A SAT-Based Debugging Tool for State Machines and Sequence Diagrams*

Petra Kaufmann[1], Martin Kronegger[2], Andreas Pfandler[2], Martina Seidl[1,3], and Magdalena Widl[4]

[1] Business Informatics Group, TU Wien, Austria
[2] Database and Artificial Intelligence Group, TU Wien, Austria
[3] Institute for Formal Models and Verification, JKU Linz, Austria
[4] Knowledge-Based Systems Group, TU Wien, Austria
{firstname.lastname@tuwien.ac.at}

Abstract. An effective way to model message exchange in complex settings is to use UML sequence diagrams in combination with state machine diagrams. A natural question that arises in this context is whether these two views are consistent, i.e., whether a desired or forbidden scenario modeled in the sequence diagram can be or cannot be executed by the state machines. In case of an inconsistency, a concrete communication trace of the state machines can give valuable information for debugging purposes on the model level. This trace either hints to a message in the sequence diagram where the communication between the state machines fails, or describes a concrete forbidden communication trace between the state machines. To detect and explain such inconsistencies, we propose a novel SAT-based formalization which can be solved automatically by an off-the-shelf SAT solver. To this end, we present the formal and technical foundations needed for the SAT-encoding, and an implementation inside the Eclipse Modeling Framework (EMF). We evaluate the effectiveness of our approach using grammar-based fuzzing.

1 Introduction

The abstraction power of multi-view modeling languages like UML comes along with the possibility of inconsistencies in the description of the system under development [18]. On the one hand, different diagram types lower the complexity of describing and understanding large software systems by providing focused views on specific aspects like, for example, interprocess communication [2]. On the other hand, performing modifications on one diagram may require changes in other diagrams. If these changes are not implemented carefully in the other diagrams as well, the model can contain inconsistent information which, in the worst case, might propagate up to the running application. Hence, if the diagrams do not complement one another in a consistent manner, then the benefits of

* This work was partially funded by the Vienna Science and Technology Fund (WWTF) under grant ICT10-018 and by the Austrian Science Fund (FWF) under grants P25518-N23, S11408-N23, and S11409-N23.

B. Combemale et al. (Eds.): SLE 2014, LNCS 8706, pp. 21–40, 2014.

multi-view modeling are rendered void [23]. Especially when the models are not directly executable or when no simulation environment is available, then testing and debugging is difficult.

Therefore, mechanisms are required which support the *evolution* of a model [10] and ensure consistency. In this paper, we are concerned with the consistency between state machines and sequence diagrams. State machines describe the internal behavior of objects and sequence diagrams focus on interaction scenarios between different instances of the objects. These scenarios model either required or forbidden message exchange. Our approach verifies whether the communication described by a sequence diagram can be executed by a given set of state machines. If a sequence of messages can be executed although it is forbidden by the sequence diagram, then a concrete communication trace is returned. If a sequence of messages is not possible although according to the sequence diagram it should be, then a reason for the failure is given. On this basis, inconsistencies introduced during the evolution of a model cannot only be discovered easily, but also be corrected immediately. Hence, sequence diagrams are test cases describing desired or undesired behavior of the state machines. With our approach the test cases can be evaluated even if no execution environment for the state machines is available.

For solving this consistency checking problem, we propose to use an approach based on the satisfiability problem of propositional logic (SAT) [3]. For SAT powerful solvers are available which are successfully used out of the box in many verification applications. For instance, we have made very positive experiences with using SAT encodings to solve the merging problem in the context of optimistic model versioning [30] as well as for reachability checking of composite state machines [14]. Based on these experiences, we developed the consistency checking encoding presented in this paper. This considerably improves our previous work on consistency checking using the model checker Spin [4,5]. Spin offers the high level input language Promela which seems to be very appealing for formulating the consistency checking problem. However, due to the semantic differences of Promela and UML-like languages the encoding becomes rather complicated. In SAT, however, we do not have any semantical restrictions. With a concise problem formulation together with encoding techniques borrowed from planning applications [21] the SAT encoding turns out intuitive and flexible. Further, the SAT-based approach integrates smoothly into our model evolution framework FAME[1].

This paper is structured as follows. First, we review related approaches in Section 2. Then we motivate this work with a concrete example in Section 3 and informally explain the modeling language concepts relevant for this work. In Section 4 we give a concise formal problem definition. To this end, we formally describe the sequence diagram and the state machine along with their interplay. Further, we introduce the notion of lifeline consistency, which is what we want to check. This problem definition directly allows us to derive a problem encoding to SAT which can be handed to a SAT solver (Section 5). Section 6 discusses

[1] `http://www.modelevolution.org/`

the implementation based on the Eclipse Modeling Framework and Section 7 presents a detailed evaluation of our approach based on grammar-based white-box fuzzing. Finally, we conclude this paper with an outlook on future work.

2 Related Work

The problem tackled in this paper is a typical model checking problem. Therefore, it is not surprising that different works [5,13,15,20,22] propose a formulation in languages like Promela, the input language of the popular model checker Spin. Due to semantical differences of state machines and Promela, it turns out that an equivalence preserving translation capturing all language concepts is challenging. For example, in our previous work [4,5], we employed Spin to ensure that given traces do not occur during the execution of a set of state machines, but with this encoding we could not ensure that a given message sequence is possible.

In the past, many other formal approaches have been presented, but most of the implementations do not seem to have gone beyond a proof of concept state and are either not updated to UML 2 or are not available at all. We summarize the approaches most related to our work in the following. For a detailed discussion we refer the interested reader to specific surveys like [18]. Lam and Vitus [16] present an algebraic approach to express the consistency checking problem in the π-calculus. The practical realizability of the approach is not discussed. Van der Straeten et al. [27] propose to use description logics to formally describe the consistency between class diagrams, sequence diagrams, and state machines. Compared to SAT, description logics are more expressive in general, but their satisfiability checking problem is located in higher complexity classes than NP. Bernardi et al. propose to use petri nets for checking the consistency between different diagrams [1]. Communication, however, is only considered at the class level and not at the object level. Engels et al. [8] propose to check consistency by evaluating dedicated consistency constraints represented in form of collaborations. Therefore, an interpreter is provided. Egyed [7] applies instant consistency validation by rules formulated in OCL which shows to be very efficient on large models. For capturing the same kind of inconsistency, which we deal with in this paper, however, a temporal extension of OCL is necessary.

A different approach for consistency checking is presented by Graaf and Van Deursen [12] who suggest to synthesize a state machine from the given sequence diagram as in [28] and then compare the automatically generated state machine to the given state machine. Therefore, they realize normalization, transformation, and comparison steps, respectively. In [12], however, the comparison requires manual intervention. Feng and Vangheluwe propose to use a simulation-based approach for consistency checking [9].

Besides checking the consistency between state machine diagrams and sequence diagrams, a lot of effort has been spent for consistency checking between other diagrams like class diagrams, collaboration diagrams, activity diagrams, etc. We refer to [11,26] for detailed surveys.

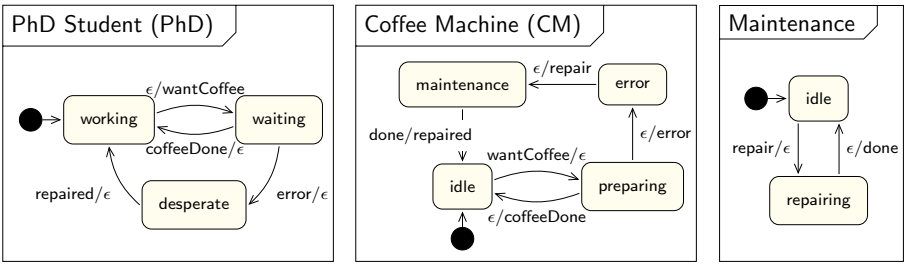

Fig. 1. Three state machines modeling a PhD student, a coffee machine, and a maintenance unit

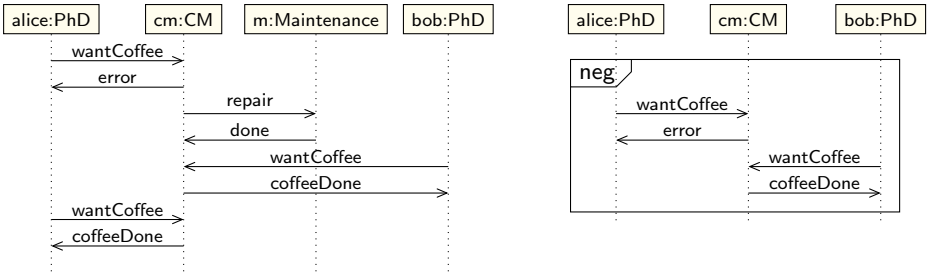

Fig. 2. (Left) A sequence diagram depicting a desired scenario that is inconsistent with the state machines of Fig. 1. The state machines have to be changed in order to allow the scenario. (Right) A sequence diagram depicting a forbidden scenarios that is inconsistent with the state machines of Fig. 1. No changes are required.

3 A Motivating Example

To motivate our work and to illustrate useful application scenarios we present the following example. Fig. 1 shows three state machines that describe the behaviors of a PhD student, a coffee machine, and a maintenance unit for the coffee machine. As typical for the UML state machine view, rectangles with rounded corners present *states* which are connected by *transitions*. Each transition carries a label consisting of a *trigger* on the left side of the "/" and an *effect* on the right side. The special symbol ϵ on the left side of the "/" indicates that no trigger is necessary for the transition to fire. The initial state is indicated by an incoming arc from a black dot.

Instances of state machines communicate with each other by message passing. They change states according to messages that are sent and received. A state change is initiated by the receipt of a symbol indicated as trigger in one of the outgoing transitions of the current state. An outgoing transition carrying the special symbol ϵ as trigger can be initiated without receiving any symbol.

The transition is fully executed only if the effect can be sent successfully, i.e., if it also can be received by another instance of a state machine.

Fig. 2 shows two sequence diagrams that describe communication scenarios between instances of the state machines in Fig. 1. A state machine is instantiated by one or more *lifelines*. Similar as in UML, they are shown as rectangles with a dashed vertical line underneath. Each lifeline's name is shown inside the rectangle before the ":", followed by the name of the state machine it instantiates after the ":". For space reasons, we have abbreviated these names. Along the lifelines, a sequence of messages is shown. A message is depicted as an arrow from the sender lifeline to the receiver lifeline labeled with the symbol being sent. The set of symbols used in the sequence diagrams is the same as the set of symbols used in the state machines.

In order to be *consistent* with the state machines, the message sequence of a sequence diagram must be executable from some global state of the lifelines which is reachable from the global initial state, where a global state is a tuple of states of the state machines instantiated by the lifelines. More precisely, from such a global state it must be possible for each message after another to be a trigger in the sending lifeline's state machine instantiation and to be an effect in the receiving lifeline's state machine instantiation.

We present two possible application scenarios for checking a set of state machine diagrams and a sequence diagram for consistency. (1) A desired scenario is depicted in the sequence diagram. If the sequence diagram is consistent with the state machines, then we know that the state machines fulfill the scenario. Otherwise, we can obtain information about the global state of the state machines where the sequence first fails, which helps to discover erroneous or missing transitions in the state machines view. (2) An unwanted scenario is depicted in the sequence diagram. If a sequence diagram is consistent with the state machines, then we know that there is a bug in the state machines and we can obtain a counter-trace, namely a sequence of global states which follows from the application of the message sequence.

In Fig. 2 an example for each scenario is depicted. The left sequence diagram shows a desired scenario. However, it is inconsistent with the state machines for the following reason: The PhD student "alice" changes into state "desperate" after receiving the symbol "error" from the coffee machine. She must remain there until the symbol "repaired" is received. According to the sequence diagram, the coffee machine never sends this symbol. This also means, that the coffee machine never returns to state "idle" and therefore cannot receive the symbol "wantCoffee" from PhD student "bob". Therefore, the message sequence of the sequence diagram can only be executed up to and including the fourth message, "done". In this case, our tool returns the sequence of messages up to the message that cannot be sent or received, in this case, up to and including "done" from "m:Maintenance" to "cm:CM". A possible fix for this broken scenario would be to remove the state "desperate" from the PhD student and to connect the transition with trigger "error" from the state "waiting" directly to state "working".

Further, in the coffee machine the effect of the transition with trigger "done" from state "maintenance" to state "idle" would have to be replaced by ϵ.

Similarly to the "neg" fragment used in UML sequence diagrams, we mark the negative scenario in the right diagram of Fig. 2 using this notation. Note that we refer to a complete application scenario rather than to a subsequence of a sequence diagram. Hence, the second diagram shows an unwanted scenario. It allows the coffee machine to prepare coffee while being in the error state. This scenario is not implemented in the state machines, so no bug can be found. If it was implemented, the tool would return a sequence of global states of the instances of the state machines representing this message sequence.

4 Problem Definition

Given a sequence diagram and a set of communicating state machines modeling the behavior of the sequence diagram's lifelines, the *Multiview Sequence Consistency Problem* (MSCP) asks whether the communication sequence modeled in the sequence diagram is executable by the state machines. If this is the case, then we call the two views *consistent*. The desired outcome of a positive scenario (no "neg" label) depicted in a sequence diagram is to be consistent with the state machine view, i.e., the desired scenario is indeed implemented in the state machines. The desired outcome of a negative scenario ("neg" label) is to be inconsistent with the state machine view, which means, that the state machines do not implement the undesired trace. In the following, we present a precise definition of the semantics of the state machine view and of the sequence view in order to present the formal definition of MSCP.

The core elements for defining state machines, sequence diagrams, and their interaction are the symbols of the alphabets Σ_A and Σ_L where the special symbol ϵ is in Σ_A. The alphabet Σ_A contains symbols which label messages in the sequence diagrams and which trigger transitions and occur as effects in the state machines. The special symbol ϵ is the "empty symbol" used for transitions triggered by on-completion-events and for empty events on transitions. The alphabet Σ_L contains names for the instances of the state machines, also called lifelines. Based on Σ_A we define state machines as follows.

Definition 1 (State Machine). *Given an alphabet Σ_A, a* state machine *M is a quadruple (S, ι, A, T), where*

- *S is a finite set of* states*,*
- *$\iota \in S$ is a designated* initial state*,*
- *$A \subseteq \Sigma_A$ with $\epsilon \in A$ is the alphabet of M, and*
- *$T \subseteq S \times A \times A \times S$ is a transition relation such that for all $s, s' \in S$ there is no transition $(s, \epsilon, \epsilon, s') \in T$.*

A state machine consists of a set of states, a designated initial state, an alphabet, and a transition relation which connects the states. The rightmost state machine shown in Fig. 1 which is called Maintenance, contains the set

$S = \{\mathsf{idle}, \mathsf{repairing}\}$ of states, the initial state $\iota = \mathsf{idle}$, and the alphabet $A = \{\epsilon, \mathsf{repair}, \mathsf{done}\}$. For a transition $t \in T$ with $t = (s, tr, \mathit{eff}, s')$, s is the source state of the transition, s' is the target state, tr is a symbol (trigger) which upon receipt triggers the execution of transition t, and *eff* is a symbol (effect) that is sent and has to be received by another state machine when the transition is executed. The state machine Maintenance in Fig. 1 has two transitions: $(\mathsf{idle}, \mathsf{repair}, \epsilon, \mathsf{repairing}), (\mathsf{repairing}, \epsilon, \mathsf{done}, \mathsf{idle}) \in T$.

For a transition to be executed in a state machine M, the trigger symbol of the transition must be received by M from a state machine different to M and the effect symbol must be received by a state machine different to M. Either trigger or effect can be the special symbol ϵ which stands for an empty trigger or effect. A transition containing ϵ as trigger is triggered without receiving any symbol, e.g., by an on-completion-event, and the execution of a transition containing ϵ as effect can be finished without sending any symbol. We assume that no transition of a state machine contains ϵ as both trigger and effect. Such transitions can be eliminated by contracting the connected states. Furthermore notice that the requirement of having a single effect does not impose a strong restriction as multiple effects can be simulated by a state machine that sends a predefined sequence of effects upon receiving a designated trigger symbol.

In order to give a precise semantics to the interaction between state machines, we introduce the notion of an *extended* state machine.

Definition 2 (Extended State Machine). *Given a state machine M, the* extended state machine M^* *of $M = (S, \iota, A, T)$ is a quadruple $(S \cup S^*, \iota, A, T^*)$ where*

- $S^* = \{s_t^* \mid t \in T\}$ *and*
- $T^* = \{(s, tr, \epsilon, s_t^*), (s_t^*, \epsilon, \mathit{eff}, s') \mid t = (s, tr, \mathit{eff}, s') \in T\}$

An extended state machine introduces an *intermediate state* s_t^* for each transition t. This intermediate state has exactly one incoming transition, which is triggered by the trigger of t and contains the effect ϵ, i.e., has no effect. It also has exactly one outgoing transition, which leads to the target state of t with ϵ as trigger and the effect of t. We call S the *original states* and S^* the *intermediate states*.

The extended state machine helps to distinguish between the event of having received the trigger and the event of being able to send the effect. Note that any state machine can be translated to exactly one extended state machine and vice versa. Fig. 3 depicts the extended state machine of the state machine PhD Student. The intermediate states are represented by black diamonds with rounded corners.

Next we formally define sequence diagrams, starting with the concept of lifelines.

Definition 3 (Lifeline). *Given a set $\mathcal{M}$ of extended state machines and the alphabet Σ_L, a lifeline is a pair $L = (l, M^*)$ where $l \in \Sigma_L$ is the name of the lifeline and $M^* \in \mathcal{M}$ is associated with the lifeline.*

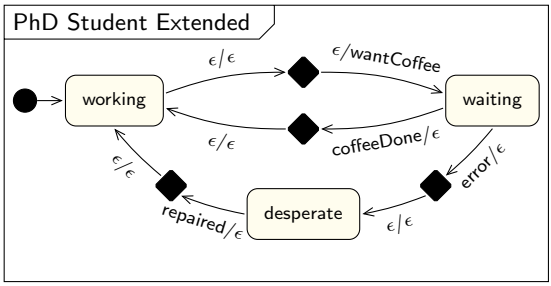

Fig. 3. Extended state machine corresponding to the state machine "PhD student"

A lifeline is an instance of an extended state machine. The name l of a lifeline is used to distinguish different instances of the same state machine and M^* is the extended state machine the lifelines refers to. In the sequel, we refer to M^* of a lifeline L by $\mathsf{sm}(L)$. The communication between lifelines takes place through *messages*, which are defined as follows.

Definition 4 (Message). *Given an alphabet Σ_A and a set $\mathcal{L}$ of lifelines such that each lifeline's extended state machine is defined over Σ_A, a* message *is a triple (σ, a, ρ) where*

- *$\sigma \in \mathcal{L} \cup \{\varepsilon\}$ is the sending lifeline,*
- *$a \in \Sigma_A$ is the message symbol, and*
- *$\rho \in \mathcal{L} \setminus \{\sigma\}$ is the receiving lifeline*

such that $\sigma = \varepsilon$ if and only if $a = \epsilon$.

For a message (σ, a, ρ), the sender lifeline σ either refers to an extended state machine or is the empty sender ε when the empty symbol ϵ is received. Note that for better readability, we do not show empty messages in the concrete syntax of the sequence diagrams. The receiver lifeline ρ refers to an extended state machine.

Based on the definition of a lifeline and of a message, we can now formally define a sequence diagram.

Definition 5 (Sequence Diagram). *Given the alphabets Σ_A and Σ_L, and a set $\mathcal{M}$ of extended state machines over Σ_A, a sequence diagram is a pair $(\mathcal{L}, \mu)$ where*

- *$\mathcal{L}$ is a set of lifelines over $\mathcal{M}$ and Σ_L*
- *the names of the lifelines are pairwise distinct*
- *$\mu = [m_1, \ldots, m_n]$ is a sequence of messages such that for each $(\sigma, a, \rho) \in \mu$ it holds that $\sigma, \rho \in \mathcal{L}$ and $a \in \Sigma_A$.*

The right-hand sequence diagram of Fig. 2 contains the set $\mathcal{L} = \{(\mathsf{alice}, \mathsf{PhD}), (\mathsf{cm}, \mathsf{CM}), (\mathsf{bob}, \mathsf{PhD})\}$ of lifelines and the sequence $\mu = [((\mathsf{alice}, \mathsf{PhD}), \mathsf{wantCoffee}, (\mathsf{cm}, \mathsf{CM})), \ldots, ((\mathsf{cm}, \mathsf{CM}), \mathsf{coffeeDone}, (\mathsf{bob}, \mathsf{PhD}))]$.

To describe the interaction between lifelines via messages we define a *global state* which captures a configuration of a set of lifelines.

Definition 6 (Global State). *Given a set* $\mathcal{L} = \{L_1, \ldots, L_l\}$ *of lifelines, let* $\mathsf{sm}(L_i) = (S_i, \iota_i, A_i, T_i)$ *be the extended state machine of lifeline* L_i*, for* $1 \leq i \leq l$*. Then a global state* $\hat{s}$ *is a tuple* $(s_1, \ldots, s_l) \in S_1 \times \cdots \times S_l$*.*

For three lifelines instantiating the three state machines of Fig. 1, an example for a global state is (desperate, <maintenance/done/repaired/idle>, idle) where the second state refers to the intermediate state on the transition from maintenance to idle in state machine CM.

In each global state, there exists a (possibly empty) set of messages that can be sent and a set of messages that can be received. After sending or receiving a message out of these sets, a different global state is reached. This semantics is described in the following definition.

Definition 7 (Admissibility and Application of a Message). *Given the alphabet* Σ_A*, a set* $\mathcal{L} = \{L_1, \ldots, L_l\}$ *of lifelines with* $\mathsf{sm}(L_i) = (S_i, \iota_i, A_i, T_i)$*, and a global state* $\hat{s} = (s_1, \ldots, s_l) \in S_1 \times \cdots \times S_l$*, the message* $m = (L_s, a, L_r)$ *with* $L_s \in \mathcal{L} \cup \{\varepsilon\}$*,* $L_r \in \mathcal{L}$*,* $L_s \neq L_r$*, and* $a \in \Sigma_A$ *is* admissible *in* $\hat{s}$ *if the following holds: If* $L_s \neq \varepsilon$*, then*

1. $(s_s, \epsilon, a, s'_s) \in T_s$*, and*
2. $(s_r, a, \epsilon, s'_r) \in T_r$*.*

Otherwise, i. e., if $m = (\varepsilon, \epsilon, L_r)$*, then* $(s_r, \epsilon, \epsilon, s'_r) \in T_r$*.*
By applying the admissible message m *in the global state* $\hat{s}$*, a* global successor state $\hat{s}' = (s_1, \ldots, s'_s, \ldots, s'_r \ldots, s_l) \in S_1 \times \cdots \times S_l$ *is reached.*

A message is *admissible* in some global state if (1) (for $L_s \neq \varepsilon$) the state of the sender lifeline is an intermediate state whose outgoing transition has the message symbol a as effect and (2) unless $L_s = \varepsilon$, the state of the receiver lifeline is an original state which has as least one outgoing transition with the message symbol a as trigger. If $L_s = \varepsilon$, the receiver can also be in an intermediate state.

In the global state $s =$ (desperate, <maintenance/done/repaired/idle>, idle) of the lifelines (alice, PhD), (cm, CM), (m, Maintenance) the set of applicable messages contains only one message, namely $\{m = ((\mathsf{cm}, \mathsf{CM}), \mathsf{repaired}, (\mathsf{alice}, \mathsf{PhD}))\}$.

Note that lifelines refer to extended state machines, which means that a transition cannot carry a trigger symbol other than ϵ together with an effect other than ϵ. Therefore, it can never happen that a receiver lifeline sends any effect while executing a transition triggered by a symbol other than ϵ.

The global successor state $\hat{s}'$ is reached by applying a message. Then, $\hat{s}'$ differs from $\hat{s}$ in the states of the sender and the receiver lifeline: The sender's state changes from an intermediate state to its only successor state, and the receiver's state changes accordingly to the received symbol into an intermediate state. Applying the above message m to the global state $\hat{s}$ reaches the global successor state (<desperate/repaired/ϵ/working>, idle, idle).

The set of admissible messages in a global state can contain a subset of messages that are *independent*, i.e., that have no sender or receiver in common. The messages in such a set can be executed simultaneously. We call a set of independent messages a *transaction*. It is defined as follows.

Definition 8 (Transaction). *Let $\mathcal{L} = \{L_1, \ldots, L_l\}$ be a set of lifelines. A* transaction *is a nonempty set $m = \{m_1, \ldots, m_t\}$ of messages such that for distinct $i, j \in \{1, \ldots, t\}$, $m_i = (\sigma_i, a_i, \rho_i)$, and $m_j = (\sigma_j, a_j, \rho_j)$ it holds that all σ_i, σ_j, ρ_i, and ρ_j are pairwise distinct.*

A transaction is admissible if all its messages are admissible. The global state reached by applying a transaction is the global state reached by applying each of the transaction's messages. Note that a sequence of messages can also be seen as a sequence of transactions that are singletons, i.e., each transaction contains a single message. A sequence of messages, such as depicted in a sequence diagram, can therefore be seen as a sequence of singleton transactions.

We further define a path as a sequence of transactions connecting global states as follows.

Definition 9 (Path). *A* path *μ from a global state $\hat{s}_0$ to a global state $\hat{s}_k$ is a sequence $\mu = [m_1, \ldots, m_k]$ of transactions such that there exists a sequence $[\hat{s}_0, \ldots, \hat{s}_k]$ of global states where for all $1 \leq i \leq k$, m_i is admissible in state $\hat{s}_{i-1}$ and $\hat{s}_i$ is the global successor state of $\hat{s}_{i-1}$ after applying m_i.*

A global state $\hat{s}_j$ *is reachable from* $\hat{s}_i$ if there is a path from $\hat{s}_i$ to $\hat{s}_j$. The *length* of a path is the number of its transactions.

The *Multiview Sequence Consistency Problem* (MSCP) deals with the question whether from some global state that is reachable from the global initial state, i.e., $\hat{s}_\iota = (\iota_1, \ldots, \iota_l)$ for the initial states of the state machines of l lifelines, there is a path representing the sequence of messages described in the sequence diagram. In order to be able to express this problem as a propositional formula of polynomial size with respect to the input, we have to bound the length of the path leading to the beginning of the sequence. This bound is included in the *k-Multiview Sequence Consistency Problem* (k-MSCP).

Definition 10 (k-Multiview Sequence Consistency). *Given a sequence diagram $SD = (\mathcal{L}, \mu)$ with $\mathcal{L} = \{L_1, \ldots, L_l\}$ and $\mathsf{sm}(L_i) = (S_i, \iota_i, A_i, T_i)$ for $1 \leq i \leq l$ over a set $\mathcal{M}$ of extended state machines and the alphabets Σ_A and Σ_L, SD and $\mathcal{M}$ are* k-consistent *if there exists a path of length at most k starting at $\hat{s} = (\iota_1, \ldots, \iota_l)$ and leading to a global state $\hat{s}'$ such that a global state $\hat{s}''$ is reachable from $\hat{s}'$ by applying the sequence of messages μ.*

Finally, the *k-Multiview Sequence Consistency Problem* is defined as follows.

k-Multiview Sequence Consistency Problem (k-MSCP)	
Instance:	A sequence diagram $SD = (\mathcal{L}, \mu)$ over a set $\mathcal{M}$ of state machines and the alphabets Σ_A and Σ_L.
Question:	Are SD and $\mathcal{M}$ k-consistent?

5 Encoding

To solve the k-MSCP problem we propose to encode it to the satisfiability problem of propositional logic (SAT). We assume the reader to be familiar with the basics of propositional logic and SAT-solvers (for details we refer to [3,19]). To this end, we build a propositional formula representing an instance of the k-MSCP problem and hand it to a SAT solver. The solver returns `SAT` and a logical model if the sequence diagram of the k-MSCP problem instance can be executed after at most k transactions between the lifelines. The logical model can then be translated back into a concrete sequence of transactions between the lifelines as well as to the state transitions triggered by the application of the messages. The solver returns `UNSAT` if the sequence diagram cannot be executed by the lifelines after at most k message exchanges. In this case, we remove trailing messages one after another from the sequence diagram and call the solver again until the first failing message is found. The encoding presented below is an extension of the encoding discussed in [14, Section 4] where we check the reachability of a global state regardless of a particular message sequence.

We encode an instance of the k-MSCP as the propositional formula φ over a set of variables representing original states, intermediate states, transitions, and alphabet symbols. We assume that all states of all lifelines are pairwise distinct. This natural assumption can be achieved by indexing the states with the name of the respective lifeline. Observe that this also ensures that all transitions of all lifelines are pairwise distinct. Let $\mathcal{M}$ be a set of extended state machines over the alphabet Σ_A, let $SD = (\mathcal{L}, \mu)$ be a sequence diagram over $\mathcal{M}$ with $\mathcal{L} = \{L_1, \ldots, L_l\}$ and $\mu = [m_1, \ldots, m_n]$, let $\mathcal{T} \coloneqq \bigcup_{1 \leq i \leq l} T_i$ be the set of all transitions in all extended state machines, let $\mathcal{S} \coloneqq \bigcup_{1 \leq i \leq l} S_i$ be the set of all original states of all lifelines (all instances of extended state machines), let $\mathcal{S}^* \coloneqq \bigcup_{1 \leq i \leq l} S_i^*$ be the set of all intermediate states of all lifelines, and let $\mathcal{A} \coloneqq \Sigma_A \setminus \{\epsilon\}$. Recall that k is an integer defining the maximum length of the path leading to a global state from which the message sequence in SD is executed. Further, let $k' \coloneqq k + 4n$ be the maximum number of timesteps needed to apply n messages after a path of a maximum length of k. The factor 4 is necessary because moving forward on a transition with the empty symbol ϵ as trigger or effect requires additional timesteps. Then the set of variables occurring in the encoding is given by $\{v^i \mid v \in (\mathcal{T} \cup \mathcal{A} \cup \mathcal{S} \cup \mathcal{S}^*), 0 \leq i \leq k'\}$. That is, each transition, symbol, original state, and intermediate state together with an index up to k' is represented by a variable. We refer to this index as *timestep*.

We further use the following functions to simplify the presentation of the formula. Let $L = (S, \iota, A, T)$ be a lifeline, (s, tr, ϵ, s_t^*) and $(s_t^*, \epsilon, \mathit{eff}, s')$ be transitions of the extended state machine $\mathsf{sm}(L)$. Recall that the states of $\mathsf{sm}(L)$ are made distinct by indexing as described above. The two transitions correspond to a transition $t = (s, tr, \mathit{eff}, s')$ of a non-extended state machine. Additionally, let $m = (\sigma, a, \rho)$ be a message. Then $\mathsf{trans}(L) = T$, $\mathsf{src}(t) \coloneqq s$, $\mathsf{int}(t) \coloneqq s_t^*$, $\mathsf{trg}(t) \coloneqq tr$, $\mathsf{eff}(t) \coloneqq \mathit{eff}$, $\mathsf{tgt}(t) \coloneqq s'$, $\mathsf{snd}(m) \coloneqq \sigma$, $\mathsf{rec}(m) \coloneqq \rho$, and $\mathsf{symb}(m) \coloneqq a$.

The formula φ is given by a conjunction of the following subformulas.

$$\varphi_{\text{init}} \coloneqq \bigwedge_{i=1}^{l} \left(\iota_i^0 \wedge \bigwedge_{s \in S_i \cup \mathcal{S}_i^*, s \neq \iota_i} \overline{s}^0 \right) \wedge \bigwedge_{a \in \mathcal{A}} \overline{a}^0$$

$$\varphi_1 \coloneqq \bigwedge_{i=0}^{k'-1} \bigwedge_{t \in \mathcal{T}} \Big[t^i \rightarrow \Big(\mathsf{src}(t)^i \wedge \mathsf{int}(t)^{i+1} \wedge \Big(\mathsf{trg}(t)^i \neq \epsilon \rightarrow \Big(\mathsf{trg}(t)^i \wedge \overline{\mathsf{trg}(t)}^{i+1} \Big) \Big) \wedge \Big(\mathsf{eff}(t)^i \neq \epsilon \rightarrow \Big(\overline{\mathsf{eff}(t)}^i \wedge \mathsf{eff}(t)^{i+1} \Big) \Big) \Big) \Big]$$

$$\varphi_2 \coloneqq \bigwedge_{i=0}^{k'-1} \bigwedge_{trg \in \mathcal{A}} \Big[trg^i \wedge \overline{trg}^{i+1} \rightarrow \Big(\bigvee_{\substack{t \in \mathcal{T}, \\ \mathsf{trg}(t) = trg}} t^i \bigwedge_{\substack{t_1, t_2 \in \mathcal{T}, \\ \mathsf{trg}(t_1) = \mathsf{trg}(t_2) = trg}} (\overline{t_1}^i \vee \overline{t_2}^i) \Big) \Big]$$

$$\varphi_3 \coloneqq \bigwedge_{i=0}^{k'-1} \bigwedge_{eff \in \mathcal{A}} \Big[\overline{eff}^i \wedge eff^{i+1} \rightarrow \Big(\bigvee_{\substack{t \in \mathcal{T}, \\ eff = \mathsf{eff}(t)}} t^i \bigwedge_{\substack{t_1, t_2 \in \mathcal{T}, \\ \mathsf{eff}(t_1) = \mathsf{eff}(t_2) = eff}} (\overline{t_1}^i \vee \overline{t_2}^i) \Big) \Big]$$

$$\varphi_4 \coloneqq \bigwedge_{i=0}^{k'-1} \bigwedge_{s \in S} \Big[s^i \wedge \overline{s}^{i+1} \rightarrow \bigvee_{t \in \mathcal{T}, s = \mathsf{src}(t)} t^i \Big]$$

$$\varphi_5 \coloneqq \bigwedge_{i=0}^{k'-1} \bigwedge_{t \in \mathcal{T}, \mathsf{eff}(t) \neq \epsilon} \Big[\big(\mathsf{int}(t)^i \wedge \mathsf{int}(t)^{i+1} \big) \rightarrow \mathsf{eff}(t)^{i+1} \Big]$$

$$\varphi_6 \coloneqq \bigwedge_{i=0}^{k'-1} \bigwedge_{t \in \mathcal{T}, \mathsf{eff}(t) \neq \epsilon} \Big[\Big(\mathsf{int}(t)^i \wedge \overline{\mathsf{int}(t)}^{i+1} \Big) \rightarrow \overline{\mathsf{eff}(t)}^{i+1} \Big]$$

$$\varphi_7 \coloneqq \bigwedge_{i=0}^{k'-1} \bigwedge_{t \in \mathcal{T}} \Big[\Big(\mathsf{int}(t)^i \wedge \Big(\mathsf{eff}(t)^{i+1} \neq \epsilon \rightarrow \overline{\mathsf{eff}(t)}^{i+1} \Big) \Big) \rightarrow \Big(\overline{\mathsf{int}(t)}^{i+1} \wedge \mathsf{tgt}(t)^{i+1} \Big) \Big]$$

$$\varphi_8 \coloneqq \bigwedge_{i=0}^{k'-1} \bigwedge_{j=1}^{l} \Big[\bigvee_{s \in (S_j \cup S_j^*)} s^i \wedge \bigwedge_{\substack{s_1, s_2 \in (S_j \cup S_j^*), \\ s_1 \neq s_2}} \big(\overline{s_1}^i \vee \overline{s_2}^i \big) \Big]$$

$$\varphi_{\text{seq}} \coloneqq \bigwedge_{\substack{i \in [1, \ldots, n], \\ j \in [k, k+4, \ldots, k+4n]}} \Big[\mathsf{symb}(m_i)^j \wedge \overline{\mathsf{symb}(m_i)}^{j+1} \wedge \bigvee_{\substack{t \in \mathsf{trans}(\mathsf{snd}(m_i)), \\ \mathsf{eff}(t) = m_i}} \Big(\mathsf{int}(t)^j \wedge \overline{\mathsf{int}(t)}^{j+1} \Big) \wedge \bigwedge_{\substack{a \in \mathcal{A} \\ a \neq m_i}} \big((a^j \rightarrow a^{j+1}) \wedge (a^{j+1} \rightarrow a^{j+2}) \wedge (a^{j+2} \rightarrow a^{j+3}) \big) \Big]$$

The formula φ is satisfiable if and only if a state $\hat{s}$ is reachable by a path of length at most k starting at the global initial state such that starting from $\hat{s}$, the messages in μ are applicable one after another, i. e., there exists a solution to the k-MSCP instance. The intuition behind the encoding can be explained as follows: A state $s \in \mathcal{S}$ is active at timestep i if s^i is *true*. A symbol $a \in \mathcal{A}$

is waiting to be received at timestep i if a^i is *true*. This way, when a transition with a as an effect is triggered at timestep i, then a^i is set to *true*. A state machine which is currently in a state with an outgoing transition with a as a trigger, can consume a in the same or a following timestep $j \geq i$. By doing so, a^j is set to *false*, i.e., it cannot be consumed anymore. Then the subformulas can be understood as follows.

- φ_{init} sets the global state at timestep 0 to the initial states of the lifelines. All other variables representing states and symbols are set to *false*.
- φ_1 ensures that whenever a transition is triggered, the corresponding lifeline changes to the respective intermediate state. Then the trigger symbol is set to *false* and the effect symbol is set to *true*.
- The subformulas φ_i with $i \in \{2, \ldots, 6\}$ are also called *framing axioms*. They ensure that each change of a symbol or of a state has a cause.
  - φ_2 and φ_3 make sure that whenever the polarity of a symbol is changed, there has also been a transition causing this change.
  - φ_4 ensures that a state is only left if a transition causes the change.
  - φ_5 and φ_6 encode that whenever a lifeline leaves an intermediate state, the corresponding symbol is consumed; otherwise the symbol stays available.
- φ_7 forces a lifeline to move to the target state if the effect symbol has been consumed.
- φ_8 ensures that each lifeline is in exactly one state at each timestep.
- Finally, φ_{seq} forces the sequence of messages μ to be executed after the preparation phase.

In formulas φ_1 and φ_7, the expressions $\mathsf{trg}(t)^i \neq \epsilon$ and $\mathsf{eff}(t)^i \neq \epsilon$ occurring in the formula are replaced by the corresponding logical constants ($\top$ and $\bot$) during generation of the formula. The formula is converted to conjunctive normal form, the input format of most SAT solvers. To this end, we apply the Tseitin transformation [25] where necessary.

Note that the encoding allows that nothing happens, i. e., no transaction takes place at a timestep. It is ensured by the framing axioms that in this case, the global state remains the same. This relaxation implicitly encodes the "at most k" steps formulation. If at x indices nothing happens and the execution of the message sequence starts at index k, it means that the length of the transaction sequence executed before the message sequence of the sequence diagram is of length $k - x$. The framing axioms also ensure that lifelines not participating in a transaction do not change.

A solution returned by the SAT solver consists of a set of positive and negative literals representing variables set to *true* or *false*. By extracting the positive literals whose variables represent states and transitions (sets $\mathcal{S}$, $\mathcal{S}^*$, and $\mathcal{T}$) we obtain the path of at most k steps leading to the execution of the sequence diagram, as well as the state changes of the lifelines during the execution of the sequence diagram. If the length of the path is less than k, then for some consecutive indices the state variables represent identical states.

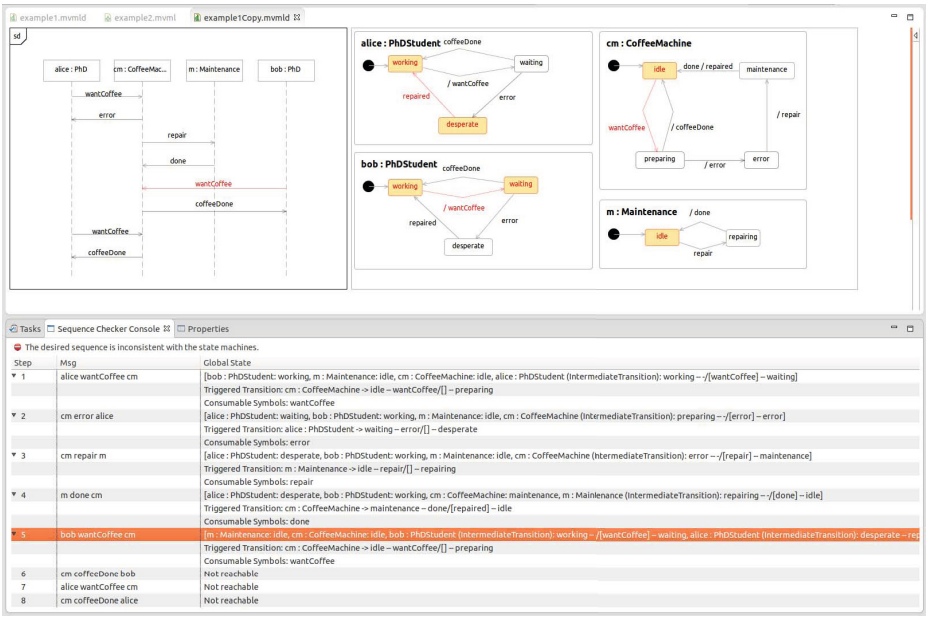

Fig. 4. Screenshot of the graphical user interface

In order to simplify the encoding, we assume that after applying a transaction each symbol can be consumable only once at a timestep. Allowing a symbol to be consumable multiple times requires the integration of counters, which can be realized, e.g., by building upon ideas presented in [24].

6 Implementation

We implemented a tool to solve k-MSCP instances based on the SAT encoding presented above as plugin for the Eclipse framework[2]. It can be downloaded from

http://modelevolution.org/updatesite/

To define the input language of our tool, i.e., the language of state machines and sequence diagrams, we formulated a metamodel in Ecore, the modeling language of the Eclipse modeling framework (EMF)[3]. This metamodel contains all language concepts discussed in this paper. Strongly inspired by the UML metamodel, it is designed for the easy integration of future language extensions.

The input models provided by the user of our tool are automatically translated to propositional logic using the encoding described above. After the encoding phase, the obtained formula is passed to the solver Sat4j [17], a Java-based SAT solver integrated in our tool.

[2] `http://www.eclipse.org/`
[3] `http://www.eclipse.org/emf/`

If the SAT solver returns `SAT`, then at least one execution path in the state machines exists which conforms to the message sequence in the sequence diagram. If the SAT solver returns `UNSAT` then the state machines and the sequence diagram are inconsistent. In this case, the last message is removed, and the SAT solver is called again, until it eventually returns `SAT`. The remaining sequence diagram is consistent with the state machines, and the information about the removed messages can be used for debugging purposes.

Then the solution returned by the SAT solver is mapped back to the model elements and visualized in the graphical user interface as shown in Fig. 4. Our user interface allows the user to step through a whole trace by coloring the current messages, transitions, and states. This visualization is very useful to understand the interplay and the behavior of the different state machines and provides valuable debugging assistance.

7 Evaluation

We thoroughly tested our tool using a grammar-based white-box fuzzing approach [29]. This method generates random input models based on a grammar provided by an EMF metamodel. We employed a random input model generator based on the tool presented in [29] but using a different definition of consistency between sequence diagrams and state machines. Other than taking into account only the receive event of a message exchange as is the case in [29], we consider both the send and the receive events. The tool consists of two components, a *generator* to build syntactically correct diagrams, and a *simulator* to ensure that a message sequence can be executed after a certain number of steps.

We applied white-box fuzzing for both debugging and performance evaluation purposes of our SAT encoding of k-MSCP. In the following we describe the random generation of instances and the results of the evaluation of our SAT-based k-MSCP solving tool.

7.1 Random Instance Generation

The instance generation tool first builds a set of state machines and then generates a sequence diagram consistent with these state machines. Consistency is ensured by the simulator following the generated sequence and proposing subsequent messages. In order to also generate inconsistent diagrams, messages are removed at random from an already generated message sequence. Further, if the considered bound for the generation of the diagrams is higher than the bound set in the encoding, the SAT solver may return `UNSAT` even though the message sequence is executable. The tool takes the following parameters to define the two views:

- nrStateMachines: Number of state machines to be created.
- minNrStates and maxNrStates: Bounds on the number of states per state machine. The actual number of states is chosen randomly between and including these bounds for each state machine.

- minNrTrans and maxNrTrans: Bounds on the number of transitions per state machine. The actual number of transitions is chosen randomly between and including these bounds for each state machine.
- nrSymbols: The size of the alphabet the state machines are defined over.
- probTrigger: The probability of a transition to contain a trigger symbol other than ϵ.
- probEff: The probability of a transition to contain an effect symbol other than ϵ.
- nrLifelines: The number of lifelines to be contained in the sequence view.
- nrMessages: The number of messages to be contained in the sequence view.

For each state machine, the algorithm randomly chooses a number of states and transitions in between the bounds minNrStates, maxNrStates, minNrTrans, and maxNrTrans, and connects the states by transitions randomly in a way such that no state is isolated. To at least one outgoing transition of the initial state, the trigger ϵ is added, and to all other transitions, a trigger other than ϵ is added with probability probTrigger. To each transition containing ϵ as trigger an effect other than ϵ is added, and to all other transitions an effect other than ϵ is added with probability probEff. Each time a trigger or an effect is added, a fresh symbol is created and added to the alphabet until the alphabet has reached size nrSymbols. After that, the trigger and effect symbols are chosen randomly.

Then a sequence diagram consistent with the state machine view is created according to the two parameters nrLifelines and nrMessages. In order to ensure the consistency, a model simulator keeps track of the global state of the lifelines' state machines. For each lifeline, a state machine is chosen at random from the state machine view. If nrLifelines $>$ nrStateMachines then it is ensured that each state machine is instantiated at least once. The main data structure in the simulator represents possible global states as a hashmap with lifelines as keys and a set of states of the state machine instanced by the lifeline as value. For each lifeline, the hashmap is initialized with all original and intermediate states of the respective state machine. All admissible messages are calculated according to the current global state stored in the simulator. One message is chosen at random and the simulator is updated according to all possible successor states with respect to the application of the chosen message. Note that the state machines are non-deterministic, and therefore the number of possible states and admissible messages can become very large.

To obtain unsatisfiable instances, we generate one more message than required and remove one message at random among all messages except the first one. This procedure, however, still results in a satisfiable instance in many cases because a different path than the one followed by the simulator might be possible.

7.2 Testing Environment and Results

We selected a set of parameter values for the parameters described in Section 7.1 in order to generate sets of instances. The parameter values influence each other to a great extent, and it can easily happen that no or only a small message sequence can be generated for the sequence diagram. For example, a high value for

Table 1. Parameter settings

	small	medium	large
minNrStates	2	4	7
maxNrStates	3	6	10
minNrTrans	4	8	21
maxNrTrans	6	12	30
nrLifelines	3	5	8
nrMessages	4	10	20

probTrigger along with a high value for nrSymbols results in transitions containing different triggers and effects, making the generation of a consistent communication sequence difficult.

We grouped instances created according to different parameter sets into three different groups according to their size. Table 1 describes the parameter settings for each group. The following parameters have been set to the same values for all instances. nrStateMachines has been set to 3 for all instances because the size of the instance is regulated by the nrLifelines, i.e., the number of instantiations of the state machines, nrSymbols has been set to minNrStates, probTrigger and probEff have been set to 0.9, and k has been set to maxNrStates.

The experiments were executed on a computer with an Intel Core i5-540M CPU with 2.53GHz and 8GB of RAM. Table 2 describes the results of our experiments over 1,000 randomly generated instances in each category. We distinguish both encoding and solving time by `UNSAT` and `SAT` instances. The time required to determine the failing message in an `UNSAT` instance is significantly longer than the time required to determine satisfiability and to return a model. This is the case because unsatisfiable instances are modified by removing the last message and are sent back to the SAT solver until the failing message is found. The numbers of clauses and numbers of variables refer to the initial encoding of each instance, not taking into account the modified instances after unsatisfiability is detected, as the re-encoding results in less variables and clauses than the initial encoding.

Table 2. Average results over 1,000 runs for each category

	small	medium	large
Encoding time `SAT` (ms)	11	180	2,543
Solving time `SAT` (ms)	4	201	9,476
Encoding time `UNSAT` (ms)	34	970	27,848
Solving time `UNSAT` (ms)	8	727	179,914
Nr variables	1,802	12,746	88,560
Nr clauses	9,652	118,245	1,700,101
Nr instances `SAT`	837	750	803
Nr instances `UNSAT`	163	250	197

The difference in numbers of `SAT` instances and `UNSAT` instances can be explained by the way instances are created. In order to generate a sequence diagram at random without too much overhead, the state machines need many transitions with not too many symbols. However, in this case, when a valid sequence is found and a message removed, chances are high, that this "cropped" sequence can still be found by a path other than the one followed by the simulator, because of the previous requirement to have many transitions and few symbols.

It can be seen that the overall runtimes are acceptable even if executed on an standard hardware. As can be expected, the solving time scales worse than the encoding time. The overall runtime for `UNSAT` instances could probably be improved by implementing a binary search to find the failing message, instead of removing trailing messages one after another. This way, the SAT solver has to be called less often.

8 Conclusion and Future Work

We presented a novel SAT-based approach to check the consistency between state machines and sequence diagrams. To this end, we concisely formulated a formal semantics of the considered modeling language concepts. On this basis we were able to obtain an exact formal description of the consistency checking problem which was then directly mapped to SAT. The encoding reuses ideas and techniques well established for formulating planning problems. We obtained an encoding which is extremely flexible, efficiently processable, and still keeps the information necessary to map the solutions obtained from the SAT solver back to the modeling environment.

With our current solution we have a powerful tool for checking the consistency between different views in UML models. In combination with our other SAT-based encodings [14,30] we have now the means to establish a uniform framework supporting safe model evolution.

In future work, we plan to consider additional modeling language concepts like hierarchical states in the state machines or combined fragments in the sequence diagrams. Especially for the latter case which introduces programming language constructs like loops into the diagram, techniques applied in software verification must be considered. Also it is possible to apply ideas from this encoding to other diagram types like the UML activity diagram. Further, we plan to extend our approach for automatic repair. In particular, the encoding can be modified such that missing messages in the sequence diagram can be filled. This scenario can happen in automated merging environments as, for instance, in model versioning systems [6].

References

1. Bernardi, S., Donatelli, S., Merseguer, J.: From UML sequence diagrams and statecharts to analysable Petri net models. In: 3rd International Workshop on Software and Performance, pp. 35–45. ACM (2002)

2. Bézivin, J.: On the unification power of models. Software & Systems Modeling 4(2), 171–188 (2005)
3. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. FAIA, vol. 185. IOS Press (2009)
4. Brosch, P., Egly, U., Gabmeyer, S., Kappel, G., Seidl, M., Tompits, H., Widl, M., Wimmer, M.: Towards semantics-aware merge support in optimistic model versioning. In: Kienzle, J. (ed.) MODELS 2011 Workshops. LNCS, vol. 7167, pp. 246–256. Springer, Heidelberg (2012)
5. Brosch, P., Egly, U., Gabmeyer, S., Kappel, G., Seidl, M., Tompits, H., Widl, M., Wimmer, M.: Towards Scenario-Based Testing of UML Diagrams. In: Brucker, A.D., Julliand, J. (eds.) TAP 2012. LNCS, vol. 7305, pp. 149–155. Springer, Heidelberg (2012)
6. Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: An introduction to model versioning. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) SFM 2012. LNCS, vol. 7320, pp. 336–398. Springer, Heidelberg (2012)
7. Egyed, A.: Instant consistency checking for the UML. In: 28th International Conference on Software Engineering (ICSE), pp. 381–390. ACM (2006)
8. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Testing the consistency of dynamic UML diagrams. In: 6th International Conference on Integrated Design and Process Technology (IDPT) (2002)
9. Feng, T.H., Vangheluwe, H.: Case study: Consistency problems in a UML model of a chat room. In: Workshop on Consistency Problems in UML-based Software Development, p. 18 (2003)
10. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering (FOSE), pp. 37–54. IEEE Computer Society (2007)
11. Gabmeyer, S., Kaufmann, P., Seidl, M.: A classification of model checking-based verification approaches for software models. In: STAF Workshop on Verification of Model Transformations (VOLT), pp. 1–7 (2013)
12. Graaf, B., van Deursen, A.: Model-driven consistency checking of behavioural specifications. In: 4th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES), pp. 115–126 (2007)
13. Inverardi, P., Muccini, H., Pelliccione, P.: Automated check of architectural models consistency using SPIN. In: 16th Annual International Conference on Automated Software Engineering (ASE), pp. 346–349. IEEE Computer Society (2001)
14. Kaufmann, P., Kronegger, M., Pfandler, A., Seidl, M., Widl, M.: Global state checker: Towards SAT-based reachability analysis of communicating state machines. In: 10th Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa). CEUR Workshop Proceedings, vol. 1069, pp. 31–40 (2013)
15. Knapp, A., Wuttke, J.: Model checking of UML 2.0 interactions. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 42–51. Springer, Heidelberg (2007)
16. Lam, V.S.W., Padget, J.: Consistency Checking of Sequence Diagrams and Statechart Diagrams Using the π-Calculus. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 347–365. Springer, Heidelberg (2005)
17. Le Berre, D., Parrain, A.: The Sat4j Library, Release 2.2, System Description. Journal on Satisfiability, Boolean Modeling and Computation 7, 59–64 (2010)
18. Lucas, F.J., Molina, F., Toval, A.: A systematic review of UML model consistency management. Information and Software Technology 51(12), 1631–1645 (2009)
19. Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)

20. Pelliccione, P., Inverardi, P., Muccini, H.: CHARMY: A Framework for Designing and Verifying Architectural Specifications. IEEE Transactions on Software Engineering 35(3), 325–346 (2008)
21. Rintanen, J.: Planning and SAT. In: Handbook of Satisfiability. FAIA, vol. 185, pp. 483–504. IOS Press (2009)
22. Schäfer, T., Knapp, A., Merz, S.: Model Checking UML State Machines and Collaborations. Electronic Notes in Theoretical Computer Science 55(3), 357–369 (2001)
23. Selic, B.: What will it take? A view on adoption of model-based methods in practice. Software & Systems Modeling 11(4), 513–526 (2012)
24. Sinz, C.: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)
25. Tseitin, G.S.: On the complexity of derivations in the propositional calculus. Studies in Mathematics and Mathematical Logic, Part II, 115–125 (1968)
26. Usman, M., Nadeem, A., Kim, T., Cho, E.: A survey of consistency checking techniques for UML models. In: Advanced Software Engineering and Its Applications (ASEA), pp. 57–62. IEEE Computer Society (2008)
27. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between UML models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 326–340. Springer, Heidelberg (2003)
28. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: 22nd International Conference on Software Engineering (ICSE), pp. 314–323. ACM (2000)
29. Widl, M.: Test Case Generation by Grammar-Based Fuzzing for Model-Driven Engineering. In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2013. LNCS, vol. 7857, pp. 278–279. Springer, Heidelberg (2013)
30. Widl, M., Biere, A., Brosch, P., Egly, U., Heule, M., Kappel, G., Seidl, M., Tompits, H.: Guided Merging of Sequence Diagrams. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 164–183. Springer, Heidelberg (2013)