Intra- and Interdiagram Consistency Checking of Behavioral Multiview Models

Petra Kaufmann[a,*], Martin Kronegger[b,*], Andreas Pfandler[b,e,*], Martina Seidl[a,c,*], Magdalena Widl[d,*]

[a] *Business Informatics Group, TU Wien, Karlsplatz 13, 1040 Wien, Austria*
[b] *Database and Artificial Intelligence Group, TU Wien, Karlsplatz 13, 1040 Wien, Austria*
[c] *Inst. f. Formal Models and Verification, JKU Linz, Altenbergerstr. 69, 4040 Linz, Austria*
[d] *Knowledge-Based Systems Group, TU Wien, Karlsplatz 13, 1040 Wien, Austria*
[e] *School of Economic Disciplines, Univ. Siegen, A.-Reichweinstr. 2, 57076 Siegen, Germany*

Abstract

Multiview modeling languages like UML are a very powerful tool to deal with the ever increasing complexity of modern software systems. By splitting the description of a system into different views—the diagrams in the case of UML—system properties relevant for a certain development activity are highlighted while other properties are hidden. This multiview approach has many advantages for the human modeler, but at the same time it is very susceptible to various kinds of defects that may be introduced during the development process. Besides defects which relate only to one view, it can also happen that two different views, which are correct if considered independently, contain inconsistent information when combined. Such inconsistencies between different views usually indicate a defect in the model and can be critical if they propagate up to the executable system.

In this paper, we present an approach to formally verify the reachability of a global state of a set of communicating UML state machines, i.e., we present a solution for an intradiagram consistency checking problem. We then extend this approach to solve an interdiagram consistency checking problem. In particular, we verify whether the message exchange modeled in a UML sequence diagram conforms to a set of communicating state machines.

For solving both kinds of problems, we proceed as follows. As a first step, we formalize the semantics of UML state machines and of UML sequence diagrams. In the second step, we build upon this formal semantics and encode both verification tasks as decision problems of propositional logic (SAT) allowing the use of efficient SAT technology. We integrate both approaches in a graphical modeling environment, enabling modelers to use formal verification techniques without any special background knowledge. We experimentally evaluate the

*Corresponding authors
Email addresses: kaufmann@big.tuwien.ac.at (Petra Kaufmann), kronegger@dbai.tuwien.ac.at (Martin Kronegger), pfandler@dbai.tuwien.ac.at (Andreas Pfandler), martina.seidl@jku.at (Martina Seidl), widl@kr.tuwien.ac.at (Magdalena Widl)

scalability of our approach.

Keywords: Multiview Modeling, Unified Modeling Language, Consistency Checking, SAT Encodings

1. Introduction

A major difference between traditional software engineering and model-driven engineering (MDE) [5] lies in the nature of the core development artifacts. These artifacts, which in traditional software engineering comprise mainly textual code, are represented by (visual) software models in MDE. Often software models are expressed in multiview modeling languages like the *Unified Modeling Language* (UML)[25], where a focused view on specific aspects (e.g., behavioral or structural aspects) of the system under consideration is given. The goal of MDE is to leverage the abstraction power offered by software models to deal with the complexity of modern software systems [3], and to further exploit the models to automatically generate executable code with little or no intervention of a human developer [28].

The increasing valorization of software models imposes stronger demands and expectations on their correctness. In their role as core development artifacts, software models are increasingly sensitive to the impact of evolution and therefore more exposed to the introduction of errors [13]. Especially the abstraction power of multiview modeling languages as offered by UML bears the danger of introducing inconsistencies into the model under development [22].

Inconsistent software models can be the root of severe problems if they are employed for automatic code generation because inconsistencies can propagate to the executable system and result in serious errors in the application. Hence, if the diagrams do not complement each other in a consistent manner, then the benefits of multiview modeling will decrease or even vanish [28]. Due to the multiview nature and the size of software models, inconsistencies are often hard to spot for a human developer. Especially when the models are not directly executable or when no simulation environment is available, testing and debugging is difficult. Here, formal verification methods can help to ensure that the models fulfill intradiagram and interdiagram consistency criteria, i.e., the consistency is ensured within one diagram and between different diagrams, respectively.

In this paper, we first consider the following *intradiagram consistency checking problem*: For a set of communicating state machines, which describe the internal behavior of objects, we check if it is consistent to assume that a specific system configuration, i.e., a (partial) global state, is reachable from the initial state. If the answer is affirmative, then the respective execution path is returned. Hence, the (partial) global states are test cases, asserting allowed or forbidden system configurations.

We then extend this intradiagram consistency checking problem to an *interdiagram consistency checking problem* of state machines and sequence diagrams. Sequence diagrams focus on interaction scenarios between different instances of

classes and the respective state machines. 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 in a state reachable from the initial state. If a forbidden sequence of messages can be executed, 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.

A crucial ingredient for an implementation of the above mentioned consistency checks is a well-defined, formal semantics of the diagrams types that are to be verified. Therefore, we first introduce a formal semantics for UML state machines and UML sequence diagrams, and then we propose an approach to solve the consistency problems based on a reduction to the satisfiability problem of propositional logic (SAT) [4]. For SAT powerful solvers are available, which can successfully be used out of the box in many applications.

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 sequence diagrams and state machines along with their interplay. Further, we introduce the notion of global state reachability and sequence consistency, which are essential for our problem definitions. These problem definitions allow us to come up with a translation of the consistency checking problems to propositional formulas, which can be handed to a SAT solver (Section 5). In Section 6 we discuss the implementation based on the Eclipse Modeling Framework and in Section 7 we present a detailed evaluation on randomly generated and on crafted models. Finally, we conclude with an outlook on future work.

This paper is an extended and revised version of our SLE 2014 [17] paper. Besides details on the technical realization and further experiments, we present the complete workflow of our verification framework. This includes enhancements of the global state checking approach, which was presented at the MoDeVVa 2013 Workshop [16].

2. Related Work

We consider two different streams of work related to our approach. On the one hand, we review literature on reachability checking for state machines and on the other hand we give an overview on approaches for consistency checking between state machines and sequence diagrams.

Reachability Checking. Several works have been presented which deal with the transformation of UML state machines to input languages of model checkers (see

for example [1, 9, 18, 21, 24]). These languages provide high-level constructs to model software systems and in many aspects they provide similar constructs as modeling languages like UML do, in particular UML state machines.

However, one of the major challenges of such approaches is to overcome semantic heterogeneities. This has already been recognized by Niewiadomski et al., who propose an encoding to propositional logic for bounded reachability analysis of state machines, which they show to be more efficient than translations to standard model checkers [23]. In this paper, we follow the approach of [23] to encode the reachability problem to SAT, but propose an alternative encoding where we formulate the reachability analysis problem of UML state machines inspired by encodings as used for solving planning problems [27]. As a result, we obtain an intuitive encoding which allows us to directly extract a path from the initial state to the given global state if it is reachable.

Consistency Checking between State Machines and Sequence Diagrams. In our previous work [6, 7], we employed Spin to ensure that given traces do not occur during the execution of a set of state machines. With this encoding we could not ensure that a given message sequence is possible, and we had to overcome the semantic differences of UML state machines and Promela constructs.

Many other formal approaches have been presented to check the consistency between different diagrams, 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.x 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 reader to comprehensive surveys [14, 22, 32]. Lam and Vitus [19] 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. [33] 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 to check the consistency between different diagrams [2]. Communication, however, is only considered at the class level and not at the object level. Engels et al. [11] propose to check consistency by evaluating dedicated consistency constraints represented in form of collaborations. For this purpose, an interpreter is provided. Egyed [10] 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 [15] who suggest to synthesize a state machine from given sequence diagrams and then compare the automatically derived state machine to the state machine implemented by the developers. The comparison step has to be performed manually because naturally, there are many mismatches between the automatically generated and the manually developed state machines. Feng and Vangheluwe propose to use a simulation-based approach for consistency

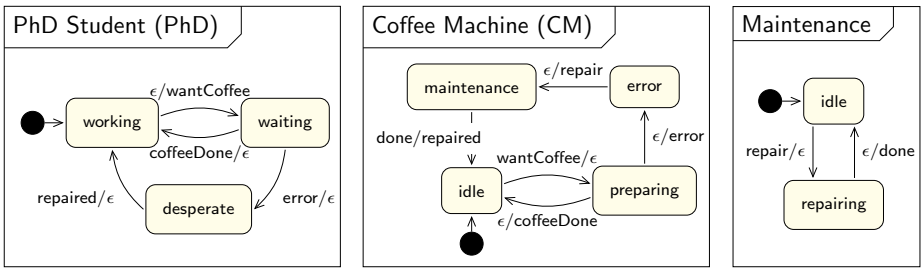

Figure 1: Three state machines modeling a PhD student, a coffee machine, and a maintenance unit.

checking [12].

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 [14, 22, 32] for detailed surveys.

3. A Motivating Example

To motivate our work we present an intuitive example before introducing the formal definitions of the problems and the details of our encodings. 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 UML state machines, 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. If ϵ occurs on the right side of the "/", this indicates that no effect happens. The initial state is indicated by an incoming arc from a black dot. Therefore, in the example, a PhD student starts in state working, a coffee machine starts in state idle, and a maintenance unit starts also in state idle. For checking the internal consistency of the state machine diagram, it helps to know whether certain combinations of states are reachable or not. For example it should never happen that the PhD student is in state waiting, the coffee machine is in state maintenance and the maintenance unit is in state idle at the same time. Because then neither of the state machines can continue.

Instances of state machines communicate with one another 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. The transition is fully executed only if the effect can be sent successfully, i.e., if this effect can also be received by another instance of a state machine. An outgoing transition carrying the special symbol ϵ as trigger can be initiated without receiving any symbol. This happens if an on-completion-event is triggered. Such an event occurs in UML

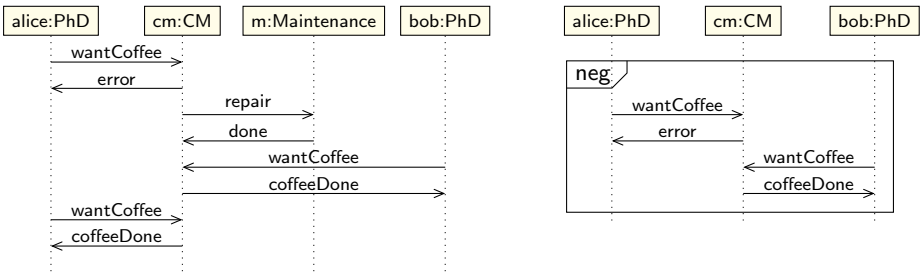

Figure 2: (Left) A sequence diagram depicting a desired scenario that is inconsistent with the state machines of Fig. 1. The state machines have to be changed in order to allow the scenario. (Right) A sequence diagram depicting a forbidden scenario that is inconsistent with the state machines of Fig.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

state machines if the internal actions executed in a state have terminated. In our case, an on-completion-event is immediately triggered.

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 multiple *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. A global state is a tuple of states of the state machines instantiated by the lifelines.

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 does not send 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 approach returns the sequence of messages up to the message that cannot be sent or received, i.e., 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. 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

In the following, we formally introduce the modeling language concepts used in the rest of the paper. Based on these definitions we then formulate the Global State Checking Problem (GSC) and the Multiview Sequence Consistency Problem (MSC).

The core elements for defining state machines, sequence diagrams, and their interaction are the symbols of the alphabets Σ_A and Σ_L where Σ_A contains the special symbol ϵ. The alphabet Σ_A contains symbols which label messages in a sequence diagram, trigger transitions, and occur as effects in 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 a state machine 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 there is no transition $(s, \epsilon, \epsilon, s') \in T$ for any $s, s' \in S$.

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, Maintenance, shown in Fig. 1 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. The state machine Maintenance in Fig. 1 has two transitions: $T = \{(\mathsf{idle}, \mathsf{repair}, \epsilon, \mathsf{repairing}), (\mathsf{repairing}, \epsilon, \mathsf{done}, \mathsf{idle})\}$.

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 from M and the effect symbol must be received by a state machine different from M. Either trigger or effect can be the special symbol ϵ which stands for an empty trigger or an empty 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. For the ease of presentation, we restrict the maximal number of effects on a transition to one. Both, definitions of state machines and the encoding presented in the next section can be easily extended to transitions with an arbitrary number of effects.

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

An extended state machine introduces an *intermediate state* s_t^* for each transition t. An 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*. An intermediate state can be identified by combining the source state, the trigger, the effect, and the target state of the transition it refers to. Any state machine can be translated to exactly one extended state machine and vice versa.

This way, an extended state machine helps to distinguish between the event of having received the trigger and the event of being able to send the effect. Other than a non-extended state machine, it can contain transitions which have ϵ as both trigger and effect. These transitions connect intermediate states to original states when the corresponding transition of the non-extended state machine has ϵ as effect. We call such intermediate states the *environment* of the

respective original state. If an extended state machine is inside the environment of some state s, then it can be treated as if it were in s.

Definition 3 (Environment). Given a state machine $M = (S, \iota, A, T)$, its extended state machine $M^* = (S \cup S^*, \iota, A, T^*)$, and a state $s \in S$, the *environment* of s is given by the function $\mathsf{environ} : S \to \mathcal{P}(S^*)$ such that $\mathsf{environ}(s) \coloneqq \{s^* \mid (s^*, \epsilon, \epsilon, s) \in T^*\}$.

Fig. 3 depicts the extended state machine of the state machine PhD Student. The intermediate states are represented by black diamonds with rounded corners. The intermediate state between the states desperate and working can be identified by its source state, trigger, effect, and target state as <desperate/repaired/ϵ/working>. The environment of the state working is {<desperate/repaired/ϵ/working>,<waiting/coffeeDone/ϵ/working>}, i.e., the two intermediate states on its incoming transitions that contain ϵ as effect.

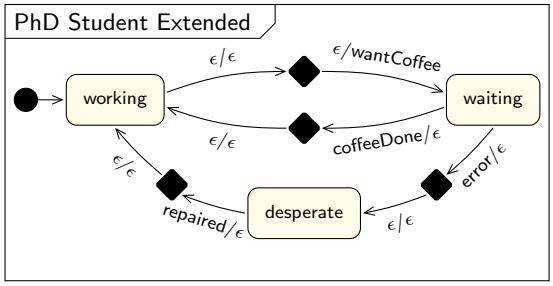

Figure 3: Extended state machine corresponding to the state machine PhD student.

Next, we formally define sequence diagrams, starting with their two components, lifelines and messages.

Definition 4 (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 an extended state machine associated with the lifeline.

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 extended state machine and M^* is the extended state machine the lifeline 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 5 (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 sender lifeline,
- $a \in \Sigma_A$ is the message symbol, and

- $\rho \in \mathcal{L} \setminus \{\sigma\}$ is the receiver 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 on figures depicting sequence diagrams. The receiver lifeline ρ refers to an extended state machine.

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

Definition 6 (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, and
- $\mu = [m_1, \ldots, m_n]$ 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 $\mu = [((\mathsf{alice}, \mathsf{PhD}), \mathsf{wantCoffee}, (\mathsf{cm}, \mathsf{CM})), \ldots, ((\mathsf{cm}, \mathsf{CM}), \mathsf{coffeeDone}, (\mathsf{bob}, \mathsf{PhD}))]$ of messages.

To describe the interaction between lifelines via messages we define a *global state* which captures a configuration of a collection of (extended) state machines instantiated by the lifelines. We base this definition on a collection rather than on a set of (extended) state machines since a state machine can be instantiated by more than one lifeline.

Definition 7 (Global State). Given a collection $\mathcal{M} = \{M_1, \ldots, M_l\}$ of (extended) state machines, a global state $\hat{s}$ is a tuple $(s_1, \ldots, s_l) \in S_1 \times \cdots \times S_l$ where S_i is the set of states of M_i for $1 \leq i \leq l$.

A *global state* is a tuple of states containing exactly one state per instantiation of an (extended) state machine.

For instances of extended versions of the three state machines in Fig. 1, an example for a global state is $(\mathsf{desperate}, <\mathsf{maintenance/done/repaired/idle}>, \mathsf{idle})$ where the second state refers to the intermediate state on the transition from $\mathsf{maintenance}$ to idle in state machine CM.

In order to allow that states of some state machines remain unspecified, i.e., it is not important which state is reached in these state machines when the other state machines are in certain states, we introduce the notion of partial global state.

Definition 8 (Partial Global State). Given a collection $\mathcal{M} = \{M_1, \ldots, M_l\}$ of (extended) state machines with $M_i = (S_i, \iota_i, A_i, T_i)$ for $1 \leq i \leq l$, a *partial global state* is an l-tuple $\hat{s}_p \in S_1 \cup \{?\} \times \cdots \times S_l \cup \{?\}$, where ? is a new symbol not contained in any S_i.

A partial global state and a global state *match* if the global state and the partial global state agree on all components where the respective component of the partial state is different from ?. If the partial global state refers to a state machine and the global state to an extended state machine, then the environment (cf. Definition 3) has to be taken into account.

Definition 9 (Matching). Let $\mathcal{M}^* = \{M_1^*, \ldots, M_l^*\}$ be a set of extended state machines with $M_i^* = (S_i \cup S^*, \iota_i, A_i, T_i)$ for $1 \leq i \leq l$, $\hat{s} = (s_1, \ldots, s_l)$ be a global state with $s_i \in S_i \cup S_i^*$ for $1 \leq i \leq l$, and $\hat{s}_p = (p_1, \ldots, p_l)$ be a partial state with $p_i \in S_i$ for $1 \leq i \leq l$. Then $\hat{s}$ *matches* $\hat{s}_p$ if for all $1 \leq i \leq l$ with $p_i \neq ?$ it holds that if $s_i \in S$ then $s_i = p_i$ and if $s_i \in S^*$ then $s_i \in \mathsf{environ}(p_i)$.

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 10 (Admissibility and Application of a Message). Given the alphabet Σ_A, a set $\mathcal{L} = \{L_1, \ldots, L_l\}$ of lifelines with an extended state machine $\mathsf{sm}(L_i) = (S_i, \iota_i, A_i, T_i)$ for $1 \leq i \leq l$, 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 $L_s \neq \varepsilon$, i. e., $L_s \in \mathcal{L}$, and $r, s \in \{1, \ldots, l\}$, then
  (1) $(s_s, \epsilon, a, s'_s) \in T_s$, and
  (2) $(s_r, a, \epsilon, s'_r) \in T_r$.
- If $L_s = \varepsilon$, i. e., $m = (\varepsilon, \epsilon, L_r)$, and $1 \leq r \leq l$, 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* $\hat{s}' \in S_1 \times \cdots \times S_l$ is reached. If $L_s \neq \varepsilon$ then $\hat{s}' = (s_1, \ldots, s'_s, \ldots, s'_r, \ldots, s_l)$ (w.l.o.g. assume that $s < r$). Otherwise, if $L_s = \varepsilon$, then $\hat{s}' = (s_1, \ldots, s'_r, \ldots, s_l)$.

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

In the global state $s = (\mathsf{desperate}, <\mathsf{maintenance/done/repaired/idle}>, \mathsf{idle})$ of the lifelines $(\mathsf{alice}, \mathsf{PhD})$, $(\mathsf{cm}, \mathsf{CM})$, $(\mathsf{m}, \mathsf{Maintenance})$ the set of admissible 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 lifeline, unless it is ε, 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* which is defined as follows.

Definition 11 (Transaction). Given a set $\mathcal{L} = \{L_1, \ldots, L_l\}$ 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. Along a *path* we can step through the global states of a set of state machines.

Definition 12 (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 path is a sequence of transactions connecting global states. The *length* of a path is the number of its transactions.

Definition 13 (Reachability). A global state $\hat{s}'$ is *reachable* from a global state $\hat{s}$ if there is a path from $\hat{s}$ to $\hat{s}'$.

The *k-Global State Checking* problem (k-GSC) asks whether for a given global state of a set of state machines a certain (partial) global state is reachable by a path of length at most k.

k-Global State Checking (k-GSC)	
Instance:	A set $\mathcal{M}$ of state machines, a global state $\hat{s}$ over $\mathcal{M}$, and a partial global state $\hat{s}_p$.
Question:	Is there a path of length at most k from $\hat{s}$ to a global state $\hat{s}'$ that matches $\hat{s}_p$?

We also refer to the global state $\hat{s}$ as *global initial state* and to the partial global state $\hat{s}_p$ as *goal*. The global initial state typically contains each state machine's initial state.

The *k-Multiview Sequence Consistency* problem (k-MSC) 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) in k steps, there is a path representing the sequence of messages described in the sequence diagram. 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.

Definition 14 (k-Multiview Sequence Consistency). Given a set $\mathcal{M}$ of extended state machines, the alphabets Σ_A and Σ_L, and a sequence diagram $SD = (\mathcal{L}, \mu)$ over $\mathcal{M}$, Σ_A, and Σ_L with $\mathcal{L} = \{L_1, \ldots, L_l\}$ and $\mathsf{sm}(L_i) = (S_i, \iota_i, A_i, T_i)$ for $1 \leq i \leq l$, 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 μ of messages.

The *k-Multiview Sequence Consistency* problem is then defined as follows.

k-Multiview Sequence Consistency (k-MSC)	
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-GSC and k-MSC problems we propose to encode them into the satisfiability problem of propositional logic (SAT). We assume the reader to be familiar with the basics of propositional logic and with SAT solvers (for details we refer to [4, 26]). To this end, we build a propositional formula representing an instance of the k-GSC problem or the k-MSC problem and hand it to a SAT solver. If the (partial) global state in an instance of k-GSC is reachable, the solver returns `SAT` together with an assignment to the variables that represents the path leading to the given (partial) global state. Similarly, if the sequence diagram of an instance of the k-MSC problem can be executed after at most k transactions between the lifelines, the solver returns `SAT` and an assignment representing such a path. In both cases a logical model, i.e., a satisfying truth assignment, can easily be translated back into a concrete sequence of transactions between lifelines and to the state transitions triggered by the messages exchange. The solver returns `UNSAT` if the (partial) global state cannot be reached or if the sequence diagram cannot be executed by the lifelines after at most k message exchanges. In the latter case, we remove trailing messages one

after another from the sequence diagram and call the solver again until the first failing message is found.

We encode instances of the k-GSC problem and of the k-MSC problem as a propositional formula over a set of variables representing original states, intermediate states, transitions, and alphabet symbols. We denote this formula by φ^{gsc} for k-GSC and by φ^{msc} for k-MSC.

The set of variables and a large part of the encoding are very similar for both problems. We first introduce the set of variables both encodings are based on, then present the subformulas the two problems have in common, and finally complete the encodings with two individual formulas for each problem.

The set of variables is created for both problems as follows. 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 extended state machines, let $\mathcal{S}^* \coloneqq \bigcup_{1 \leq i \leq l} S_i^*$ be the set of all intermediate states of all extended state machines, and let $\mathcal{A} \coloneqq \Sigma_A \setminus \{\epsilon\}$. In the case of k-MSC, state machines are duplicated when instantiated more than once. We index the state variables with the name of the lifeline instantiating the (extended) state machine in order to make them unique. This way also the variables representing transitions are pairwise distinct for each instantiation.

The integer k plays a slightly different role in both problems. It defines the maximum length to reach a certain goal state in k-GSC. In contrast, for k-MSC, it defines the maximum length of the path leading to a global state from which the message sequence specified in the sequence diagram can be executed. For k-MSC, let $k' \coloneqq k + 4n$ be the maximum path length needed to apply n messages after an "initial" 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 steps. In the following, we use κ in the definition of the sets of variables and the common subformulas of the two problems to denote k for k-GSC and to denote k' for k-MSC, respectively.

The set of variables occurring in the encoding is given by the union of the following sets representing message symbols, transitions, original states, and intermediate states at different positions of a path. We refer to these positions as *timesteps*.

- $\{a^i \mid a \in \mathcal{A}, 0 \leq i \leq \kappa\}$ is a set of variables that encode whether a message symbol is available to be consumed by another machine at a certain timestep. A variable a^i is set to true if at timestep i some extended state machine M tries to send a, i.e., a transition in M has received a trigger and is waiting for a to be consumed by a different extended state machine in order to complete the transition. If a^i is set to true, this indicates that the symbol is available to be consumed at a timestep $j > i$. When the symbol is consumed at timestep j, then the respective variable is set to false.
- $\{t^i \mid t \in \mathcal{T}, 0 \leq i \leq \kappa\}$ is a set of variables that encode transitions executed

at a timestep due to a message placed at that timestep. A transition variable t^i is set to true if the transition t is being executed at timestep i.

- $\{s^i \mid s \in \mathcal{S}, 0 \leq i \leq \kappa\}$ is a set of variables that encode states at timesteps. A state variable s^i is set to true if the extended state machine to which s belongs is in state s at timestep i.
- $\{sx^i \mid sx \in \mathcal{S}^*, 0 \leq i \leq \kappa\}$ is a set of variables that encode intermediate states at timesteps. If a state variable sx^i set to true, the extended state machine to which sx belongs is in state sx at timestep i.

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 $\mathsf{sm}(L)$ corresponding to a transition $t = (s, tr, \mathit{eff}, s')$ of the corresponding non-extended state machine, and let $m = (\sigma, a, \rho)$ be a message. Then, $\mathsf{trans}(L) \coloneqq T$, $\mathsf{src}(t) \coloneqq s$, $\mathsf{int}(t) \coloneqq s_t^*$, $\mathsf{trg}(t) \coloneqq tr$, $\mathsf{eff}(t) \coloneqq \mathit{eff}$, $\mathsf{tgt}(t) \coloneqq s'$, $\mathsf{snd}(m) \coloneqq \sigma$, $\mathsf{rec}(m) \coloneqq \rho$, and $\mathsf{symb}(m) \coloneqq a$.

The formulas φ^{gsc} and φ^{msc} are given by the conjunction of the subformulas φ_1 to φ_8 together with the two additional problem-specific formulas, $\varphi_{\text{init}}^{\text{gsc}}$ and φ_{goal}, and $\varphi_{\text{init}}^{\text{msc}}$ and φ_{seq}, respectively.

We now discuss the subformulas and their intuition. We start with subformula φ_1, which ensures that whenever a transition is executed at some timestep i, then the extended state machine changes from the transition's source state at timestep i to its target state at timestep $i+1$, the trigger symbol is set to false, and the effect symbol is set to true.

$$\varphi_1 \coloneqq \bigwedge_{i=0}^{\kappa-1} \bigwedge_{t \in \mathcal{T}} \Bigg[t^i \rightarrow \Big(\mathsf{src}(t)^i \wedge \mathsf{int}(t)^{i+1} \wedge \Big(\mathsf{trg}(t)^i \neq \epsilon \rightarrow \Big(\mathsf{trg}(t)^i \wedge \overline{\mathsf{trg}(t)}^{i+1} \Big) \Big) \wedge \Big(\mathsf{eff}(t)^i \neq \epsilon \rightarrow \Big(\overline{\mathsf{eff}(t)}^i \wedge \mathsf{eff}(t)^{i+1} \Big) \Big) \Big) \Bigg]$$

The next two formulas, φ_2 and φ_3, restrict the truth values of the effect symbols. φ_2 ensures that if an extended state machine does not leave its intermediate state, then the effect symbol remains true and φ_3 ensures that if the machine leaves its intermediate state, then the effect symbol is set to false at the following timestep.

$$\varphi_2 \coloneqq \bigwedge_{i=0}^{\kappa-1} \bigwedge_{\substack{t \in \mathcal{T}, \\ \mathsf{eff}(t) \neq \epsilon}} \Big[\mathsf{int}(t)^i \wedge \mathsf{int}(t)^{i+1} \rightarrow \mathsf{eff}(t)^{i+1} \Big]$$

$$\varphi_3 \coloneqq \bigwedge_{i=0}^{\kappa-1} \bigwedge_{\substack{t \in \mathcal{T}, \\ \mathsf{eff}(t) \neq \epsilon}} \Big[\mathsf{int}(t)^i \wedge \overline{\mathsf{int}(t)}^{i+1} \rightarrow \overline{\mathsf{eff}(t)}^{i+1} \Big]$$

The state following an intermediate state is always the target state of the respective transition in the non-extended state machine. The subformula φ_4 ensures that if the machine is in an intermediate state at timestep i and the effect has been consumed at timestep $i+1$, then at timestep $i+1$ the extended state machine leaves the intermediate state and changes into the target state of the transition.

$$\varphi_4 \coloneqq \bigwedge_{i=0}^{\kappa-1} \bigwedge_{t\in\mathcal{T}} \left[\left(\mathsf{int}(t)^i \wedge \left(\mathsf{eff}(t)^{i+1} \neq \epsilon \rightarrow \overline{\mathsf{eff}(t)}^{i+1} \right) \right) \rightarrow \left(\overline{\mathsf{int}(t)}^{i+1} \wedge \mathsf{tgt}(t)^{i+1} \right) \right]$$

The three subformulas φ_5 to φ_7 are called *framing axioms*. They ensure that there always is a transition that is "responsible" for changes of symbols (φ_5 and φ_6) or changes of states (φ_7). They also ensure that instances of state machines not participating in a transaction do not change their states.

$$\varphi_5 \coloneqq \bigwedge_{i=0}^{\kappa-1} \bigwedge_{trg\in\mathcal{A}} \left[trg^i \wedge \overline{trg}^{i+1} \rightarrow \left(\left(\bigvee_{t\in\mathcal{T}, trg=\mathsf{trg}(t)} t^i \right) \wedge \bigwedge_{\substack{t_1,t_2\in\mathcal{T},\\ \mathsf{trg}(t_1)=\mathsf{trg}(t_2)=trg}} \left(\overline{t_1}^i \vee \overline{t_2}^i \right) \right) \right]$$

$$\varphi_6 \coloneqq \bigwedge_{i=0}^{\kappa-1} \bigwedge_{\mathit{eff}\in\mathcal{A}} \left[\overline{\mathit{eff}}^i \wedge \mathit{eff}^{i+1} \rightarrow \left(\left(\bigvee_{t\in\mathcal{T}, \mathit{eff}=\mathsf{eff}(t)} t^i \right) \wedge \bigwedge_{\substack{t_1,t_2\in\mathcal{T},\\ \mathsf{eff}(t_1)=\mathsf{eff}(t_2)=\mathit{eff}}} \left(\overline{t_1}^i \vee \overline{t_2}^i \right) \right) \right]$$

$$\varphi_7 \coloneqq \bigwedge_{i=0}^{\kappa-1} \bigwedge_{s\in S} \left[s^i \wedge \overline{s}^{i+1} \rightarrow \left(\bigvee_{\substack{t\in\mathcal{T},\\ s=\mathsf{src}(t)}} t^i \right) \right]$$

φ_8 expresses that each machine is in exactly one state at each timestep.

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

The subformula $\varphi_{\text{init}}^{\text{gsc}}$ initializes the path by setting the variables representing states contained in the global state $\hat{s} = (x_1, \ldots, x_l)$, at timestep 0 to true, and all other variables representing states or symbols at timestep 0 to false. Similarly, $\varphi_{\text{init}}^{\text{msc}}$ sets the variables representing the initial states of the extended state machines at timestep 0 to true and all other variables at timestep 0 to false. This means that at timestep 0, all extended state machines are in their state as specified by $\hat{s}$ for k-GSC or in their initial states for k-MSC, and no

symbol is ready to be consumed.

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

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

To φ^{gsc} we add the subformula φ_{goal} encoding the (partial) global state $\hat{s}_p = (g_1, \ldots, g_l)$ at timestep k. It sets the states contained in the goal state together with their environments (cf. Definition 3).

$$\varphi_{\text{goal}} \coloneqq \bigwedge_{i=1, g_i \neq ?}^{l} \left(g_i^k \vee \bigvee_{s \in \mathsf{environ}(g_i)} s^k \right)$$

To φ^{msc} we add the subformula φ_{seq} encoding the sequence of messages to be executed after k steps. It first sets the symbol of each message first to true and in the subsequent timestep to false, then it ensures the state changes of intermediate states, and finally, it ensures that no other messages are received during the execution of the sequence diagram. Note that the changes of the transition variables are being taken care of by the framing axioms.

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

During the transformation phase, the expressions $\mathsf{trg}(t)^i \neq \epsilon$ and $\mathsf{eff}(t)^i \neq \epsilon$ occurring in the formulas are replaced by the corresponding logical constants $\top$ and $\bot$. Furthermore, the formulas are converted to conjunctive normal form, the input format of most SAT solvers. To this end, we apply the Tseitin transformation [31] 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 condition of both problems. If at x timesteps nothing happens and the execution of the message sequence starts at index k, it means that the length of the transaction sequence executed before timestep k is of length $k-x$.

In summary, the following formula encodes k-GSC:

$$\varphi^{\text{gsc}} \coloneqq \varphi^{\text{gsc}}_{\text{init}} \wedge \bigwedge_{i=1}^{8} \varphi_{\text{i}} \wedge \varphi_{\text{goal}}$$

and the following formula encodes k-MSC:

$$\varphi^{\text{msc}} \coloneqq \varphi^{\text{msc}}_{\text{init}} \wedge \bigwedge_{i=1}^{8} \varphi_{\text{i}} \wedge \varphi_{\text{seq}}$$

Here we see that the two different problems are very closely related, which also indicates that various other consistency checking problem can be encoded in a similar manner.

For k-MSC, a solution returned by the SAT solver consists of a satisfying assignment that sets the variables to true or false. By extracting the variables set to true which represent states and transitions (variable 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 and the state changes of the state machine instances during the execution of the sequence diagram. If the length of the path is less than k, then for some consecutive timesteps the state variables represent identical states. If a transformed instance of k-GSC is handed over to the SAT solver, the information provided by the SAT solver is similar.

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 [29].

6. Implementation

We implemented a tool to solve both k-GSC instances and k-MSC instances based on the SAT encodings presented above as plugin for the Eclipse framework[1]. It can be downloaded from:

`http://modelevolution.org/updatesite/`

6.1. Backend

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)[2]. This metamodel (cf. Fig. 4)

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

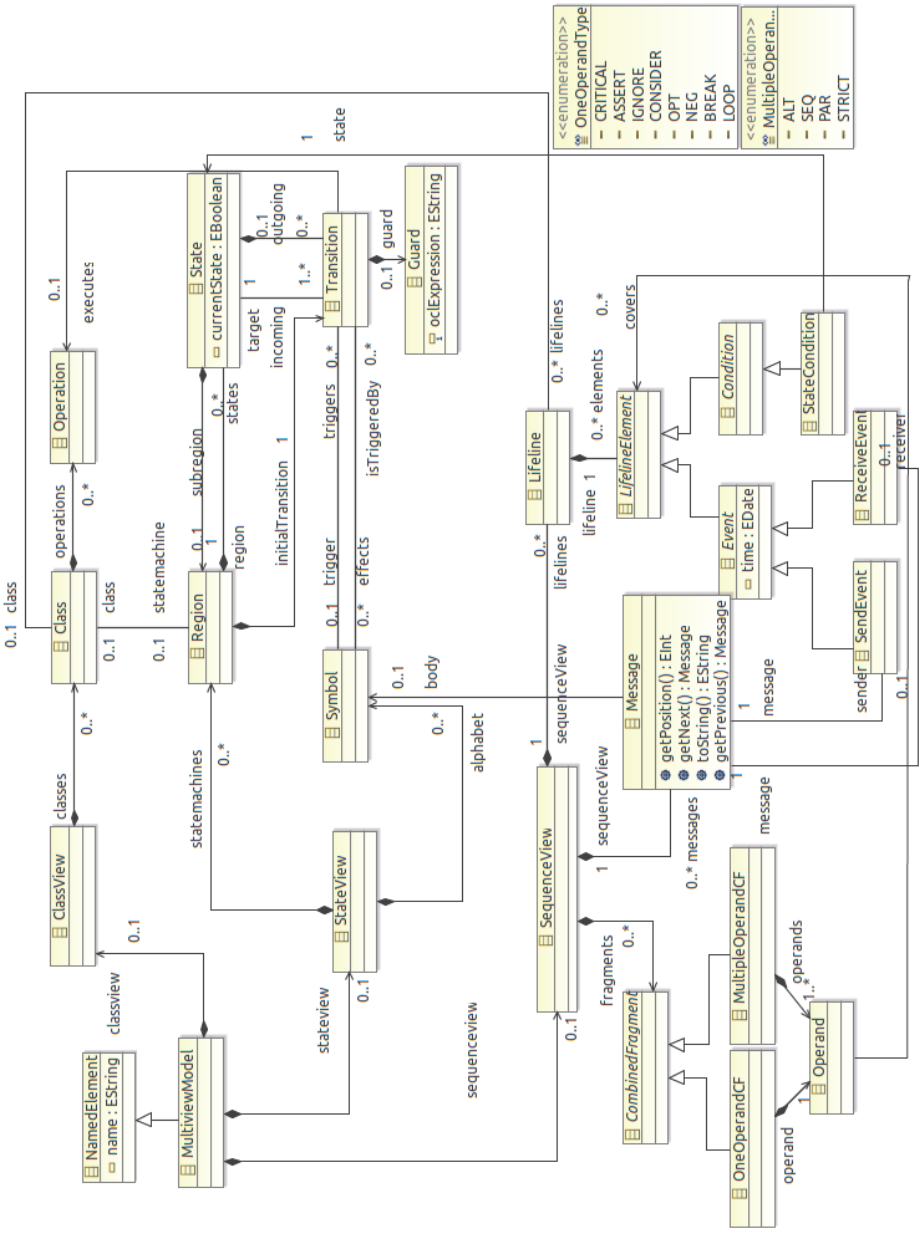

Figure 4: Metamodel in Ecore of our modeling language.

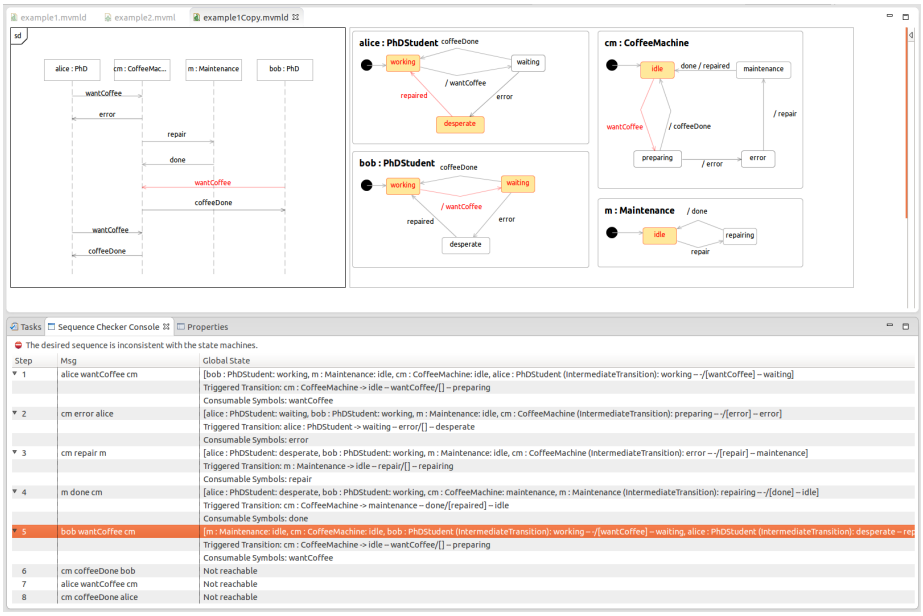

Figure 5: Screenshot of the graphical user interface.

contains all language concepts discussed in this paper as well as language concepts for future extensions.

The input models provided by the user of our tool are automatically translated to propositional logic using the encodings described above. Notice that some benchmarks require multiple effects per transition, which is also supported by our implementation, but not described in the sections above. After the encoding phase, the obtained formula is passed to the solver Sat4j [20], a Java-based SAT solver, integrated in our tool.

If the SAT solver returns `SAT`, then the considered partial global state is reachable by a path of length at most k (k-GSC) or at least one execution path of the state machines exists which conforms to the message sequence in the sequence diagram after an "initial" path of length at most k (k-MSC). If the SAT solver returns `UNSAT` then for the given path length k, the considered partial global state is not reachable or the state machines and the sequence diagram are inconsistent.

6.2. Frontend

Our tool provides a graphical editor for modeling state machines and sequence diagrams according to the metamodel shown in Fig. 4. In order to solve an instance of k-GSC, the user selects the states belonging to the (un)desired partial global state directly in the diagram and starts the checking process via a menu item. The user also provides the maximum bound (k) for the number of transactions to be considered. If not specified otherwise, the search starts

from the global initial state, i.e., the global state where each state machine is in its initial state. If the selected partial global state is reachable from the global initial state, a message sequence is returned showing the path from the global initial state to the respective partial global state. At the moment, this sequence is shown in form of a list, but principally it could also be visualized in the form of a sequence diagram.

The solving process of an instance of k-MSC can be initiated as soon as the necessary models are entered. Again, selecting a menu item starts the checking process.

In both cases, 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. 5. 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.

To sum up, our tool provides a graphical modeling environment, in which both verification approaches are fully integrated. Hence, the modelers neither have to deal with low-level SAT encodings nor with any other formal concept.

7. Evaluation

In order to test correctness and scalability of our tool we considered randomly generated instances and crafted instances. In the following, we describe both approaches. All benchmark instances are available on our project web site:

`http://modelevolution.org/prototypes/gsc`

7.1. Testing with random instances

We thoroughly tested our tool using a grammar-based white-box fuzzing approach [34]. This method generates random input models based on a grammar derived from the EMF metamodel shown in Fig. 4. We employed a random input model generator based on the tool presented in [34], which we adopted for the two problems presented in this paper. Other than taking into account only the receive events of a message sequence as in [34], we consider both the send and the receive events in the experiments presented below. The tool consists of two components, a *generator* to build syntactically correct diagrams, and a *simulator* to ensure that a message sequence can indeed be executed after a certain number of steps and that a given partial global state is indeed reachable.

We applied white-box fuzzing for debugging and performance evaluation purposes of our SAT encoding of k-GSC and k-MSC. In the following, we shortly describe the random generation of instances.

For instances of both problems, first, a set of state machines is built based on the following parameters.

- nrStateMachines: Number of state machines to be created.

- minNrStates and maxNrStates: Bounds on the number of states per state machine.
- minNrTrans and maxNrTrans: Bounds on the number of transitions per 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.
- probEff: The probability of a transition to contain an effect symbol.

For each state machine, the algorithm uniformly at random chooses a number of states and transitions in between the bounds minNrStates, maxNrStates and minNrTrans, maxNrTrans, and connects the states with transitions randomly in a way such that no state is isolated. To at least one outgoing transition from the initial state, the trigger ϵ is added. To all other transitions, a trigger different to ϵ is added with probability probTrigger. To each transition containing ϵ as trigger an effect 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 uniformly at random out of the alphabet.

7.1.1. Random instance generation for k-GSC

We build random instances of the k-GSC problem by generating sets of state machines as described above and then randomly choosing a partial global state. The additional parameter relPartial defines the size of the partial global state (goal state) relative to the number of state machines. All instances use the global initial state (initial states over all state machines) as global state (state from where the path starts).

7.1.2. Random instance generation for k-MSC

We build random instances of the k-MSC problem by adding a sequence diagram to a previously generated set of state machines. Here the following parameters are additionally considered.

- nrLifelines: The number of lifelines to be contained in the sequence diagram.
- nrMessages: The number of messages to be contained in the sequence diagram.

For each lifeline, a state machine is chosen uniformly at random from the state machine view. If nrLifelines $>$ nrStateMachines then it is ensured that each state machine is instantiated at least once. In order to ensure consistency, the simulator keeps track of the visited global states of the lifelines' state machines. 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 instantiated by the lifeline as value. For each lifeline, the hashmap is initialized with all initial states of the respective state machine All admissible messages from these states are calculated according to the current global state stored in the simulator. One message is chosen uniformly at random, appended to the message sequence, and the simulator is updated according to all possible successor states with respect to the application of the chosen message. To obtain unsatisfiable instances, we generate one more message than required and remove one message uniformly at random among all messages except the first one. This procedure, however, still can result in a satisfiable instance because there can be a different path than the one followed by the simulator. Further, note that 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.

Note that the state machines are non-deterministic, and therefore the number of possible states and admissible messages can become very large.

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

7.2. Testing with crafted instances

We designed three different benchmark sets to test our implementations of both problems on more natural and intuitive examples. In particular, we considered a coffee machine instance, which is similar to our running example above. We further designed a variant of the well-known dining philosophers problem and a simplified variant of the SMTP protocol (used in email transmission). For each instance we created a correct and an erroneous version. For all test cases, the starting state of the path was set to the global initial state, i.e., the global state where each state machine is in its initial state.

In the case of k-GSC, for instances "coffee" and "mail", we considered each possible combination of states for each state machine, and for the instance "philosophers" a meaningful selection of reachable and unreachable combinations of states was considered. In the case of k-MSC, we considered a consistent sequence diagram and an inconsistent sequence diagram for each benchmark set.

7.3. Results for the k-GSC problem

In the following, we report on the results of our evaluation based on randomly generated instances and the crafted set of instances regarding our implementation for solving the k-GSC problem. All experiments were executed on a computer with an Intel Core i5-540M CPU with 2.53GHz and 8GB of RAM.

7.3.1. Random instances

We group the randomly generated instances into three different sets according to their size. The following parameters are set to the same values for all instances. The value of nrSymbols is set to minNrStates, probTrigger and probEff are set to 0.9, and k is set to maxNrStates. Table 4 shows the values of the remaining parameters for each group. Parameters minNrTrans and maxNrTrans depend on the parameters minNrStates and maxNrStates, respectively, in that their value is multiplied by 3 for the sets "small" and "medium" and multiplied by 4 in the set "large". For parameter relPartial we test the three values .25, .5, and 1 for each category. It describes the size of the goal state relative to nrStateMachines. The size of the goal state is determined by the product relPartial · nrStateMachines.

	small	medium	large
minNrStates	2	4	6
maxNrStates	3	6	9
minNrTrans	6	12	24
maxNrTrans	9	18	36
nrStateMachines	8	15	30

Table 1: Parameter settings for the random instances of k-GSC.

Table 2 describes the results of our experiments over 1,000 randomly generated instances in each category except for "large" with only 100 instances due to time constraints, and for each value for relPartial.

	small			medium			large		
relPartial	.25	.5	1	.25	.5	1	.25	.5	1
Enc. time `SAT` (ms)	7	7	5	97	93	n/a	1,543	1,589	n/a
Enc. time `UNSAT` (ms)	6	6	5	100	100	102	1,579	1,595	1,517
Solv. time `SAT` (ms)	1	2	3	32	46	n/a	17,569	9,570	n/a
Solv. time `UNSAT` (ms)	<1	<1	<1	11	8	4	4,378	2,740	1,142
Nr. of clauses	1,821	1,831	1,816	56,884	57,235	56,955	846,798	866,320	843,238
Nr. of variables	375	376	375	3,490	3,497	3,498	19,427	19,507	19,429
Nr. of instances `SAT`	336	80	4	93	7	0	17	1	0
Nr. of instances `UNSAT`	664	920	996	907	993	100	83	99	821

Table 2: Average results over 1,000 runs for each category (100 runs for "large") for the random instances of k-GSC.

We distinguish encoding and solving times by `UNSAT` and `SAT` instances. It can be seen that the solving time for `UNSAT` instances is considerably lower than for `SAT` instances. Also, the solving time scales worse than the encoding time for `SAT` instances, which is an expected behavior given the complexity of the problem. However, this is not the case for `UNSAT` instances. One possible explanation for this is that `UNSAT` instances are likely to contain a contradiction that is easy to find for the SAT solver Sat4j, which we are employing.

As expected, the number of `SAT` instances decreases with increasing values

Bound	3			15			100			500		
Time	enc.	sol.	total	enc.	sol.	total	enc.	sol.	total	enc.	sol.	total
mail	8	1	9	5	15	20	34	1,173	1,207	201	63,043	63,244
coffee	13	3	16	12	24	36	28	2,165	2,193	143	39,291	39,434
philosophers	18	3	21	21	44	65	43	21,192	21,235	-	-	-

Table 3: Average runtimes in ms over different goal states on crafted instances of the k-GSC.

for relPartial since it becomes harder to find a global state that is fully specified than one that is only partially specified. Also, the solving times decrease with increasing values for relPartial, possibly due to the additional constraints given to the SAT solver. The number of clauses and the number of variables increase considerably, however, up to the instances in the "large" set, this can still be handled reasonably by the SAT solver.

We conclude that that the overall runtimes on our randomly generated instance set are good even if executed on standard hardware.

7.3.2. Crafted instances

We executed the crafted instances of k-GSC with the value of k set to 3, 15, 100, and 500. Details on the outcomes of the test cases are presented in Table 3. The results of all test cases are as expected. All bugs in the erroneous versions were found. The approach performs well up to a bound of $k = 500$. As can be expected, the bottleneck for higher bounds is the task of solving the SAT instance.

7.4. Results for the k-MSC problem

In the following, we report on the results of our evaluation based on randomly generated instances and the crafted set of instances regarding our implementation for solving the k-MSC problem. Same as for the k-GSC problem, all experiments were executed on a computer with an Intel Core i5-540M CPU with 2.53GHz and 8GB of RAM.

7.4.1. Random instances

Same as for the k-GSC problem, we group the randomly generated instances according to different parameter sets into three different groups depending on their size. Except for nrStateMachines, all parameter settings are the same as for k-GSC. The value of nrStateMachines is 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. Table 4 shows the values of the parameters whose values differ between the groups.

Table 5 describes the results of our experiments over 1,000 randomly generated instances in each category.

Same as for the k-GSC problem, 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

	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

Table 4: Parameter settings for the random instances of the k-MSC problem.

	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

Table 5: Average results over 1,000 runs for the random instances of the k-MSC.

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. As can be expected, the solving time scales worse than the encoding time.

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 few 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 of the one followed by the simulator because of the previous requirement to have many transitions and few symbols.

Same as for the k-GSC problem, even though the number of variables and the number of clauses increase considerably from the small to the large instances, the SAT solver still manages to solve the instances with acceptable runtimes.

We conclude that the overall runtimes for our randomly created instances are good even if executed on standard hardware. The overall runtime for `UNSAT` instances can 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.

Bound	3			15			100			500		
Time	enc.	sol.	total	enc.	sol.	total	enc.	sol.	total	enc.	sol.	total
mail `SAT`	488	129	618	465	232	697	1,314	1,161	2,475	2,722	2,862	5,584
coffee `SAT`	229	45	274	296	125	421	600	661	1,261	1,419	2,585	4,004
philosophers `SAT`	230	46	276	289	61	350	569	413	983	1,639	849	2,489
mail `UNSAT`	151	113	264	189	236	426	584	1,145	1,693	896	2,436	3,332
coffee `UNSAT`	28	89	117	53	218	272	114	577	691	243	2,519	2,863
philosophers `UNSAT`	198	32	230	308	75	383	692	460	1,153	2,681	1,304	3,923

Table 6: Runtimes for instances of k-MSC on `SAT` and `UNSAT` crafted instances.

7.4.2. Crafted instances

Same as for the k-GSC problem, we also tested our implementation on crafted instances of k-MSC with k set to 3, 15, 100, and 500. Details on the outcomes of the test cases are presented in Table 6. The results of all test cases are as expected. All bugs in the erroneous versions were found. Same as for the randomly generated instances of this problem, if the SAT solver returns `UNSAT`, we remove the last message of the sequence diagram repeatedly until `SAT` is returned. The approach performs well up to a bound of $k = 500$. Interestingly, other than for the crafted instances for k-GSC, the bottleneck for higher bounds is the task of encoding the SAT instance. It seems that the presence of the clauses encoding the sequence diagram allows the SAT solver to work very efficiently on these instances.

8. Conclusion and Future Work

We presented a novel SAT-based approach to solve an intradiagram consistency checking problem of state machines and an interdiagram consistency checking problem between state machines and sequence diagrams. To this end, we concisely formulated the formal semantics of the considered modeling language concepts. On this basis we were able to obtain an exact formal description of the consistency checking problems. This description allowed us to develop a transformation to SAT, the satisfiability problem of propositional logic. The SAT encodings reuse ideas and techniques well established for formulating planning problems.

We showed that few modifications enable us to apply the encoding originally designed for solving an intradiagram consistency checking problem to solve an interdiagram consistency checking problem. In a similar manner, the encoding can be applied for other consistency checking problems as well. Our encodings are flexible with respect to the semantics of the modeling language and efficiently processable. Furthermore, the information necessary to map the solutions obtained from the SAT solver back to the modeling environment is preserved allowing the realization of a verification approach which is tightly integrated with the modeling environment.

In future work, we plan to consider additional modeling language concepts like hierarchical states in the state machines or combined fragments and invari-

ants in the sequence diagrams. For example, the evaluation of guards controlling the execution of branches or loops may require to use stronger formalisms than SAT like SMT, which has successfully been used for the verification of dynamic properties of class diagrams [30]. In addition, 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 to automatic repair. In particular, the encoding can be modified such that missing messages in the sequence diagram can be determined. This scenario can occur in automated merging environments as, for instance, in model versioning systems [8]. Another important task is the optimization of the SAT encoding to further improve the performance.

Acknowledgements

This work was partially funded by the Vienna Science and Technology Fund (WWTF) under grant ICT10-018, by the Austrian Science Fund (FWF) under grants P25518-N23, S11408-N23, and S11409-N23, and the German Research Foundation (DFG) under grant ER 738/2-1.

References

[1] M. H. t. Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. A State/Event-based Model-Checking Approach for the Analysis of Abstract System Properties. *Science of Computer Programming*, 76(2):119–135, 2011.

[2] S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable Petri net models. In *Proc. of the 3rd International Workshop on Software and Performance (WOSP)*, pages 35–45. ACM, 2002.

[3] J. Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[4] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[5] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[6] P. Brosch, U. Egly, S. Gabmeyer, G. Kappel, M. Seidl, H. Tompits, M. Widl, and M. Wimmer. Towards semantics-aware merge support in optimistic model versioning. In *Proc. of the MoDELS Workshops*, volume 7167 of *LNCS*, pages 246–256. Springer, 2011.

[7] 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 International Conference on Tests and Proofs (TAP)*, volume 7305 of *LNCS*, pages 149–155. Springer, 2012.

[8] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. An introduction to model versioning. In *Formal Methods for Model-Driven Engineering*, volume 7320 of *LNCS*, pages 336–398. Springer, 2012.

[9] J. Dubrovin and T. A. Junttila. Symbolic Model Checking of Hierarchical UML State Machines. In *Proc. of the 8th International Conference on Application of Concurrency to System Design (ACSD)*, pages 108–117. IEEE, 2008.

[10] A. Egyed. Instant consistency checking for the UML. In *Proc. of the 28th International Conference on Software Engineering (ICSE)*, pages 381–390. ACM, 2006.

[11] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Testing the consistency of dynamic UML diagrams. In *Proc. of the 6th International Conference on Integrated Design and Process Technology (IDPT)*. Society for Design and Process Science, 2002.

[12] T. H. Feng and H. Vangheluwe. Case study: Consistency problems in a UML model of a chat room. In *Proc. of the Workshop on Consistency Problems in UML-based Software Development*, pages 18–26, 2003.

[13] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE)*, pages 37–54. IEEE Computer Society, 2007.

[14] S. Gabmeyer, P. Kaufmann, and M. Seidl. A classification of model checking-based verification approaches for software models. In *Proc. of STAF Workshop on Verification of Model Transformations (VOLT)*, pages 1–7, 2013.

[15] B. Graaf and A. van Deursen. Model-driven consistency checking of behavioural specifications. In *Proc. of the 4th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, pages 115–126, 2007.

[16] P. Kaufmann, M. Kronegger, A. Pfandler, M. Seidl, and M. Widl. Global state checker: Towards SAT-based reachability analysis of communicating state machines. In *Proc. of the 10th Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa)*, number 1069 in CEUR Workshop Proceedings, pages 31–40, 2013.

[17] P. Kaufmann, M. Kronegger, A. Pfandler, M. Seidl, and M. Widl. A sat-based debugging tool for state machines and sequence diagrams. In *Proc. of the 7th International Conference on Software Language Engineering (SLE)*, volume 8706 of *LNCS*, pages 21–40. Springer, 2014.

[18] A. Knapp, S. Merz, and C. Rauh. Model Checking—Timed UML State Machines and Collaborations. In *Proc. of the 7th International Symposium on*

Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT), volume 2469 of *LNCS*, pages 395–416, 2002.

[19] V. S. Lam and J. Padget. Consistency Checking of Sequence Diagrams and Statechart Diagrams Using the π-Calculus. In *Proc. of the 5th International Conference on Integrated Formal Methods (IFM)*, volume 3771 of *LNCS*, pages 347–365. Springer, 2005.

[20] D. Le Berre and A. Parrain. The Sat4j Library, Release 2.2, System Description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[21] J. Lilius and I. Paltor. vUML: A Tool for Verifying UML Models. In *Proc. of the 14th IEEE International Conference on Automated Software Engineering (ASE)*, pages 255–258. IEEE, 1999.

[22] F. J. Lucas, F. Molina, and A. Toval. A systematic review of UML model consistency management. *Information and Software Technology*, 51(12):1631 – 1645, 2009.

[23] A. Niewiadomski, W. Penczek, and M. Szreter. Towards Checking Parametric Reachability for UML State Machines. In *Proc. of the International Andrei Ershov Memorial Conference*, volume 5947 of *LNCS*. Springer, 2010.

[24] I. Ober, S. Graf, and I. Ober. Validation of UML Models via a Mapping to Communicating Extended Timed Automata. In *Proc. of the 11th International SPIN Workshop—Model Checking Software*, volume 2989 of *LNCS*, pages 127–145. Springer, 2004.

[25] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure. Technical Report Version 2.3, OMG, May 2010.

[26] C. H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[27] J. Rintanen. Planning and SAT. In *Handbook of Satisfiability*, pages 483–504. IOS Press, 2009.

[28] B. Selic. What will it take? A view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, 2012.

[29] C. Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Proc. of the 11th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 3709 of *LNCS*, pages 827–831. Springer, 2005.

[30] M. Soeken, R. Wille, and R. Drechsler. Verifying dynamic aspects of uml models. In *Proc. of the Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6. IEEE, 2011.

[31] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

[32] M. Usman, A. Nadeem, T. Kim, and E. Cho. A survey of consistency checking techniques for UML models. In *Proc. of Advanced Software Engineering and Its Applications (ASEA)*, pages 57–62. IEEE, 2008.

[33] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML models. In *Proc. of the 6th International Conference on UML*, volume 2863 of *LNCS*, pages 326–340. Springer, 2003.

[34] M. Widl. Test Case Generation by Grammar-based Fuzzing for Model-driven Engineering. In *Proc. of the 8th International Haifa Verification Conference (HVC)*, volume 7875 of *LNCS*, pages 278–279. Springer, 2012.