \{1, 2, \ldots, |M|\}$. *Then* $\mathsf{allow} : M \to \mathcal{P}(I)$ *assigns to each message* m *a set of positions, such that*

- *if* $m \in M^o$ *and* $\mathsf{pos}^o(m) = \mathsf{pos}^\alpha(m) = \mathsf{pos}^\beta(m)$ *then* $\mathsf{allow}(m) = \{\mathsf{pos}^o(m)\}$ *(*m *remains on the same position),*
- *if* $m \in M^o$ *and* $\mathsf{pos}^o(m) \neq \mathsf{pos}^\alpha(m)$ *or* $\mathsf{pos}^o \neq \mathsf{pos}^\beta(m)$, *then* $\mathsf{allow}(m) = \{\mathsf{pos}^o(m) + |N|\}$ *where* $N = \{n \in M^\alpha \mid \mathsf{pos}^\alpha(n) < \mathsf{pos}^\alpha(m)\} \cup \{n \in M^\beta \mid \mathsf{pos}^\beta(n) < \mathsf{pos}^\beta(m)\}$
- *if* $m \notin M^o$ *and* $m \in M^x$ *with* $x \in \{\alpha, \beta\}$ *then* $\mathsf{allow}(m) = \{i \in I \mid \mathsf{pos}^x(m) + |N'| \leq i \leq \mathsf{pos}^x(m) + |N''|\}$ *for*

$$N' = \begin{cases} \{n \in M^y \mid \mathsf{pos}^y(n) < \mathsf{pos}^y(m')\} & \text{if } \exists\, m' \in M^o \text{ with } \mathsf{pos}^x(m') < \mathsf{pos}^x(m) \\ \emptyset & \text{otherwise} \end{cases}$$

for $m' \in M^o$, $\mathsf{pos}^x(m') = \max_{n \in M^o \mid \mathsf{pos}^x(n) < \mathsf{pos}^x(m)} \mathsf{pos}(n)$, $y \in \{\alpha, \beta\}$ *and* $y \neq x$, *and*

$$N'' = \begin{cases} \{n \in M^y \mid \mathsf{pos}^y(n) < \mathsf{pos}^y(m'')\} & \text{if } \exists\, m'' \in M^o \text{ with } \mathsf{pos}^x(m'') > \mathsf{pos}^x(m) \\ M^y \setminus M^o & \text{otherwise} \end{cases}$$

where $m'' \in M^o$, $\mathsf{pos}^o(m'') = \mathsf{pos}^o(m) + 1$, $y \in \{\alpha, \beta\}$ *and* $y \neq x$.

m	$\mathsf{pos}^o(m)$	$\mathsf{pos}^\alpha(m)$	$\mathsf{pos}^\beta(m)$	$\mathsf{allow}(m)$	
o1	1	1	1	{1}	
o2	2	2	2	{2}	
o3	3	4	5	{6}	$N = \{$a4,b4,b5$\}$
a4	-	3	-	{3,4,5}	$N' = \emptyset$, $N'' = \{$b4,b5$\}$
a5	-	5	-	{7,8}	$N' = \{$b4,b5$\}$, $N'' = \{$b4,b5,b6$\}$
b4	-	-	3	{3,4}	$N' = \emptyset$, $N'' = \{$a4$\}$
b5	-	-	4	{4,5}	$N' = \emptyset$, $N'' = \{$a4$\}$
b6	-	-	6	{7,8}	$N' = \{$a4$\}$, $N'' = \{$a4,a5$\}$

Table 1: Allowed positions for each message of Fig. 5.

Consider the sequence diagrams SD^o, SD^α and SD^β shown in the upper part of Fig. 5, where SD^α and SD^β are revisions of SD^o. In SD^α, the message a4, and in SD^β the messages b4 and b5 are added between the original messages o2 and o3. In a merged sequence diagram, each of a4, b4 and b5 must again be placed between o2 and o3. Also, in order to maintain their order from the revisions, b5 must be placed after b4. Similar conditions are given for the messages a5 and b6 inserted after o3. Table 1 shows the values $\mathsf{pos}^x(m)$ and $\mathsf{allow}(m)$ for each message m in the upper part of Fig. 5.

If in the merged sequence diagram each message m is placed on one of the positions defined in $\mathsf{allow}(m)$ and exactly one message has been placed at each position, the merged sequence diagram is time consistent. However, in order for the merged sequence diagram to be correct, the messages have to be placed such that the lifelines conform to their state machines. If this is also the case, then the merged diagram is a *consolidated version*, defined as follows:

Definition 10 (Consolidated Version). *Given the correct sequence diagrams* $SD^o = (L^o, M^o, \mathsf{lprop}^o, \mathsf{msg}^o)$, $SD^\alpha = (L^\alpha, M^\alpha, \mathsf{lprop}^\alpha, \mathsf{msg}^\alpha)$, *as well as* $SD^\beta = (L^\beta, M^\beta, \mathsf{lprop}^\beta, \mathsf{msg}^\beta)$, *where* SD^α *and* SD^β *are revisions of* SD^o, *a consolidated version* $SD^\gamma = (L^\gamma, M^\gamma, \mathsf{lprop}^\gamma, \mathsf{msg}^\gamma)$ *is a sequence diagram such that*

1. $L^\gamma = L^\alpha \cup L^\beta$,
2. $M^\gamma = M^\alpha \cup M^\beta$,
3. *for each* $i \in \{\alpha, \beta, o\}$ *and for each* $m \in M^i$ *it holds that* $\mathsf{msg}^\gamma(m) = \mathsf{msg}^i(m)$,
4. *for each* $i \in \{\alpha, \beta, o\}$ *and for each* $l \in L^i$ *it holds that* $\mathsf{lprop}^\gamma(l) = \mathsf{lprop}^i(l)$,
5. *for each* $m \in M^\gamma$ *it holds that* $\mathsf{pos}(m) \in \mathsf{allow}(m)$, *and*
6. S^γ *is correct.*

The source of complexity in the computation of a consolidated version arises from the exponential number of possible message orderings under consideration of the allow function and the constraint arising from the lifeline-conformance requirement.

Consider the example shown in Fig. 6 and 5. The upper part of Fig. 5 depicts an original sequence diagram (SD^o) and two revisions (SD^α and SD^β) with the values of the respective pos^x function. The revised diagrams contain added messages between messages o2 and o3 and at the end of the diagram. The lower part of the figure depicts six different time consistent merged diagrams. Fig. 6 depicts two state machines describing the behavior of the lifelines. The two rightmost merged diagrams are also consolidated versions, i.e. correct with respect to the state machines depicted in Fig. 6 E.g. the sequence of actions on messages received by lifeline y of the rightmost diagram, $(\mathsf{o1}, \mathsf{b4}, \mathsf{b5}, \mathsf{a4}, \mathsf{a5})$

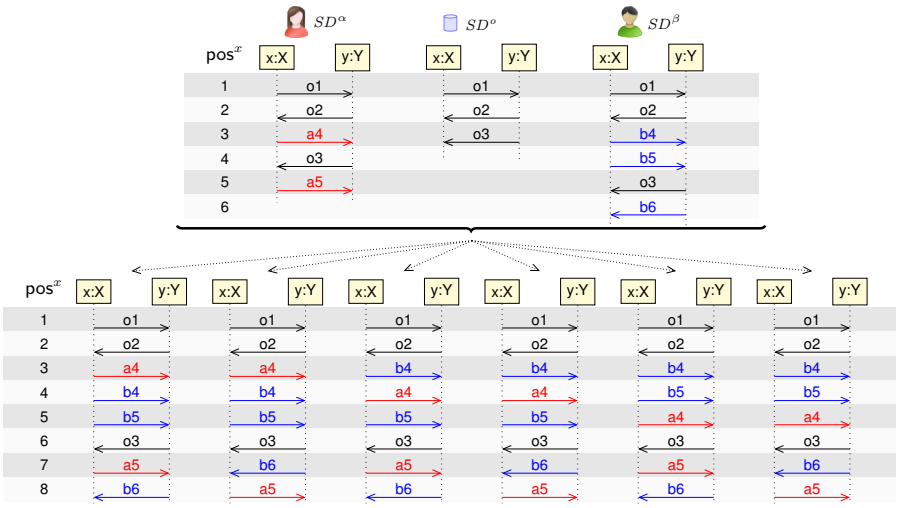

Fig. 5: Sequence diagram (SD^{o}) and two revisions (SD^{α} and SD^{β}) with its six time consistent, but not necessarily lifeline-conformant, merges (below), and the values of the respective pos^{x} function (left column).

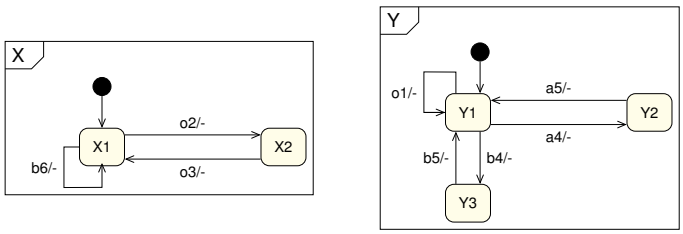

Fig. 6: The state machines modeling the behavior of the lifelines in Fig. 5

occurs as path in state machine Y and so does the sequence of lifeline x $(\mathsf{o2}, \mathsf{o3}, \mathsf{b6})$, but in the leftmost diagram, for sequence $(\mathsf{o1}, \mathsf{a4}, \mathsf{b4}, \mathsf{b5}, \mathsf{a5})$ of lifeline y this is not the case.

Definition 11 (Merging Problem). *Given a triple* $(SD^{o}, SD^{\alpha}, SD^{\beta})$ *where* $SD^{o} = (L^{o}, M^{o}, \mathsf{lprop}^{o}, \mathsf{msg}^{o})$, $SD^{\alpha} = (L^{\alpha}, M^{\alpha}, \mathsf{lprop}^{\alpha}, \mathsf{msg}^{\alpha})$, *and* $SD^{\beta} = (L^{\beta}, M^{\beta}, \mathsf{lprop}^{\beta}, \mathsf{msg}^{\beta})$ *are valid sequence diagrams, and* SD^{α} *and* SD^{β} *are revisions of* SD^{o}*, the merging problem is to find a consolidated version of* SD^{o}, SD^{α}, SD^{β}.

6 Encoding to SAT

We propose to translate the sequence diagram merging problem to a satisfiability problem of propositional logic (SAT) [6]. Over the last years, propositional logic has proven to be a powerful host language for a wide range of real-life problems like verification and planning, not least because efficient and stable solvers for this problem, SAT solvers,

are available [20]. For our merging problem, we also take advantage of this technology which allows for a direct representation of the merging problem. Before we present the encoding of the merging problem in propositional logic, we revisit the necessary concepts of propositional logic.

6.1 Preliminaries

The language of propositional logic Prop is defined over a set of propositional variables V, the truth constants $\top$ and $\bot$, conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$) as follows. If $x \in \mathsf{V} \cup \{\bot, \top\}$, then $x \in \mathsf{Prop}$; if $\phi \in \mathsf{Prop}$, then $\neg\phi \in \mathsf{Prop}$; if $\psi_1, \psi_2 \in \mathsf{Prop}$, then $\psi_1 \circ \psi_2 \in \mathsf{Prop}$ with $\circ \in \{\vee, \wedge\}$.

Most SAT solvers process only formulas of Prop, that are given in *conjunctive normal form (CNF)*, which imposes some syntactic restrictions on the formula structure. A formula is in CNF, if it is a conjunction of clauses. A clause is a disjunction of literals. A literal is a variable or its negation. For example the formula $(x \vee \neg y) \wedge (\neg x \vee y)$ is in CNF, whereas the equivalent formula $(\neg x \wedge \neg y) \vee (x \wedge y)$ is not. The usage of CNF is advocated, because of its simpler data structure and the support of specific, very efficient reasoning techniques. It can be shown that each non-CNF can be transformed to a formula in CNF preserving satisfiability [22] with linear overhead.

The semantics of propositional logic is defined over variable assignments. A variable assignment for a formula ϕ is a set $\mathcal{I} \subseteq \mathsf{vars}(\phi)$, where $\mathsf{vars}(\phi)$ denotes the variables occurring in ϕ. The truth value $v_{\mathcal{I}}(\phi)$ of formula ϕ under a variable assignment $\mathcal{I}$is defined as follows.

- If $\phi = \top$ (resp. $\phi = \bot$), then $v_{\mathcal{I}}(\phi) = 1$ (resp. $v_{\mathcal{I}}(\phi) = 0$);
- if $\phi = x$ with $x \in \mathsf{Prop}$, then $v_{\mathcal{I}}(\phi) = 1$ if $x \in \mathcal{I}$, else $v_{\mathcal{I}}(\phi) = 0$;
- if $\phi = \neg\psi$, then $v_{\mathcal{I}}(\phi) = 1 - v_{\mathcal{I}}(\psi)$;
- if $\phi = \psi_1 \wedge \psi_2$, then $v_{\mathcal{I}}(\phi) = \min(v_{\mathcal{I}}(\psi_1), v_{\mathcal{I}}(\psi_2))$;
- if $\phi = \psi_1 \vee \psi_2$, then $v_{\mathcal{I}}(\phi) = \max(v_{\mathcal{I}}(\psi_), v_{\mathcal{I}}(\psi_2))$.

A formula ϕ is called *satisfiable*, if there exists a variable assignment $\mathcal{I}$ such that $v_{\mathcal{I}}(\phi) = 1$, otherwise ϕ is called *unsatisfiable*. A variable assignment which satisfies a formula is called *model* of the formula. In the following, we encode merging of two revisions of a sequence diagram in a way that a SAT solver returns "unsatisfiable" if no valid merge is possible and "satisfiable" otherwise. In the latter case, also a model of the formula is returned which represents a solution of the merge problem, i.e. a correctly merged sequence diagram.

6.2 Merging Sequence Diagrams

Given an instance $(SD^o, SD^\alpha, SD^\beta)$ of the sequence diagram merging problem, with $SD^x = (L^x, M^x, \mathsf{lprop}^x, \mathsf{msg}^x)$ for $x \in \{o, \alpha, \beta\}$, defined over a set $\mathcal{SM}$ of state machines, let

- $S_{all} = \bigcup_{SM \in \mathcal{SM}} \pi_1(SM)$ be the set of all states of all state machines,
- $M = M^o \cup M^\alpha \cup M^\beta$ be the set of all messages, and

– $k = |M|$ the total number of messages.

Then the non-CNF formula ϕ is built over the following sets of variables:

– $\mathsf{vm} = \{m_i \mid m \in M \wedge i \in \mathsf{allow}(m)\}$. Variables of this set encode the placement of each message at each of its allowed positions. If m_i evaluates to true, it means that message m is placed at position i.
– $\mathsf{vc} = \{c_i^s \mid 1 \leq i \leq k, s \in S_{all})\}$. Variables of this set encode the source state of a state machine for each position before a message is received. If c_i^s evaluates to true, it means that at position i, before the message placed on i is received, the state machine containing s is in state s, or in other words, s is the source state of the transition triggered by the action symbol of the message placed on i.
– $\mathsf{vt} = \{t_i^s \mid 1 \leq i \leq k, s \in S_{all}\}$. Variables of this set encode the target state of a state machine for each position after a message has been received. If t_i^s evaluates to true, it means that at position i, after the message placed on i has been received, the state machine containing s is in state s, or in other words, s is the target state of the transition triggered by the action symbol of the message placed on i.

According to Definition 10, a consolidated version $\S^\gamma$ of SD^o, SD^α and SD^β has to meet the following requirements:

1. $L^\gamma = L^\alpha \cup L^\beta$,
2. $M^\gamma = M^\alpha \cup M^\beta$,
3. for each $i \in \{\alpha, \beta, o\}$ and for each $m \in M^i$ it holds that $\mathsf{msg}^\gamma(m) = \mathsf{msg}^i(m)$,
4. for each $i \in \{\alpha, \beta, o\}$ and for each $l \in L^i$ it holds that $\mathsf{lprop}^\gamma(l) = \mathsf{lprop}^i(l)$,
5. for each $m \in M^\gamma$ it holds that $\mathsf{pos}(m) \in \mathsf{allow}(m)$, and
6. S^γ is correct: S^γ is well-formed, time consistent and all lifelines are conformant to their state machines.

The first set of subformulas encodes that for each $m \in M^\gamma$ it holds that $\mathsf{pos}(m) \in \mathsf{allow}(m)$ (point 5 of Definition 10), that each message is placed on a position (point 2 of Definition 10, and that on each position exactly one message is placed (well-formedness of point 6 of Definition 10).

$$\bigwedge_{m \in M} \left(\bigvee_{i \in \mathsf{allow}(m)} m_i \right) \wedge \bigwedge_{m \in M} \bigwedge_{\substack{i,j \in \mathsf{allow}(m) \\ i \neq j}} \left(\neg m_i \vee \neg m_j \right)$$

The next subformula encodes the requirement that S^γ is time consistent (Point 6 of Definition 10).

– SD^γ is time consistent. This requirement is encoded as follows.

$$\bigwedge_{x \in \{o,\alpha,\beta\}} \bigwedge_{m \in M^x} \bigwedge_{i \in \mathsf{allow}(m)} \left(\neg m_i \vee \bigvee_{\substack{n \in M^x, \\ n \succ m}} \bigvee_{\substack{j > i, \\ j \in \mathsf{allow}(n)}} n_j \right)$$

– Each lifeline of SD^γ must be conformant to its state machine. For $m \in M$ and $\mathsf{msg}(m) = (a, s, r)$, let

- $\mathsf{lprop}(\mathsf{life}(r)) = (SM, E^{snd}, E^{rcv}, >)$, i.e. the associations of the lifeline that receives m,
- $SM = (S, A^{tr}, A^{\mathit{eff}}, T)$ the state machine describing the behaviour of the lifeline,
- $S_m^{tr,c} = \{s \in S \mid (s, a', A, s') \in T, a = a'\}$ i.e. all source states of transitions triggered by action a of message m,
- $S_m^{tr,t} = \{s' \in S \mid (s, a', A, s') \in T, a = a'\}$ i.e. all target states of transitions triggered by action a of message m.
- $S_m^{\mathit{eff},c} = \{s \in S \mid (s, a', A, s') \in T, a \in A\}$ i.e. all source states of transitions executing action a of message m,
- $S_m^{\mathit{eff},t} = \{s' \in S \mid (s, a', A, s') \in T, a \in A\}$ i.e. all target states of transitions executing action a of message m.

Then the encoding of this constraint is the conjunction of the following three subformulas:

1. For each message m, the information about the states in which a transition can be triggered by the action of m and to which states the triggered transitions lead:

$$\bigwedge_{m \in M} \bigwedge_{i \in \mathsf{allow}(m)} \left(\left(\neg m_i \vee \bigvee_{s \in S_m^{tr,c}} c_i^s \right) \wedge \left(\neg m_i \vee \bigvee_{s \in S_m^{tr,t}} t_i^s \right) \right)$$

2. Before and after an action is received, some state machine must be in one of its states:

$$\bigwedge_{i=1}^{k} \left(\left(\bigvee_{c_i^s \in \mathsf{vc}} c_i^s \right) \vee \left(\bigvee_{t_i^s \in \mathsf{vt}} t_i^s \right) \right)$$

3. If a state machine stops in state s at position i, then, when it eventually continues at position $i + l$, it must still be in state s. Other state machines may be placed at positions $i + j$, $j < k$. We abbreviate the set of states $\pi_1(M)$ for a state machine M by M_S. This is the only non-CNF part of the encoding.

$$\bigwedge_{i=1}^{k-1} \bigwedge_{M \in \mathcal{SM}} \bigwedge_{s \in M_S} \left(\left(t_i^s \rightarrow \bigwedge_{r \in M_S \setminus s} \neg c_{i+1}^r \right) \wedge \left(\bigwedge_{j=1}^{i} \left(t_i^s \wedge \bigwedge_{l=1}^{j} \neg c_l^s \rightarrow \bigwedge_{r \in M_S \setminus s} \neg c_{j+1}^r \right) \right) \right)$$

In this section, we clearly see the benefit of the extensive specification of the sequence diagram merge problem of the previous section. On the basis of this specification, the encoding to SAT is straight-forward and relies only on standard techniques of modeling with propositional logic. In order to get an implementation of a sequence diagram merging tool, only the mapping to this SAT encoding based on the involved *tMVML* models has to be realized. The actual problem solving is completely handed over to a SAT solver.

7 Case Study

The concise problem description given in Section 5 allows us to encode the merging problem of sequence diagrams as satisfiability problem of propositional logic (SAT). Since the SAT represenation of such an encoding can become very large, tool support for the automatic generation is required. We implemented a first prototype described in the following. This prototype allowed us to conduct a first case study on a representative test set.

7.1 Implementation

We implemented the presented approach in a prototype available at `http://www.modelevolution.org/prototypes/sdmerge`. Our prototype consists of six modules: The difference provider, the SAT encoder, the Tseitin transformer, the SAT solver, the model merger, and the a model verifier. All modules, except for the SAT solver are written by us and in Java. To solve SAT instances, we use the off-the-shelve SAT solver PICOSAT-936 [5]. The *difference provider* and the *model merger* are based on the Eclipse Modeling Framework (EMF)[6].

From the provided original and revised sequence diagrams along with their state machines, the SAT encoding is generated as described in Section 6. All models, sequence diagrams as well as state machines, are expressed in Ecore and respect the metamodel of *tMVML*. As PICOSAT-936 requires the input to be in CNF, we convert our encoding to using the Tseitin transformation [22]. The code is than handed to PICOSAT-936 which returns either *false*or a model for the formula. If a model is returned (i.e. the formula is *true*), we negate the model, add it to the encoding and hand it again to PICOSAT-936. We repeat this until PICOSAT-936 returns *false*. Each of the returned models corresponds to a consolidated version of our problem instance.

Finally, each model is translated back into Ecore and checked for its correctness by the *model verifier*.

7.2 Evaluation

In order to study the impact of using the information provided by state machines to guide the merging of two differently evolved sequence diagrams, we established a representative benchmark set. Available benchmark sets as presented in [10], contain only modeling scenarios of one single view and focus mainly on class diagrams. Since versioning systems do not store the two revisions explicitly, but only the merged versions, suitable test cases cannot be extracted from available projects.

Although the models of our benchmark set are formulated in *tMVML*, they might be reused in other case studies, because they are realized as Ecore models. Hence, a translation to other modeling languages like full UML can be achieved by the means of model transformations. The benchmark set is available at our project website.

The benchmark set consists of three different families, each containing five different versioning scenarios. The first family on a subset of the SMTP protocol is based on

[6] http://www.eclipse.org/modeling/emf/

Set	# SM	# action symbols	# states	# transitions
email	3	15	16	19
coffee	2	9	7	8
philosopher	2	8	7	8

Table 2: Statistics on state machines of benchmark sets.

the state machines presented in Section 2. The second family models the behavior of a coffee machine and its users similar to the example presented in [8]. Finally, in the third family we model the behavior of the famous dining philosophers problem, inspired by the running example of [25]. For each versioning scenario, we distinguish between three different cases: (1) All lifelines are fully specified by state machines, (2) some lifelines are specified by state machines, and (3) no lifeline is specified by a state machine.

If no state machine is specified for a lifeline, we assume an unconstrained state machine, which contains only one state, from which any action symbol can be received.

With this setup, we tested our approach with 45 test cases. Details of the different test cases are shown in Table 2. The evaluation allows us to test correctness of the merged version and get a first impression on scalability issues.

Table 3 shows statistics on the number of found solutions and runtimes of the different instances. The leftmost columns describe the names, number of lifelines (LL), and number of messages (Ms) of each instance, the three rightmost columns show the number of found solutions (#Sol) and runtime (Time) of the instance. Those instances, for which the number of solution exceeded 1,000 or the runtime exceeded 100s, we stopped the algorithm, as having too many solutions is impractical. The instances *philosopher 4* and *philosopher 5* are test cases for the verifier and contain incorrect models, hence no values are shown in the table.

For the instances where no state machines are defined, we can compute the number of models with $\prod_{f \in F} \frac{(n_f+m_f)!}{n_f!m_f!}$ where F is the set of fragments of the merged model. Each fragment contains a set of messages inserted between two messages of the original diagram or at the beginning or the end, n_f is the number of messages in fragment f inserted from one revision and m_f the number of messages inserted from the other.

The evaluation shows that except for *philosopher 3*, cases where all state machines are specified result in few solutions that are found quickly. With no specified state machines, the exponential growth of the number of solutions can be seen particularly in the instances of *mail*.

8 Conclusion and Future Work

In this paper, we demonstrated how information encoded in the state machine view of a software model may be used to guide the merging of concurrently evolved versions of a sequence diagram. Such merging support is urgently needed to realize optimistic model versioning systems.

We illustrated our approach for the modeling language *tMVML*, which borrows many concepts from UML. For *tMVML*, we specified a metamodel in Ecore, which provided us with the powerful tool support of the Eclipse environment to build a prototype of our approach. In order to give a formal specification of the merging problem itself, we first formalized the concepts of *tMVML* along with some important properties of the sequence

Set	ID	SD^o		SD^α		SD^β		full SM		some SM		no SM	
		LL	Ms	LL	Ms	LL	Ms	# Sol	Time	# Sol	Time	# Sol	Time
email	1	3	5	3	7	3	12	1	2.0	1	1.0	55	2.8
	2	3	5	3	8	3	15	0	1.7	2	1.2	110	7.2
	3	3	5	3	14	3	14	2	3.6	2	1.5	>48,620	>10
	4	3	5	4	14	3	16	2	3	2	2	167.690	>10
	5	3	5	4	14	3	18	2	4.6	2	2.5	$\sim 10^{36}$	>10
coffee	1	2	5	2	6	2	9	2	0.3	2	0.3	5	0.5
	2	2	5	2	6	2	6	0	0.1	0	0.1	2	0.1
	3	2	0	2	2	2	2	2	0.7	6	0.4	6	0.3
	4	2	5	2	9	2	9	2	0.4	70	3.7	70	3.2
	5	2	5	2	9	2	9	34	2.1	34	1.9	70	3.2
philosopher	1	4	0	4	2	4	5	6	0.5	15	1.0	15	0.8
	2	4	0	4	1	4	5	0	0.1	5	0.5	5	0.30
	3	4	0	4	9	4	9	506	83	$> 1,000$	> 100s	48,010	> 100s
	4	4	0	4	9	4	5	-	-	-	-	-	-
	5	4	0	4	9	4	5	-	-	-	-	-	-

Table 3: Overview on benchmarks.

diagram. On this basis, we derived an exact problem specification which can be directly encoded as a satisfiability problem of propositional logic, the prototypical problem for NP. To solve such a problem, highly optimized solving tools, SAT solvers, are available. This way, instead of implementing a complicated merging algorithm, our tool encodes the constraints of a merging problem into a propositional formula and the computation of a set of consolidated versions, that are correct sequence diagrams merged from two different versions, is done by the SAT solver. From this set of sequence diagrams the software modeler can select a convenient version. By this means, user effort is reduced and merging errors are avoided.

As we showed in our experiments, usually there are many solutions representing valid merges. It is subject to future work to develop ranking and filtering techniques to offer helpful pre-selections. Further, it should be possible that the modeler specifies additional constraints for the merged model in order to cut down the number of solutions. Then extensive user experiments have to be conducted.

So far we represent the models only in abstract syntax. However, in an ongoing project we are currently developing dedicated visualization techniques for sequence diagram merging. First mockups are available at our project website.

We aim to extend *tMVML* and plan to consider more concepts like hierarchical states for state machines or combined fragments for sequence diagrams. Also for the prototype we allowed only additions as changes. In future work, we plan to include deletions and updates. Overall, we realized a powerful approach for the merging of sequence diagrams taking the information of the state machine view into account to support the evolution of the sequence diagram view.

References

1. K. Altmanninger, P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. Why Model Versioning Research is Needed!? In *Proc. MoDSE-MCCM Workshop*, 2009.

2. S. Barrett, P. Chalin, and G. Butler. Model Merging Falls Short of Software Engineering Needs. In *Proc. of the 2nd MoDSE Workshop @ MoDELS'08*, 2008.
3. L. Bendix and P. Emanuelsson. Requirements for Practical Model Merge - An Industrial Perspective. In *Proc. of the 12th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2009)*, volume 5795 of *LNCS*, pages 167–180. Springer, 2009.
4. J. Bézivin. On the Unification Power of Models. *SoSyM*, 4(2):171–188, 2005.
5. A. Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.
6. A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Sat.* IOS Press, 2009.
7. F. P. Brooks, Jr. No Silver Bullet—Essence and Accidents of Soft. Eng. *Comp.*, 20(4), 1987.
8. P. Brosch, U. Egly, S. Gabmeyer, G. Kappel, M. Seidl, H. Tompits, M. Widl, and M. Wimmer. Towards Scenario-Based Testing of UML Diagrams. In *Proc. of the 6th Int. Conf. on Tests and Proofs (TAP 2012)*, volume 7305 of *LNCS*, pages 149–155. Springer, 2012.
9. P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. The Past, Present, and Future of Model Versioning. In *Emerging Technologies for the Evolution and Maintenance of Software Models*, chapter 15, pages 410–443. IGI Global, 2011.
10. P. Brosch, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. Colex: A Web-based Collaborative Conflict Lexicon. In *Proc. Workshop on Model Comp. in Pract.* ACM, 2010.
11. A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing Model Conflicts in Distributed Development. In *Proc. MoDELS'08*, volume 5301 of *LNCS*, pages 311–325. Springer, 2008.
12. R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232–282, 1998.
13. Z. Diskin, Y. Xiong, and K. Czarnecki. Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In *Models in SE*, volume 6627 of *LNCS*. Springer, 2011.
14. A. Egyed. Instant Consistency Checking for the UML. In *Proc. of the 28th Int. Conf. on Software Engineering (ICSE 2006)*, pages 381–390. ACM, 2006.
15. R. B. France, A. Evans, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. *Computer Standards & Interfaces*, 19(7):325–334, 1998.
16. C. Gerth, J. Küster, M. Luckey, and G. Engels. Detection and Resolution of Conflicting Change Operations in Version Management of Process Models. *SoSym*, pages 1–19, 2011.
17. S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. In *ICSE 2007*, 2007.
18. OMG. Unified Modeling Language (OMG UML), Superstructure V2.4.1. `http://www.omg.org/spec/UML/2.4.1/`, August 2011.
19. D. Parnas. Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs. In *Prog. Meth.*, volume 23 of *LNCS*, pages 225–235. Springer, 1975.
20. M. R. Prasad, A. Biere, and A. Gupta. A Survey of Recent Advances in SAT-based Formal Verification. *STTT*, 7(2):156–173, 2005.
21. M. Sabetzadeh, S. Nejati, S. Liaskos, S. M. Easterbrook, and M. Chechik. Consistency checking of conceptual models via model merging. In *Proc. RE 2007*. IEEE, 2007.
22. G. Tseitin. On the Complexity of Derivation in Propositional Calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 2(115-125):10–13, 1968.
23. A. Tsiolakis. Integrating Model Information in UML Sequence Diagrams. In *Proc. of the Satellite Workshops of the 28th ICALP*, volume 50 of *Electronic Notes in Theoretical Computer Science*, pages 268–276. Elsevier Science Publishers, 2001.
24. R. Van Der Straeten, J. Pinna Puissant, and T. Mens. Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In *Modelling Foundations and Applications*, volume 6698 of *LNCS*, pages 69–84. Springer, 2011.
25. D. Varró. Automated Formal Verification of Visual Modeling Languages by Model Checking. *SoSyM*, 3(2):85–113, 2004.
26. B. Westfechtel. A Formal Approach to Three-way Merging of EMF Models. In *Proc. of the 1st Int. Workshop on Model Comparison in Practice @ TOOLS'10*, pages 31–41. ACM, 2010.