FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Symbolic Methods for the Verification of Software Models

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor/in der technischen Wissenschaften

by

Magdalena Widl
Registration Number 0327059

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao. Univ.-Prof. Dr. Uwe Egly
Advisor: Assist.-Prof. Dr. Martina Seidl

The dissertation has been reviewed by

(Ao. Univ.-Prof. Dr. Uwe Egly) (Prof. Dr. Hans Kleine Büning)

Wien, January 8, 2016

(Magdalena Widl)

Erklärung zur Verfassung der Arbeit

Magdalena Widl
Favoritenstrasse 9-11, 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgments

During the past years, I was encouraged, influenced, and supported by a number of people in my professional and in my private environment. Their input was invaluable not only for the completion of this thesis but also for my professional and personal development. In particular, I am indebted to my advisors Uwe Egly and Martina Seidl for sharing their expertise and experience and for their guidance through the jungle of science up to a finalization of this work. I also thank my reviewer Hans Kleine Büning for his helpful comments and I thank Gerti Kappel for her support in scientific and administrative matters and for integrating me into her workgroup. I am very grateful to Jie-Hong Roland Jiang for hosting me at the National Taiwan University. My stay was a wonderful experience in many aspects.

I spent many good times, discussions, and brainstorming sessions with my co-authors Sebastian Gabmeyer, Petra Kaufmann, Martin Kronegger, Florian Lonsing, Andreas Pfandler, Hans Tompits, and Valeriy Balabanov. Thank you for your support, for your ideas, and for writing up papers with me. Many thanks also to Florian Zoubek for developing a graphical user interface for our tools, to Martin Lackner for a complexity theoretical hint, and to Reinhard Karzel for proofreading the German part of this work. I was also very fortunate to have assistance from our excellent administrative staff including Eva Nedoma, Matthias Schlögel, and Toni Pisjak.

I enjoyed the company of many other colleagues and friends. An unordered and incomplete list includes Magdalena Ortiz, Sarah Gaggl, Thomas Krennwallner, Peter Schüller, Nysret Musliu, Minh Dao-Tran, Antonius Weinzierl, Emanuel Sallinger, Marion Scholz, Irene Fastl, Angela Witzani, Ana Pizarroso, Sascha Vanicek, and Christof and Birgit Schmidt. Thank you.

Writing up this thesis included intensive work during a period of substantial change in my personal life. I thank my parents, my brother, my sister, and my parents-in-law for their unconditional support that enabled me to concentrate on my work. I am most fortunate to share with my husband José both joy and workload that come with founding a family. Thank you so much for your encouragement and for putting me at ease whenever needed. Thank you Nuria, our beautiful baby, for being just the way you are.

My work was funded by Vienna Science and Technology Fund (WWTF) under grant ICT10-018, by the Austrian Science Fund (FWF) under grant S11409-N23, and by grants from the Faculty of Informatics and the International Office of the Vienna University of Technology.

Abstract

One of the major challenges in modern software engineering is dealing with the increasing complexity of software systems. The new paradigm of model-driven engineering (MDE) promises to handle this complexity using the abstraction power of software models based on languages such as the Unified Modeling Language (UML). The objective of MDE is to generate executable code from a set of diagrams with little or no intervention of a developer. These diagrams are referred to as *model* of the system to be implemented. Usually *multi-view* models are employed, where each diagram shows a different view on the system, altogether providing a holistic representation. This shift from code-centric development to MDE requires a modeling language based on a solid formal semantics, which is considered one of the major current challenges in MDE research and in future improvements of the UML. Further, with this high valorization of software models, also stronger requirements on their consistency come along since errors introduced on the model level can result in faulty code. The models are central to the evolution of a software system and therefore undergo continuous and often parallel modifications that can introduce inconsistencies. However, necessary consistency management tasks are often too cumbersome to be performed manually due to the size of the models. Hence, automated methods are required.

In this work we formalize a modeling language based on the UML. Our language contains two views: the state machine diagram and the sequence diagram. We then identify three problems that can occur in models based on this language. The *Sequence Diagram Merging Problem* deals with merging two modified versions of a sequence diagram in a way that keeps them consistent with the set of state machines they are related to. The *State Machine Reachability Problem* asks whether a combination of states in a set of state machines can be reached from some global state by sending and receiving messages between the state machines. The *Sequence Diagram Model Checking Problem* asks whether a sequence diagram is consistent with the set of state machines it instantiates. For each problem, we propose an encoding to the satisfiability problem of propositional logic (SAT) in order to solve it with an off-the-shelf SAT solver and we determine its computational complexity. We evaluate our approaches based on a set of handcrafted models and on grammar-based whitebox fuzzing, for which we develop a random model generator. The results of our experiments show that we can solve instances of these problems of reasonable size on standard hardware with state-of-the-art SAT solvers.

Kurzfassung

Der Umgang mit der zunehmenden Komplexität moderner Softwaresysteme stellt eine der größten Herausforderungen der modernen Softwareentwicklung dar. Modellgetriebene Entwicklung (engl. model-driven engineering, MDE) verspricht einen einfacheren Umgang mit dieser Komplexität durch Nutzung der Abstraktionsfähigkeiten modellbasierter Sprachen wie zum Beispiel der Unified Modeling Language (UML). Das Ziel von MDE ist es von einer Menge an Diagrammen, dem *Modell* für das zu entwickelnde System, lauffähigen Programmcode ohne oder mit minimaler Zuhilfenahme eines Entwicklers automatisch zu generieren. Für diesen Zweck werden meist Modelle mit *mehrfachen Sichten* verwendet. Solche Modelle enthalten mehrere Arten von Diagrammen, wobei jede Art eine andere Sicht auf das System ermöglicht und die Kombination der Diagramme das System in seiner Gesamtheit beschreibt.

Dieser Wechsel des Fokus von textuellem Programmiercode auf die Softwaremodelle der MDE verlangt nach einer über eine solide formale Semantik verfügenden Modellierungssprache. Die Entwicklung einer solchen Modellierungssprache gilt als eine der größten Herausforderungen in der Forschung zu MDE und für zukünftige Verbesserungen der UML. Weiters bringt diese wesentliche Aufwertung der Softwaremodelle strengere Anforderungen bezüglich deren Konsistenz mit sich, da Fehler auf Modellebene sich in den Programmcode weiterpropagieren. Ihre neue Führungsrolle im Entwicklungsprozess stellt die Modelle ausserdem in den Mittelpunkt der Sofwareevolution, wodurch sie häufigen und oftmals parallelen Änderungen ausgesetzt sind. Die damit entstehenden Aufgaben im Bereich des Konsistenzmanagements, des Testens und der Fehlerbehebung sind bedingt durch die Größe der Modelle zu komplex um manuell durchgeführt zu werden. Automatische Methoden sind daher unumgänglich.

In dieser Arbeit formalisieren wir eine Modellierungssprache, die auf der UML basiert und zwei Sichten beinhaltet, nämlich das Zustandsdiagramm und das Sequenzdiagramm. Wir identifizieren drei Probleme, die in auf dieser Sprache basierenden Modellen auftreten können. Das *Sequence Diagram Merging Problem* befasst sich mit der Integration von parallel an einem Sequenzdiagramm durchgeführten Änderungen mit dem Ziel, ein mit einer Menge von Zustandsdiagrammen konsistentes Sequenzdiagramm zu erstellen. Das *State Machine Reachability Problem* beschreibt, ob eine Kombination aus in verschiedenen Zustandsdiagrammen enthaltenen Zuständen von einem gegebenen globalen Zustand erreichbar ist. Das *Sequence Diagram*

Model Checking Problem fragt, ob ein Sequenzdiagramm konsistent ist mit der Menge an Zustandsdiagrammen, die es instanziiert. Für jedes der drei Probleme erstellen wir eine Kodierung als aussagenlogisches Erfüllbarkeitsproblem, das mit einem gängigen, frei verfügbaren Solver gelöst werden kann, und ermitteln seine computationale Komplexität. Wir evaluieren unseren Lösungsansatz mit handgefertigten Instanzen und mittels einer grammatikbasierenden White-Box-Fuzzing-Methode, wofür wir einen randomisierenden Modellgenerator entwickeln. Die Resultate unserer Experimente zeigen, dass Instanzen unserer Probleme von brauchbarer Größe von einem gängigen Solver auf gängiger Hardware in vertretbarer Zeit gelöst werden können.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There is unlikely to exist a silver bullet, says Frederick Brooks in his 1987 seminal paper [20], to conquer the increasing complexity of software. The term "complexity", however, is rather overloaded even if restricted to the context of computer science, and it concerns software in many different ways. Brooks' statement is certainly correct regarding complexity in the algorithmic sense: Unless P equals NP, nothing can tame the computational complexity of a piece of software trying to solve one of the vast number of NP-hard problems, many of them very relevant in practice. Other uses of the term "complexity" concerning software systems describe the difficulty of representing the problem domain or the complicated interaction of large numbers of data items.

Brooks introduces two categories of complexity for software systems, namely *essential complexity* and *accidental complexity*. The former concerns the essence of the software and the latter deals with its representation in a programming language. He conjectures that essential complexity not only forms a significantly larger portion of the complexity of a software system but that is also very hard to reduce.

Brooks attributes the irreducibility of essential complexity to four properties, among them the property of state space complexity and the property of unvisualizability. There cannot be much doubt about the former property, however the latter property is being more and more invalidated by the advances of software and hardware technology. Indeed, his two arguments against "graphical programming", by which he refers to visual representations of software, being a silver bullet to eliminate some essential complexity, are the poor expressiveness of the then state-of-the-art visual tool of flow charts and the too small size of the screens to display a visual representation.

Five years later, David Harel, father of the *state charts*, which are a visual formalism to represent software systems, responds to Brooks' arguments with a strong focus on the then state of the art of software modeling [64]. He says

> "A 'vanilla' approach to modeling, together with powerful notions of executability and code generation, may have a profound impact on the essence of developing complex systems."

His line of argument is based on an analogy between the complexity removed by high-level programming languages from one-person programs previously written in assembler languages and the complexity of the in 1992 increasingly complex software, summarized as *reactive systems*, which he claims can be removed by visual programming formalisms.

Indeed, more than 20 years later, visualization of software is almost omnipresent. It is employed in various stages of hardware and software development and for many different purposes of formal or informal nature. Models are used as informal drafts, for communication with clients, for documentation purposes, and, with the relatively new development paradigm of model-driven engineering (MDE), as primary development artifacts from which executable code is derived [19]. Also, several modeling techniques more sophisticated than the flow chart arose during the 1980s and 1990s, many of them linked to the development paradigm of object-oriented programming, which was then gaining more and more importance.

Soon various efforts were taken to unify and standardize these developments in modeling, eventually resulting in the *Unified Modeling Language* (UML) in its first version of 1997, which was significantly revised and extended to a new version in 2005, and is currently in its version 2.4.1 [60, 106]. The UML is a *multi-view* modeling language consisting of a set of diagrams to depict a system from different angles, with different levels of abstraction, and with each diagram focusing on either a static or a dynamic aspect of the system. The information of a software system is spread over different complementary diagrams where similar diagrams make up a *view*. Some information can be redundant with respect to different views. The UML is widely employed, particularly in industry, to support software engineering processes. However, in the research community of software modeling and MDE, more and more criticism appears regarding the UML's lack of formality [27, 65, 66].

In MDE, multi-view software models take over an important role as core development artifacts in order to deal with the complexity of modern software systems [13]. The objective is to generate executable code directly from the models with little or no intervention of a developer [109]. This shift from code-centric development to MDE thus requires a modeling language based on a solid formal semantics, which is considered one of the major current challenges in MDE research [52] and in future improvements of the UML [51].

With this high valorization of software models, also stronger requirements on their consistency come along since errors introduced on the model level in the worst case result in faulty code. Performing a leading role in the development process, the models are also central to the evolution of a software system and undergo continuous and often parallel modifications. Modifications are handled in activities like synchronization, versioning, and co-evolution, each of them demanding change propagation between the different views in order to keep them consistent. Due to the size of the models, their consistency management, testing, and debugging are tasks too cumbersome to be done manually [109]. Hence, automated methods are required to support the mentioned evolution tasks [52]. Interestingly, this necessity was already predicted by Harel in his 1992 response to Brooks' position paper [64].

Techniques based on formal methods, which have been around for over 30 years for tex-

tual code [34], recently found their way into MDE [53, 116]. In particular, approaches based on model checking [33] have gained much popularity in software verification during the past 20 years. Model checking aims at verifying a system with respect to a specification by exhaustive state traversal and returns an execution trace violating the specification if such an execution trace exists. The system is represented by finite automata or similar models. The major challenge of model checking is to handle the exponential number of states in systems, also known as the *state explosion problem* [33]. One of the most successful approaches to this problem is *symbolic model checking* [32], which was initially based on binary decision diagrams [93]. *Bounded model checking*, by putting a bound on the length of the execution trace, later led to compact encodings to the satisfiability problem of propositional logic (SAT) [15].

Other symbolic approaches to bounded model checking include encodings to the satisfiability problem of *quantified Boolean logic* (QSAT) [38, 71]. Quantified Boolean formulas (QBF) are an extension of propositional formulas that allows to universally or existentially quantify propositional variables. The quantification lifts the satisfiability problem of QBF to PSPACE-completeness (from the NP-completeness of SAT) [99]. Therefore, encodings of bounded model checking to QSAT can be more compact than to SAT. Several solvers are available for this logic [10,56,69,88], but so far, QBF solving methods still seem to be too immature to be applied on an industrial scale. Apart from efficient solving, also the generation of concrete solutions for a formula is likely to be significantly harder than for SAT. Our recent contributions in this area [5,6,41,42] can lead to more efficient methods for the generation of solutions to QBFs and hence make QBF a more practical host language for PSPACE-complete problems.

Consistency problems occurring in the evolution of software models often include model checking problems. For example, a sequence diagram can be regarded as a specification for a system modeled by a set of state machines [21, 22]. Several translations to input languages of model checkers, mainly to the explicit (not symbolic) model checker SPIN have been proposed [21, 40, 68, 79, 80, 86, 98, 101, 108, 112]. However, most of their implementations do not seem to have gone beyond a prototypical state or were never updated to deal with newer versions of the UML. A reason for this could be the semantic differences between the software models and the model checker's input language, which make equivalence preserving translations very challenging [21, 22].

The challenge of overcoming these semantic heterogeneities raises the question whether this translation step is really required and how a low-level encoding, for example to propositional logic, would perform. At least one work [97] presents positive results regarding this question by proposing an encoding to propositional logic for a reachability analysis of state machines and by showing it to be more computationally efficient than translations to standard model checkers for three examples.

Further research directions in this area therefore suggest symbolic encodings of similar problems on the one hand and into different languages on the other hand. The most popular language is propositional logic, but QBFs provide another attractive language in particular when it comes to finding compact encodings for PSPACE-complete problems.

In this work, we introduce a definition of a formal semantics for a software modeling language and identify three model consistency problems that can occur in different model management tasks during the evolution of software models expressed in the formalized language.

We then suggest symbolic approaches to solving these problems and present an in-depth performance evaluation of these approaches based on handcrafted instances and on grammar-based whitebox fuzzing. We finish this work with an outlook to future work on solving more complex problems with symbolic encodings to QBF.

1.1 Contributions

The contributions of this work are threefold. They deal with a formal description of the semantics of a subset of the UML, with a set of symbolic approaches to three consistency problems occurring within these semantics, and an evaluation of these approaches based on grammar-based whitebox fuzzing, Most of our results have been published at conferences, workshops, and in journals.

1.1.1 Formalization of a Modeling Language

The use of software models in areas likes MDE requires a formal definition of their semantics. In particular, a formalization of model concepts is indispensable in order to identify and define consistency problems that can occur among them. We consider two UML-derived views of a software model, the sequence diagram and the state machine. Both adopt fundamental concepts of the respective UML views. We first define their syntax by means of a UML metamodel and then we describe the classes of the metamodel in a mathematical notation in order to formally define their semantics in terms of a set of intra-view and inter-view properties. We provide formal definitions of these properties by a set of formulas in the language of propositional logic. We presented most of these contributions at the 5th Conference on Software Language Engineering [124], in an article in "Softwaretechnik-Trends" [26], at the 10th Workshop on Model-Driven Engineering, Verification, and Validation [72], at the 7th Conference on Software Language Engineering [74], and in an article in "Computer Languages, Systems & Structures" [73]. Our work received the Best Paper Award at the 7th Conference on Software Language Engineering.

1.1.2 Verification Problems in Software Modeling

We identify three problems that can occur during the evolution of a software model, show that they are in the complexity class P, respectively NP-complete, and propose symbolic encodings to solve them. First, the *Sequence Diagram Merging Problem* deals with producing a new version of a sequence diagram out of two modified versions in the context of optimistic model versioning, where the new version must fulfill some requirements regarding semantic consistency with the set of state machines that the lifelines of the sequence diagram instantiate. Solutions to this problem directly assist the developer with the model management task of versioning. Second, we deal with two verification problems whose automated resolution supports different stages of the model evolution. Both problems ask whether a certain specification of the system is fulfilled. In the *State Machine Reachability Problem*, a specification is given by a combination of states, each from a different state machine. In the *Sequence Diagram Model Checking Problem*, a specification is given by a sequence diagram. Both problems ask whether a set of state machines implements the specification. We presented most of these contributions at the 5th Conference

on Software Language Engineering [124], in an article in "Softwaretechnik-Trends" [26], at the 10th Workshop on Model-Driven Engineering, Verification, and Validation [72], at the 7th Conference on Software Language Engineering [74], and in an article in "Computer Languages, Systems & Structures" [73].

1.1.3 Experimental Evaluation by Grammar-based Whitebox Fuzzing

We provide prototype implementations of the symbolic approaches described in the previous contribution within the Eclipse Modeling Framework (EMF) [49]. Since real-life test cases are difficult to obtain and often cover only very specific areas of the problem space, we provide a set of handcrafted models from which instances of our problems can be derived, and further suggest a randomized method to generate artificial scenarios in order to facilitate fuzz-testing of our tools. This method is easily extensible to be applied to other tools dealing with Ecore models of the EMF.

We use our handcrafted models to evaluate all approaches, apply fuzz-testing to two of our prototype implementations, and report on the scalability of our approaches based on the results. We presented our method of grammar-based whitebox fuzzing for modeling tools at the 8th Haifa Verification Conference [123], its application to the *Sequence Diagram Model Checking Problem* at the 7th Conference on Software Language Engineering [74], and its application to the *State Machine Reachability Problem* in an article in "Computer Languages, Systems & Structures" [73].

1.2 Outline

The structure of this thesis follows roughly the contributions described above. After introducing some preliminaries and basic notions, we devote a chapter to a precise definition of the software models we are dealing with. Based on these definitions, we present a set of consistency problems and methods based on propositional logic to solve these problems in the following chapter. We then dedicate a chapter to an in-depth evaluation of our approach. After drawing the conclusions of this work, we suggest some future work that combines our contributions of this work and some of our recent contributions in the area of QBF certification.

Chapter 2, Preliminaries. In this chapter we first introduce some basic formal notions and terminology used in this thesis. Then we present the syntax and semantics of propositional logic and give some information on the syntax and semantics of the UML.

Chapter 3, The Tiny Multiview Modeling Language (*tMVML*). We introduce the Tiny Multiview Modeling Language (*tMVML*), the language of the software models used in this thesis. We devote three sections to its definition. First, we describe the abstract syntax of the language as a UML metamodel. Then we derive a formal description of the *tMVML* from the metamodel, and finally, we describe the behavioral aspects of the *tMVML* along with some semantic properties. Parts of this chapter are based on our publications [26, 72–74, 124].

Chapter 4, Verification Problems in Software Modeling. This chapter is devoted to software model consistency problems in the two areas of model versioning and model verification. We first review related work for both fields and then present three problems, one of them in the former, and two of them in the latter field. For each problem, we give an intuitive example before formally describing the problem. We then show how the problem can be solved by an encoding to propositional logic and prove its computational complexity. Parts of this chapter are based on our publications [26, 72–74, 124].

Chapter 5, Evaluation. We describe the implementations of the encodings presented in Chapter 4 within the Eclipse Modeling Framework [49], our set of handcrafted models to derive instances of the three problems presented in Chapter 4, and our approach to grammar-based whitebox fuzzing to generate random instances of problems related to software models. We then present results regarding the scalability of our symbolic encodings for each of the problems. Parts of this chapter are based on our publications [72–74, 123, 124].

Chapter 6, Conclusions. We draw the conclusions of our work, describe possible extensions and give an overview regarding future research directions towards solving model consistency problems of higher complexity by applying encodings to QBF.

Chapter 2

Preliminaries

This chapter introduces basic notions and terminology used throughout this work. In particular, we introduce some mathematical conventions, the syntax and semantics of propositional logic, and some basics on the Unified Modeling Language (UML), in particular the UML class diagram.

2.1 Mathematical Conventions and Relational Properties

We use the following mathematical notations.

We refer to the power set of a set X by $\mathcal{P}(X)$. For a tuple $Y = (y_1, \ldots, y_n)$ and $i \in [1..n]$ we refer to the projection to the i-th element by $\pi_i(Y) = y_i$.

We enclose a sequence of symbols with the brackets "[" and "]", e.g., we use $[a, b, c]$ to denote a sequence S. We refer to a symbol at position i in sequence S by $S[i]$, e.g, $S[2] = b$. A sequence can be empty and we denote this empty sequence by [].

We use the following properties of *binary relations*, where a binary relation is a collection of pairs over some alphabet $\mathcal{A}$. The relation R is *total* if for all $a, b \in \mathcal{A}$ it holds that $(a, b) \in R$ or $(b, a) \in R$ (or both). The relation R is *transitive* if for all $a, b, c \in \mathcal{A}$ it holds that if $(a, b) \in R$ and $(b, c) \in R$ then $(a, c) \in R$. The relation R is *reflexive* if for all $a \in \mathcal{A}$ it holds that $(a, a) \in R$. It is *irreflexive* if for all $a \in \mathcal{A}$ it holds that $(a, a) \notin R$. The relation R is *antisymmetric* if for all $a, b \in \mathcal{A}$ it holds that if $(a, b) \in R$ and $(b, a) \in R$ then $a = b$.

2.2 Propositional Logic

We first present an informal overview of propositional logic and then formally define its syntax and semantics as can be found in standard literature [28]. Propositional logic is concerned with *statements* represented by *Boolean variables* that can be assigned the *truth values true* and *false*.

Statements can be connected to form *sentences* by *connectives* like *and* or *or*. Depending on the truth values assigned to the variables and on the connectives, a sentence evaluates to *true* or *false*. For example, the variable x can represent the statement "it is raining today" and y can represent the statement "the streets are wet". If x is assigned *true* and y is assigned *false*, then the sentence "it is raining today and the streets are wet", represented by $x \wedge y$ where $\wedge$ means "and", evaluates to *false*.

The *syntax* of a logic describes how symbols (variables and connectives) can be connected to form a well-formed sentence. The *semantics* of a logic describes the meaning of well-formed sentences, that is, a mapping of sentences to a domain. In the case of propositional logic or quantified Boolean logic, this domain contains the two truth values *true* and *false*.

The representation of an application problem in a certain domain by sentences in some logic is called an *encoding*. Reasoning in some domain by using an encoding is called *symbolic* since facts inside the domain are represented by symbols.

We continue with a formal presentation of the syntax and the semantics of propositional logic.

2.2.1 Syntax

The language of formulas in *propositional logic* is defined over a set $\mathcal{V}$ of variables, the two logical symbols Verum ($\top$) and Falsum ($\bot$), and the logical operators $\neg$ and $\vee$. Any $v \in \mathcal{V} \cup \{\top, \bot\}$ is a propositional formula. If ϕ is a propositional formula then so is $\neg\phi$ and if ϕ_1 and ϕ_2 are propositional formulas, then so is $\phi_1 \vee \phi_2$. We use the auxiliary symbols "(" and ")" to group subformulas and we use the additional operators conjunction ($\wedge$), implication ($\rightarrow$), and equivalence ($\leftrightarrow$), which are defined for the propositional formulas ϕ_1 and ϕ_2 as follows: $(\phi_1 \wedge \phi_2) = \neg(\neg\phi_1 \vee \neg\phi_2)$, $(\phi_1 \rightarrow \phi_2) = (\neg\phi_1 \vee \phi_2)$, and $(\phi_1 \leftrightarrow \phi_2) = (\neg\phi_1 \vee \phi_2) \wedge (\neg\phi_2 \vee \phi_1)$. When the symbols "(" and:")" are absent, then the order of precedence over the operators is $\neg$ with the highest precedence followed by $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$ in this order.

We call a variable v and its negation $\neg v$ *literals*. We often write $\overline{v}$ for $\neg v$ and say that v is a *positive literal* and $\overline{v}$ is a *negative literal*. Further, a *clause* is a disjunction of literals. A propositional formula that contains only a conjunction of clauses is in *Conjunctive Normal Form* (CNF).

Alternatively, a propositional formula ϕ in CNF over the set of variables $\mathcal{V}$ and with a set of literals $L = \mathcal{V} \cup \{\overline{v} \mid \mathcal{V}\}$ can be defined by $\phi \subseteq \mathcal{P}(L)$, i. e., as a set of sets of literals. We often write a formula in CNF as sets of literals in round brackets. We write $\square$ for the empty clause.

We refer to the set of variables occurring in the propositional formula ϕ by $\mathsf{vars}(\phi)$ and to the variable of a literal ℓ by $\mathsf{var}(\ell)$.

Example 2.2.1. The string $\phi = ((x \vee z) \wedge (\overline{y} \wedge z)) \wedge (\overline{x} \rightarrow (\overline{y} \vee \overline{z}))$ is a propositional formula. The symbols x, $\overline{y}$, z, and $\overline{z}$ are the literals of this formula. The symbols x, y, and z are the variables of this formula. The conjunction of disjunctions $(x \vee z) \wedge (\overline{y}) \wedge (z) \wedge (x \vee \overline{y} \vee \overline{z})$ is a CNF representation of ϕ. $\Diamond$

$\top$
true

$\bot$
false

ϕ	$\neg\phi$
true	*false*
false	*true*

ϕ_1	ϕ_2	$\phi_1 \vee \phi_2$
true	*true*	*true*
true	*false*	*true*
false	*true*	*true*
false	*false*	*false*

Table 2.2.1: Truth tables describing the semantics of propositional logic.

2.2.2 Semantics

The *evaluation* of a propositional formula ϕ is due to an *assignment* $\sigma : \mathsf{vars}(\phi) \to \{\mathit{true}, \mathit{false}\}$ of a *truth value* to each variable. The assignment can be *partial* or *total*. The formula evaluates to either of the truth values according to the truth tables depicted in Table 2.2.1. A total assignment leads to a truth value of the whole formula. Alternatively, an assignment σ can be seen as a set of literals such that if $\mathsf{var}(\ell)$ is assigned to *true* then $\ell \in \sigma$, and if $\mathsf{var}(\ell)$ is assigned to *false* then $\overline{\ell} \in \sigma$ (note that since a variable can be assigned only one truth value, at most one of ℓ or $\overline{\ell}$ is in σ). Accordingly, σ is total if for each $v \in \mathsf{vars}(\phi)$ either $v \in \sigma$ or $\overline{v} \in \sigma$, and it is partial otherwise.

A propositional formula is called *satisfiable* if there exists an assignment which evaluates the formula to *true*, otherwise it is called *unsatisfiable*. A propositional formula that evaluates to *true* under any possible assignment is called *valid*. The problem of deciding whether a satisfying assignment exists for some propositional formula is called the *satisfiability problem of propositional logic (SAT)*.

Example 2.2.1 (Cont. from p. 8). The formula ϕ is satisfiable witnessed by the complete assignment $x \mapsto \mathit{true}$, $y \mapsto \mathit{false}$, $z \mapsto \mathit{true}$, which can also be described by the set $\{x, \overline{y}, z\}$ of literals. A satisfying partial assignment is $y \mapsto \mathit{false}$, $z \mapsto \mathit{true}$, or the set $\{\overline{y}, z\}$ of literals. $\Diamond$

Two propositional formulas ϕ_1 and ϕ_2 are *logically equivalent*, denoted $\phi_1 \equiv \phi_2$, if they are satisfied by the same set of assignments. Two formulas are *satisfiability equivalent* if ϕ_1 is satisfiable if and only if ϕ_2 is satisfiable. For a propositional formula ϕ a logically equivalent formula in CNF can be created by applying a set of equivalence-preserving transformation rules. In particular, these are double negation elimination, i.e., $\neg\neg\phi \equiv \phi$, distribution, i.e., $(x) \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$, and deMorgan's law, i.e., $\neg(x \wedge y) \equiv \neg x \vee \neg y$ and $\neg(x \vee y) \equiv \neg x \wedge \neg y$. Applying these rules can have an exponential overhead with respect to the size of the formula due to the application of the distribution rule.

However, when only satisfiability equivalence is required, then ϕ can be transformed into a CNF formula ϕ^{cnf} with linear overhead with respect to the formula size by introducing auxiliary variables [114]. Due to the auxiliary variables, the formula ϕ^{cnf} is not logically equivalent to ϕ, but the satisfying assignments to the formulas are closely related. In particular, any satisfying assignment to ϕ can be extended to a satisfying assignment to ϕ^{cnf} by adding assignments to the auxiliary variables, and for any satisfying assignment to ϕ^{cnf} the truth values for the variables contained in both ϕ^{cnf} and ϕ are also a satisfying assignment to ϕ. Therefore, this translation provides a practical means to prepare non-CNF formulas to be solved by a SAT solver that requires a formula in CNF as input, which is the case for most of the state-of-the-art SAT solvers.

2.3 The Unified Modeling Language (UML)

The result of a unification of a myriad of modeling languages, the Unified Modeling Language (UML) is today's most popular general-purpose visual modeling language [106]. It is applied in various domains, in particular such related to software development, and in various stages of development reaching from design sketches to specification and documentation purposes. It features a set of different diagrams that allow to view a system from different angles, with different levels of abstraction, and with different focus points. This high degree of generalization and flexibility, however, comes along with an inherent complexity and different, sometimes contradicting, requirements to the design of the language. For these reasons, among others, the UML lacks a formal specification, in particular with regards to its semantics [106].

The UML consists of a set of diagrams, each with a different purpose. Some, such as the state machine diagram, focus on dynamic aspects, others, such as the class diagram, on static aspects, and others are designed for end-user documentation, such as the use-case diagram. The terminology of the UML combines diagrams that are closely related into *views* [106]. However, many views contain only one diagram, such as the state machine view (contains only the state machine diagram) or the static view (contains only the class diagram). In this work, we therefore use the terms "view" and "diagram" synonymously when the context is clear.

2.3.1 Syntax

Other than for textual languages, where the appearance of the symbols is usually integrated in the definition of their syntax, for visual languages the *abstract syntax* is distinguished from the *concrete syntax*. The former describes the components and their relations to one another, and the latter describes their concrete visual design.

Most of the abstract syntax of the UML is defined by a diagram of the UML itself — the *class diagram*. A class diagram defining the syntax of other diagrams is commonly referred to as *metamodel* [62]. Hence, the UML metamodel describes the major part of the syntax of the diagrams of the UML in class diagram notation. The syntax of the metamodel itself, i.e., of the UML class diagram, is defined by another class diagram, the metametamodel. The approach works in a bootstrapping fashion [113], where on the most basic layer, anything is described by classes and their relationships. A class, in this most basic form, has a name and can be instantiated. The second layer instantiates "class" by elements that represent the syntax of the class diagram, which can then be used to describe other diagrams.

Figure 2.3.1 shows some components of the class diagram in their concrete syntax. The basic components of a class diagram are *classes* that can relate to one another by *associations* or by *generalizations*. Classes are usually depicted as rectangles and associations as lines. A class has a name and a (possibly empty) set of attributes, and an association may have a name and attributes and may indicate cardinalities of the objects instantiating the connected classes. A *composition* is a special kind of association relating a class to its container class. If an instance of a container is deleted, then the instances of the classes it contains are also deleted. This is not the case for instances of classes that are connected via an association that is not a composition. A *generalization* relates a class to its superclass(es), meaning that the former inherits all associations and attributes from the latter. Cardinalities are given in positive integers and the

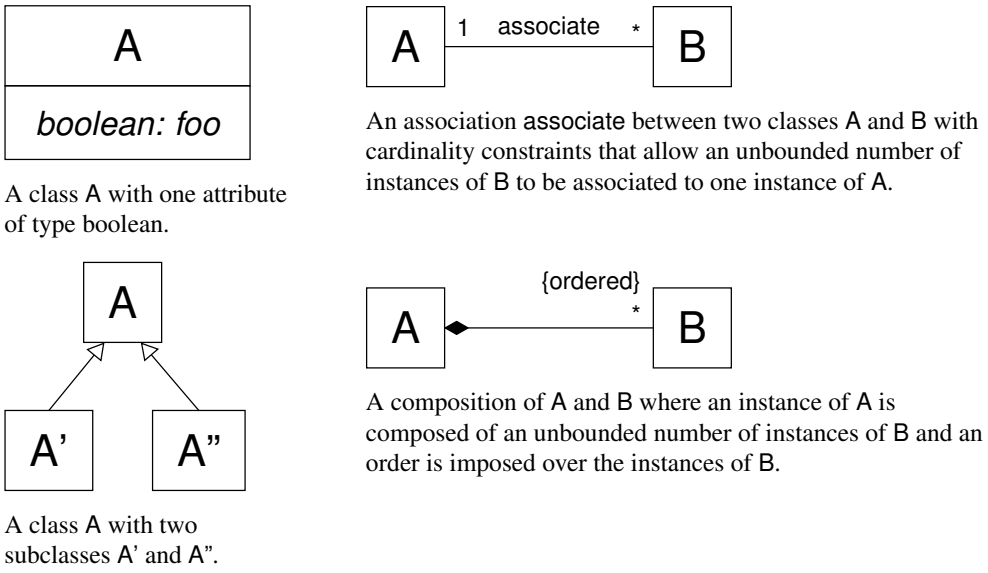

Figure 2.3.1: Components of the class diagram in their concrete syntax.

asterisk symbol ("*") for an unbounded number of associated instances. They are indicated at both ends of regular associations and at one end of compositions. Compositions per definition have cardinality 1 at the container side. Attributes of associations describe properties of the instances of the connected class. The only attribute used in this work is the {ordered} attribute, which enforces an order over the associated instances. The UML class diagram consists of many more components than described above, but we confine this overview to the components used in this work. A detailed description of the full UML can be found in the UML specifications, the UML Infrastructure [61] and the UML Superstructure [62].

Most of the syntax of all remaining diagrams of the UML is defined by the UML metamodel. Additional syntactic elements of the UML, which exceed the expressive power of the class diagram, are defined in a formal language, the Object Constraint Language (OCL) [63]. In this work, however, we rely on syntactic elements of UML diagrams that can be described by a class diagram. A precise definition of the syntax used by the UML can be found in the UML Infrastructure [61].

2.3.2 Semantics

The semantics of the UML is described only imprecisely and mostly in natural language inside the UML Superstructure [62], which is part of the official UML Standard. This lack of formality can be disregarded in many applications of visual modeling such as high-level design or general documentation of a system. However, other applications, such as code generation, as performed in model-driven engineering, or precise specifications of a system strictly require a formal semantics. For such applications, a formal definition of the semantics of the visual language, possibly a subset of the UML, is indispensable. The lack of precision and the confusion about

its semantics has been recognized by Harel and Rumpe [65, 66], who emphasize the importance of defining semantics of visual languages in the same way as for textual languages, that is, by mapping its syntactic elements to a semantic domain.

The need for a formal semantics is also reflected by various efforts to formalize the UML or subsets in the literature. First works appeared for the previous version of the UML (UML 1.3) a few years after the birth of the UML. Real-time action logic has been applied to define a structured axiomatic semantics [83] for a subset of UML 1.3. Object-Z, which is an object-oriented specification language based on Zermelo-Fränkel set theory and first-order logic, has been used to formalize the UML class diagram, the state machine diagram, and the sequence diagram [75, 95]. The initiative "precise UML" was particularly devoted towards a general formal semantics for the UML [47] and involved in the definition of the current version of the UML, UML 2.4.1.

However, as of version 2.4.1, the UML does not have a standardized formal semantics. Instead, most formalizations are provided whenever needed for a particular domain and therefore often focus on a particular subset of the UML. Mostly, it is automated code generation or formal verification of software models that motivate the introduction of a formal semantics. With the new development paradigm of MDE, the OMG itself started working on an execution semantics for some UML models, in particular the class diagram and the state machine diagram, resulting in the OMG standard fUML [59]. Another motivation to formalize the semantics of the UML is to prove particular theoretical properties of UML models. In this context, the computational complexity of general reasoning over UML class diagrams has been shown to be EXPTIME-complete by a reduction from a problem in a particular description logic and an encoding of the UML class diagram into a particular description logic [11].

Chapter 3

The Tiny Multiview Modeling Language (*tMVML*)

This chapter introduces the modeling language concepts essential for the verification problems discussed in Chapter 4. These concepts are strongly inspired by the Unified Modeling Language (UML) as specified by the OMG [60]. The UML contains a set of diagrams that describe the system under development from different points of view. However, these diagrams lack a complete formal semantics and therefore require us to establish formal definitions that go beyond what is offered by the UML Standard. We therefore establish the *tiny Multiview Modeling Language* (*tMVML*), a modeling language that is a syntactic subset of the UML and provide a formal semantics. We focus on two diagrams, the state machine diagram and the sequence diagram, as the three verification problems discussed in Chapter 4 refer to these two views.

Our approach to dealing with the syntax and semantics of the *tMVML* is as follows. We first define its syntax by the *tMVML* metamodel using the UML metamodeling approach described in Section 2.3, i.e., the *tMVML* metamodel is a UML class diagram specifying the syntax of the *tMVML* diagrams. It covers two diagrams, the state machine diagram and the sequence diagram, and their interplay. Hence, these two diagrams are instances of the *tMVML* metamodel.

With the same motivation as in other works on the formalization of the UML, e.g., [50, 78], we then represent the *tMVML* metamodel in a mathematical notation by concisely defining each component and its relations to other components. These definitions facilitate explanations on behavioral aspects of the *tMVML* diagrams. The behavioral aspects are not described by the metamodel because the metamodel is only suitable to define the syntax. These aspects can rather be considered as a part of the semantics as their definition is necessary to describe semantic properties of the diagrams. We later define the semantic properties using propositional logic within the encodings given for the problems in Sections 4.2 to 4.4. Through the encodings to propositional logic, the syntactic representations of *tMVML* models are given a formal semantics. The properties can be considered the semantic domain of the *tMVML* and the respective

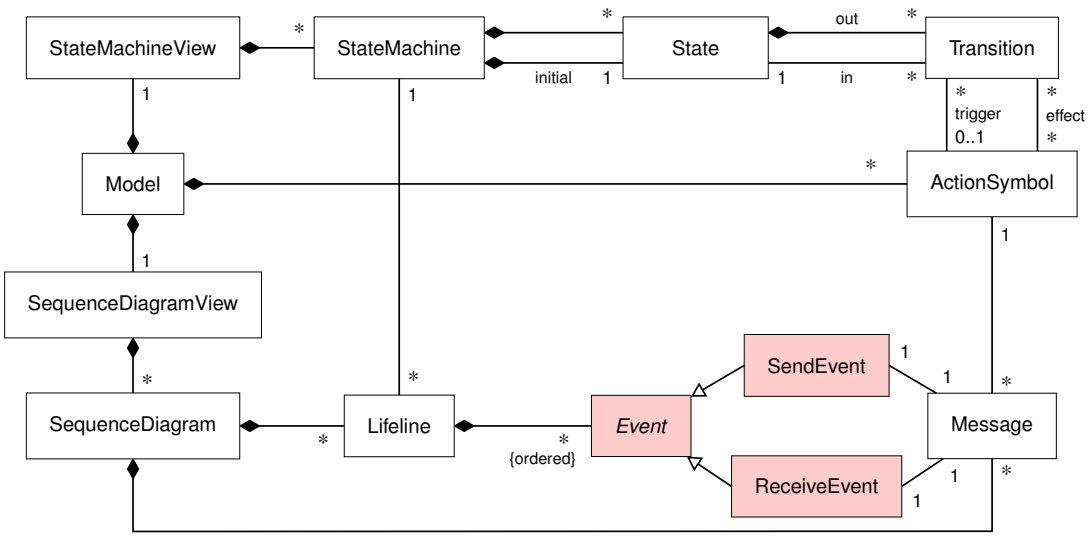

Figure 3.1.1: Event-based version of the *tMVML* metamodel.

encodings of the syntactic elements can be considered the semantic mapping.

This chapter consolidates the definitions given in our previous publications [72, 74, 124]. The implementation of the *tMVML* metamodel is available on our project website.[1]

3.1 The *tMVML* Metamodel

The *tMVML* metamodel describes the static structure and the abstract syntax of our modeling language. We present two versions of the *tMVML* metamodel, depicted in Figures 3.1.1 and 3.1.3. They differ only in the presence of the class *Event* and its children. For better readability we often typeset instances of metaclasses in standard lowercase font using the same name as the metaclass, e.g., in order to refer to an instance of the metaclass State we write "state".

The *tMVML* metamodel has a root class Model which contains two classes representing views, SequenceDiagramView and StateMachineView, and the class ActionSymbol. Action symbols realize the communication between different state machines and describe communication in sequence diagrams. When the context is clear, we often call them "symbols" instead of "action symbols".

The StateMachineView contains a set of *state machines*, each containing a set of *states*, one of which is the initial state. States are connected by *transitions*. To each transition an action symbol can be assigned as *trigger* and one or more action symbols can be assigned as *effects*.

In the concrete syntax, we mostly follow the conventions of the UML state machine. Rectangles with rounded corners denote states which are connected by transitions. Each transition carries a label that consists of two parts separated by the symbol "/"; a trigger on the left side and a set of effects on the right side. We use the special symbol ϵ on the left side of the symbol

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

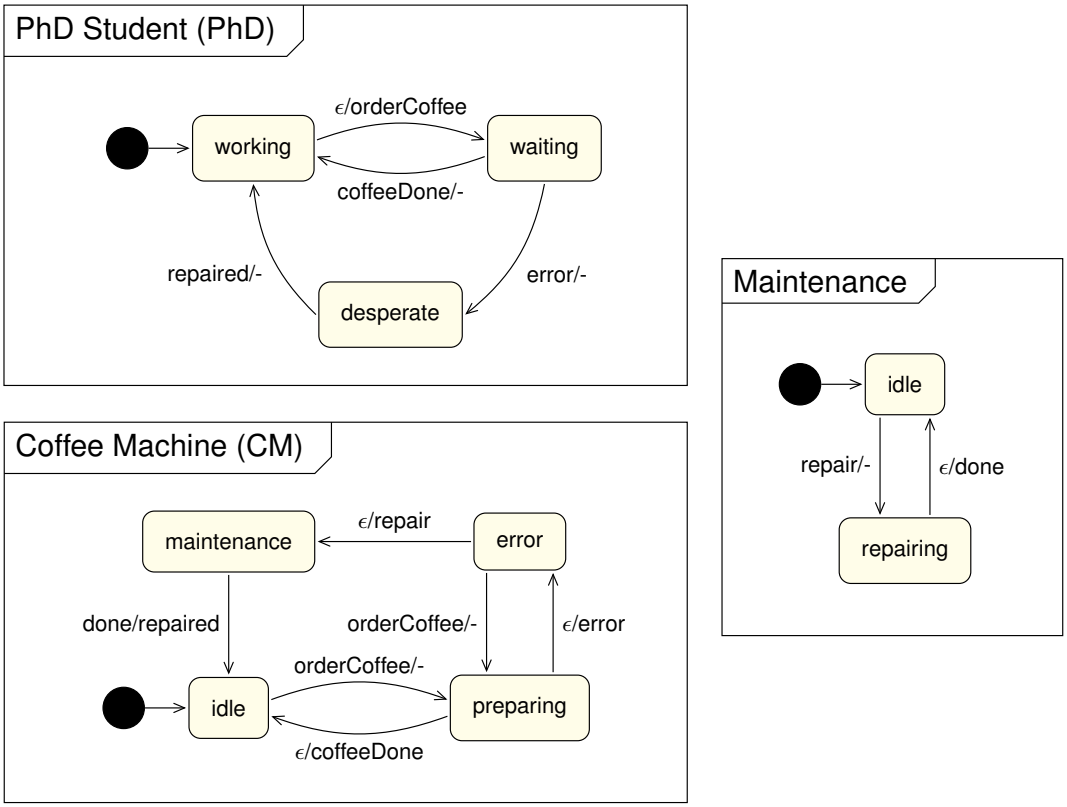

Figure 3.1.2: Three state machines modeling a PhD student, a coffee machine, and a maintenance unit.

"/" to indicate that a transition has no trigger and we use the symbol "-" to indicate the empty set of effects. As in the UML, we indicate the initial state of a state machine by an incoming edge originating from a black dot.

Example 3.1.1. Figure 3.1.2 shows three state machines instantiating the *tMVML* metamodel in concrete syntax. They describe the behaviors of a PhD student, a coffee machine, and a maintenance unit for the coffee machine. The state machine "PhD Student" contains three states, "working", "waiting", and "desperate", where "working" is the initial state, and four transitions connecting the states using action symbols as triggers and effects. ◊

The SequenceDiagramView contains a set of sequence diagrams. A sequence diagram consists of a set of *lifelines* and a set of *messages*. Each lifeline instantiates a state machine, i.e., it implements the behavior described by the state machine. Each message carries an action symbol and is attached via a send event and a receive event to one or two lifelines.

The order over events is useful to model both asynchronous and synchronous message passing with timed events, i.e., where the times of dispatch and receipt of a message are not the same. The assumption that the transfer of a message does not consume time, i.e., the time of dispatch and receipt are the same, implies a total order of the messages. Events can therefore be

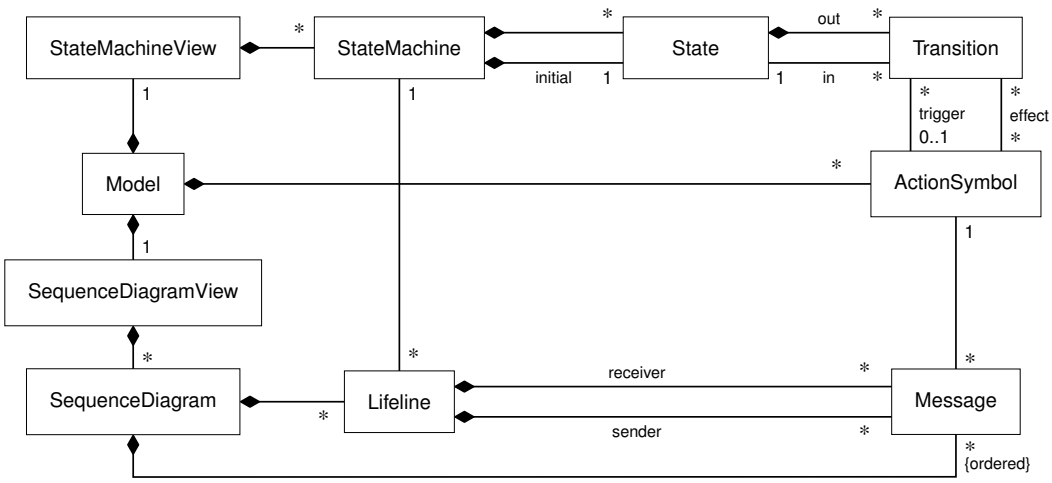

Figure 3.1.3: Alternative metamodel of the *tMVML*, disregarding the *Event* class and its children.

disregarded. A modified metamodel that excludes events and imposes an order over messages directly is depicted in Figure 3.1.3. Depending on the application, one or the other metamodel can be more appropriate.

The problems discussed in Chapter 4 are all defined under the assumption that message passing does not consume time. We nevertheless provide a formalization based on ordered events and a simplification thereof based on ordered messages. The former helps us to stay close to the definition of sequence diagrams in the UML standard and to extend our approach to timed events in future work. The latter is more straightforward and intuitive when it comes to defining the latter two of the problems in Sections 4.3 and 4.4.

Like we did for state machines, we follow the concrete syntax of the UML to depict sequence diagrams. Lifelines are shown as rectangles with a dashed vertical line underneath. Each lifeline's name is shown inside the rectangle before the symbol ":", followed by the name of the state machine it instantiates after the symbol ":". Along the lifelines, a sequence of messages can be aligned. Each message is depicted as an arrow from the sender lifeline to the receiver lifeline labeled with the symbol being sent. The set of symbols sent by messages of a sequence diagram is the same as the set of symbols used in the state machines its lifelines instantiate. Events are depicted implicitly by the arrowtips and arrowends of messages.

Example 3.1.2. Figure 3.1.4 shows a sequence diagram that describes a communication scenario between lifelines instantiating the state machines in Figure 3.1.2. The lifelines alice and bob instantiate the state machine PhD Student, the lifeline cm instantiates the state machine Coffee Machine, and the lifeline m instantiates the state machine Maintenance. The lifelines send and receive messages between one another. The messages are ordered along the lifelines from top to bottom, i.e., the first message is wantCoffee sent by alice and received by cm and the last message is coffeeDone sent by cm and received by alice. ◊

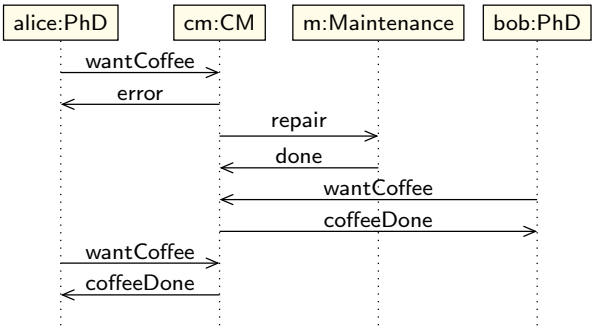

Figure 3.1.4: A sequence diagram showing an interaction between two PhD students, a coffee machine, and a maintenance unit.

3.2 Formal Description of the *tMVML*

This section provides a formal description of the static structure of the *tMVML* by translating the elements of the metamodel into a mathematical notation.

Some elements can be translated in a straightforward manner. For example, classes that only act as containers for other classes, i.e., whose only associations are compositions, can be regarded as sets of instances of the classes they are composed of. This way, Model, the root class of the metamodel can be defined as a triple containing three sets of instances of ActionSymbol, StateMachineView, and SequenceDiagramView. In the same way, the latter two classes can be defined as sets of instances of StateMachine, respectively SequenceDiagram. Both classes StateMachineView and SequenceDiagramView serve only as containers for a set of objects of the classes StateMachine and SequenceDiagram respectively. They could be omitted, but we included them as an extra layer to make the notion of a "view" explicit. The formal descriptions become more complicated when other associations, generalizations, or attributes on associations or classes are involved.

The universe of discourse for the definitions of the *tMVML* metamodel is the quintuple $\mathcal{A} = (\mathcal{A}^S, \mathcal{A}^A, \mathcal{A}^L, \mathcal{A}^E, \mathcal{A}^M)$ where $\mathcal{A}^S$ denotes a set of states, $\mathcal{A}^A$ denotes a set of action symbols, $\mathcal{A}^L$ denotes a set of lifelines, $\mathcal{A}^E$ denotes a set of events, and $\mathcal{A}^M$ denotes a set of messages.

The class StateMachine is defined as follows.

Definition 3.2.1 (State Machine)**.** Given the universe $\mathcal{A}$ with $\pi_1(\mathcal{A}) = \mathcal{A}^S$ and $\pi_2(\mathcal{A}) = \mathcal{A}^A$, a *state machine* M is a quintuple $(S, \iota, A^{tr}, A^{\mathit{eff}}, T)$, where

- $S \subseteq \mathcal{A}^S$ is a set of *states*,
- $\iota \in S$ is a designated *initial state*,
- $A^{tr}, A^{\mathit{eff}} \subseteq \mathcal{A}^A$ are sets of symbols used as *triggers* and *effects*, and
- $T \subseteq S \times A^{tr} \cup \{\epsilon\} \times \mathcal{P}(A^{\mathit{eff}}) \times S$ is a set of *transitions* such that there is no transition $(s, \epsilon, \emptyset, s') \in T$ for any $s, s' \in S$.

△

A state machine consists of a set of states including an initial state, two alphabets with symbols for triggers and effects, respectively, and a transition relation between the states. For a transition $t \in T$ with $t = (s, trg, \mathit{eff}, s')$, s is the source state of the transition, s' is the target state, trg is a symbol (trigger) which can be the special symbol $\epsilon \notin A^{tr}$, and eff is a set of symbols (effects).

On the behavioral level, which we discuss in Section 3.3, the receipt of its trigger trg initiates the execution of a transition and the receipt of each effect in eff by a different state machine completes the execution of the transition. A transition containing ϵ as trigger can be initiated no matter whether any symbol has been received. A transition containing the empty set as effects is completed without triggering other transitions. We assume that no transition of a state machine contains both ϵ as trigger and the empty set as effects. Such transitions can be eliminated by contracting the connected states.

Example 3.1.1 (Cont. from p. 15). In Figure 3.1.2, the state machine PhD Student contains the states $S = \{\mathsf{working}, \mathsf{desperate}, \mathsf{waiting}\}$, the triggers $A^{tr} = \{\mathsf{coffeeDone}, \mathsf{error}, \mathsf{repaired}\}$, the effect $A^{\mathit{eff}} = \{\mathsf{orderCoffee}\}$, and the transitions $T = \{(\mathsf{working}, \epsilon, \{\mathsf{orderCoffee}\}, \mathsf{waiting}),$
$(\mathsf{waiting}, \mathsf{coffeeDone}, \emptyset, \mathsf{working}),$
$(\mathsf{waiting}, \mathsf{error}, \emptyset, \mathsf{desperate}),$
$(\mathsf{desperate}, \mathsf{repaired}, \emptyset, \mathsf{working})\}$. ♢

The definition of the class SequenceDiagram is more complex and builds on definitions of the classes Lifeline and Message. They can be described with or without the notion of *events*, which facilitates the description of messages that are sent and received at different times. As described in the metamodel of Figure 3.1.1, a sequence diagram contains a set of lifelines, a set of events, and a set of messages. Additionally, a lifeline is connected to a set of messages via a send event and a receive event. The {ordered} constraint on the relation between the classes Lifeline and Event puts the events into a sequence relative to the lifeline.

This formulation forms a basis to model communication between lifelines and timed events. Under the assumption that no time passes between the send and the receive event of a message, however, the {ordered} constraint only needs to be imposed over messages, not over events. This way the classes Event and its two inheritors SendEvent and ReceiveEvent can be omitted, as depicted in Figure 3.1.3.

We first provide a definition of a sequence diagram based on an {ordered} constraint on events as described by the metamodel in Figure 3.1.1 and refer to it as *event-based sequence diagram*. For this type of sequence diagram, we further define two properties *well-formedness* and *time consistency*. An event-based sequence diagram fulfilling these two properties results in a sequence diagram where the transmission of messages does not consume time. Second we define a sequence diagram based on an {ordered} constraint on messages which completely disregards events as described by the metamodel in Figure 3.1.3. This alternative definition is shorter and possibly more intuitive when it comes to describing the verification problems of Section 4.3 and 4.4. Both definitions of a sequence diagram are based on definitions of its components, the classes Lifeline and Message.

Definition 3.2.2 (Event-based Lifeline)**.** Given a set $\mathcal{M}$ of state machines and the universe $\mathcal{A}$ with $\pi_3(\mathcal{A}) = \mathcal{A}^L$ and $\pi_4(\mathcal{A}) = \mathcal{A}^E$ an event-based lifeline is a tuple $(l, M, E^{sd}, E^{rv}, >)$ where

- $l \in \mathcal{A}^L$ is the name of the lifeline,
- $M \in \mathcal{M}$ is the state machine the lifeline instantiates,
- $E^{sd} \subseteq \mathcal{A}^E$ is a set of send events associated to the lifeline,
- $E^{rv} \subseteq \mathcal{A}^E$ is a set of receive events associated to the lifeline,
- E^{sd} and E^{rv} are disjoint, and
- $> \subset (E^{sd} \cup E^{rv}) \times (E^{sd} \cup E^{rv})$ is a transitive, antisymmetric, and irreflexive relation.

△

A lifeline is an instance of a state machine M. The name l allows to distinguish different instances of the same state machine. The sets E^{sd} and E^{rv} contain send and receive events on the lifeline, and the relation $>$ describes the {ordered} constraint of its events as modeled in the *tMVML* metamodel.

Definition 3.2.3 (Event-based Message)**.** Given the universe $\mathcal{A}$ with $\pi_2(\mathcal{A}) = \mathcal{A}^A$ and $\pi_4(\mathcal{A}) = \mathcal{A}^E$ a *message* is a triple (e_s, a, e_r) where

- $e_s \in \mathcal{A}^E$ is the send event,
- $a \in \mathcal{A}^A$ is an action symbol, and
- $e_r \in \mathcal{A}^E \setminus \{e_s\}$ is the receive event. △

The definition of a message is straightforward as it is associated to exactly one action symbol, one send event, and one receive event.

Definition 3.2.4 (Event-based Sequence Diagram)**.** Given the universe $\mathcal{A}$ with $\pi_2(\mathcal{A}) = \mathcal{A}^A$, $\pi_3(\mathcal{A}) = \mathcal{A}^L$, and $\pi_4(\mathcal{A}) = \mathcal{A}^E$, and a set $\mathcal{M}$ of state machines, a sequence diagram is a pair $(\mathcal{L}, \mathcal{N})$, where $\mathcal{L}$ is a set of lifelines over $\mathcal{M}$ and $\mathcal{A}^L$, and $\mathcal{N}$ is a set of messages over $\mathcal{A}^A$ and $\mathcal{A}^E$, such that

- for any two lifelines L and L' in $\mathcal{L}$ their names $\pi_1(L)$ and $\pi_1(L')$ are distinct,
- for any two lifelines L and L' the sets $\pi_3(L)$, $\pi_4(L)$, $\pi_3(L')$, and $\pi_4(L')$, i.e., their sets of send and receive events, are pairwise disjoint,
- for any two messages N and N', $\pi_1(N)$, $\pi_3(N)$, $\pi_1(N')$, and $\pi_3(N')$, i.e., their send and receive events, are pairwise distinct, and
- for $E_L = \bigcup_{L \in \mathcal{L}} \pi_3(L) \cup \bigcup_{L \in \mathcal{L}} \pi_4(L)$ and $E_N = \bigcup_{N \in \mathcal{N}} \pi_1(N) \cup \bigcup_{N \in \mathcal{N}} \pi_3(N)$ there exists a bijective function that maps E_L to E_N and vice versa, i.e., each event on a lifeline is bijectively associated to an event of a message.

△

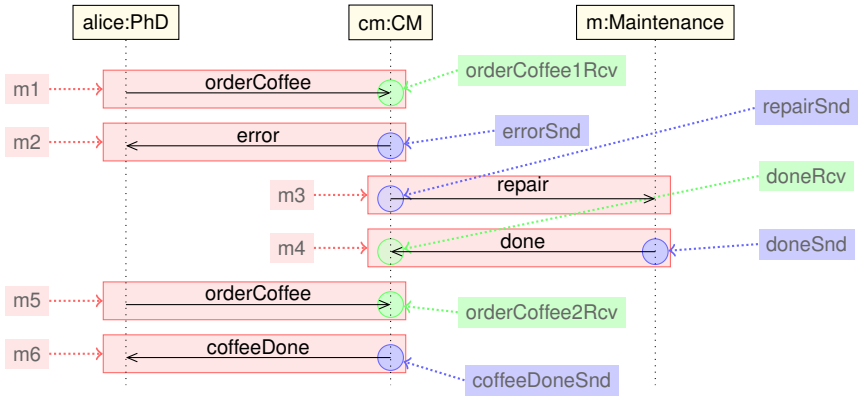

Figure 3.2.1: Sequence diagram indicating messages and events.

The definition of a sequence diagram contains a set of lifelines and a set of messages as described by the metamodel. It further establishes the relation between messages and lifelines via events. The set of events contained in an event-based sequence diagram can therefore be retrieved via the set of messages.

Example 3.1.2 (Cont. from p. 16). Figure 3.2.1 shows a sequence diagram in concrete syntax, which additionally indicates some events. (Usually, events are not explicitly depicted in the concrete syntax, cf. Figure 3.1.4.) In this sequence diagram, $\mathcal{L}$ contains the lifelines alice, cm, and m. The lifelines are instances of the state machines PhD Student, Coffee Machine, respectively Maintenance of Figure 3.1.2.

The diagram contains six messages, m1 to m6, each of which is depicted as an arrow between two lifelines. Each arrowhead represents a receive event and each arrowtail a send event. For example, for message $\mathsf{m4} = (\mathsf{doneSnd}, \mathsf{done}, \mathsf{doneRcv})$, doneSnd is its send event, done its symbol, and doneRcv its receive event. All messages' events are connected to a lifeline. For example, lifeline cm handles events orderCoffee1Rcv, errorSnd, repairSnd, doneRcv, orderCoffee2Rcv, and coffeeDoneSnd. ◊

We use the following functions to refer to elements of an event-based sequence diagram given the universe $\mathcal{A}$, a set $\mathcal{M}$ of state machines, and an event-based sequence diagram $D = (\mathcal{L}, \mathcal{N})$.

- $\mathsf{act} : \mathcal{N} \to \mathcal{A}^A$, $\mathsf{snd} : \mathcal{N} \to \mathcal{A}^E$, and $\mathsf{rcv} : \mathcal{N} \to \mathcal{A}^E$, such that for all $N \in \mathcal{N}$ it holds that $\mathsf{act}(m) = \pi_2(N)$, $\mathsf{snd}(m) = \pi_1(N)$, and $\mathsf{rcv}(m) = \pi_3(N)$, i.e., these functions refer to the action symbol, send event, and receive event of a message.

- $\mathsf{symb} : \mathcal{A}^E \to \mathcal{A}^A$ is a partial function such that $\mathsf{symb}(e) = a$ if and only if there exists a message $N \in \mathcal{N}$ with either $\mathsf{snd}(N) = e$ or $\mathsf{rcv}(N) = e$, and $\mathsf{act}(N) = a$, i.e., the action symbol of the message an event is associated to. Note that each function value is unique due to the pairwise disjointness of sets of events on lifelines and distinctness of events on messages as described in Definition 3.2.4.

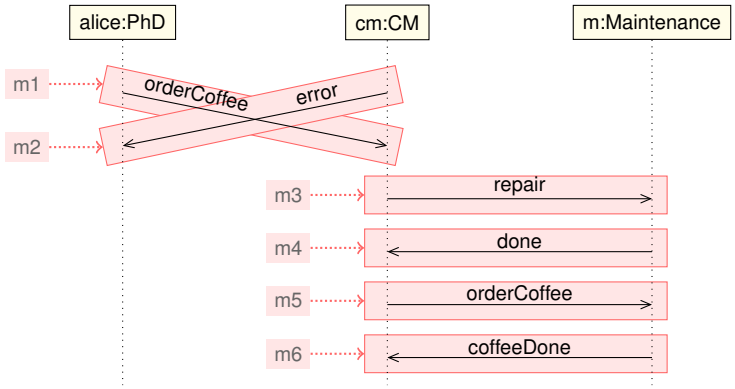

Figure 3.2.2: Time-inconsistent sequence diagram.

- life : $\mathcal{A}^E \to \mathcal{L}$ is a partial function such that $\mathsf{life}(e) = l$ if and only if $e \in \pi_3(L) \cup \pi_4(L)$, where $\pi_3(L)$ is the set of send events on lifeline L and $\pi_4(L)$ is the set of receive events on lifeline L. Note that each function value is unique due to the pairwise disjointness of sets of events on lifelines as described in Definition 3.2.4.

The following properties restrict the definition of a sequence diagram concerning its event orderings. First, *well-formedness* of a sequence diagram enforces an order on the events with respect to a lifeline.

Definition 3.2.5 (Well-Formedness)**.** A sequence diagram $(\mathcal{L}, \mathcal{N})$ is *well-formed* if and only if for each $L \in \mathcal{L}$ the relation $\pi_5(L)$ is total. △

A well-formed sequence diagram poses a total order over events per lifeline, but not over messages. This way, it allows a message N to be received after a message N' even if N has been sent before N' on the same lifeline. Figure 3.2.2 shows such a case with messages m1 and m2. When such "overtaking" of messages is not present, then a sequence diagram is *time-consistent*. The definition of time consistency is based on a *message ordering* relation over a sequence diagram, which describes an order of the sequence diagram's messages according to the order of its events.

Definition 3.2.6 (Message Ordering)**.** Given a well-formed sequence diagram $(\mathcal{L}, \mathcal{N})$, the *message ordering* relation $\succ \subseteq \mathcal{N} \times \mathcal{N}$ contains a pair (N, N') if and only if for $N = (e_s, a, e_r)$, $N' = (e'_s, a', e'_r)$, $L = \mathsf{life}(e_s)$, $>_L = \pi_5(L)$, $L' = \mathsf{life}(e_r)$, and $>_{L'} = \pi_5(L')$ it holds that

- $\mathsf{life}(e'_s) = L$ and $e_s >_L e'_s$,
- $\mathsf{life}(e'_r) = L$ and $e_s >_L e'_r$,
- $\mathsf{life}(e'_s) = L'$ and $e_r >_{L'} e'_s$, or
- $\mathsf{life}(e'_r) = L'$ and $e_r >_{L'} e'_r$.

△

Example 3.1.2 (Cont. from p. 16). In the sequence diagram of Figure 3.2.1, the message order is given by $\mathsf{m1} \succ \mathsf{m2} \succ \mathsf{m3} \succ \mathsf{m4} \succ \mathsf{m5}$. ◊

Definition 3.2.7 (Time Consistency)**.** A well-formed sequence diagram is called time-consistent if and only if its message ordering relation $\succ$ is antisymmetric. △

Example 3.1.2 (Cont. from p. 16). The message order for the sequence diagram in Figure 3.2.1 is antisymmetric. Therefore, the sequence diagram is time-consistent. In the sequence diagram of Figure 3.2.2, the message order contains the two pairs $\mathsf{m1} \succ \mathsf{m2}$ and $\mathsf{m2} \succ \mathsf{m1}$, therefore the message order is not antisymmetric and the sequence diagram is time-inconsistent. ◊

Well-formedness and time consistency of a sequence diagram result in a total order over the messages. For such diagrams we give a shorter and possibly more intuitive description below. The description disregards the notion of events and directly imposes an order on messages with respect to the sequence diagram. The corresponding metamodel is depicted in Figure 3.1.3. We first have to re-define the classes Lifeline and Message since their event-based definitions (cf. Definitions 3.2.2 and 3.2.3) refer to events.

Definition 3.2.8 (Lifeline)**.** Given a set $\mathcal{M}$ of state machines and the universe $\mathcal{A}$ with $\pi_3(\mathcal{A}) = \mathcal{A}^L$, a lifeline is a pair (l, M) where $l \in \mathcal{A}^L$ is the name of the lifeline and $M \in \mathcal{M}$ is the state machine instantiated by the lifeline. △

The new definition of a lifeline differs from the event-based definition in Definition 3.2.2 only in that the sets of send events and receive events and their relation is removed.

Definition 3.2.9 (Message)**.** Given the universe $\mathcal{A}$ with $\pi_1(\mathcal{A}) = \mathcal{A}^S$ and $\pi_2(\mathcal{A}) = \mathcal{A}^A$, and a set $\mathcal{L}$ of lifelines such that each lifeline's state machine is defined over $\mathcal{A}^S$ and $\mathcal{A}^A$, a *message* is a triple (σ, a, ρ) where

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

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

The new definition of a message differs from the event-based definition in Definition 3.2.3 in that it refers directly to lifelines instead of events. For a message (σ, a, ρ), the sender lifeline σ either refers to a 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 figures depicting sequence diagrams in concrete syntax. The receiver lifeline ρ refers to a state machine.

Based on these definitions of a lifeline and of a message, a sequence diagram can be defined as follows.

Definition 3.2.10 (Sequence Diagram)**.** Given the universe $\mathcal{A}$ with $\pi_2(\mathcal{A}) = \mathcal{A}^A$ and $\pi_3(\mathcal{A}) = \mathcal{A}^L$, and a set $\mathcal{M}$ of state machines over $\mathcal{A}^A$, a sequence diagram is a pair $(\mathcal{L}, \mu)$ where $\mathcal{L}$ is a set of lifelines over $\mathcal{M}$ and $\mathcal{A}^L$, and $\mu = [N_1, \ldots, N_n]$ is a sequence of messages such that

- for any two lifelines L and L' in $\mathcal{L}$ their names $\pi_1(L)$ and $\pi_1(L')$ are distinct, and
- for $i \in [1..n]$ and $N_i = (\sigma_i, a_i, \rho_i)$ it holds that $\sigma_i, \rho_i \in \mathcal{L}$ and $a_i \in \mathcal{A}^A$.

△

Example 3.2.1. The sequence diagram in Figure 3.2.1 contains the set

$$\mathcal{L} = \{(\mathsf{alice}, \mathsf{PhD}), (\mathsf{cm}, \mathsf{CM}), (\mathsf{m}, \mathsf{maintenance})\}$$

of lifelines and the sequence

$$\mu = [((\mathsf{alice}, \mathsf{PhD}), \mathsf{orderCoffee}, (\mathsf{cm}, \mathsf{CM})), \ldots, ((\mathsf{cm}, \mathsf{CM}), \mathsf{coffeeDone}, (\mathsf{alice}, \mathsf{PhD}))]$$

of messages. ◊

Finally, we can define the sequence diagram view and the state machine view as sets of sequence diagrams, respectively state machines, and a model as a triple containing a sequence diagram view, a state machine view, and an alphabet shared by the two views.

Definition 3.2.11 (State Machine View)**.** Given the universe $\mathcal{A}$, a state machine view is a set of state machines (Definition 3.2.1). △

Definition 3.2.12 (Sequence Diagram View)**.** Given the universe $\mathcal{A}$ and a set $\mathcal{M}$ of state machines over $\mathcal{A}$, a sequence diagram view is either a set of event-based sequence diagrams (Definition 3.2.4) or a set of sequence diagrams (Definition 3.2.10).

△

Definition 3.2.13 (Model)**.** A model is a triple $(\mathcal{A}, \mathcal{M}, \mathcal{D})$ where $\mathcal{M}$ is a state machine view over $\mathcal{A}$ and $\mathcal{D}$ is a sequence diagram view over $\mathcal{A}$ and $\mathcal{M}$. △

3.3 Behavioural Aspects of the *tMVML*

In this section we describe the interplay between instances of state machines and the connection between sequence diagrams and state machines. In particular, we describe two levels of *consistency* a sequence diagram can have with respect to the set of state machines its lifelines instantiate. The first level of consistency, *trigger consistency*, considers only receive events of a sequence diagram, which occur as triggers in state machines. Trigger consistency verifies that a sequence of receive events in a lifeline of a sequence diagram occurs as a sequence of triggers directly connecting states of the associated state machine. To check whether a sequence diagram is trigger-consistent, it suffices to check that each lifeline is trigger-consistent.

The second level of consistency, *full consistency*, considers both receive and send events of a sequence diagram, which occur as triggers and effects in state machines. It checks whether there exists a path in the communication between the instances of the state machines that represents the sequence of messages in the sequence diagram. This type of consistency depends on the interaction between the lifelines by message passing.

The definition of a *path* is relevant for both types of consistency. Since trigger consistency only relates to single state machines, the required definition of a path also relates only to states

of a single state machine. For full consistency, we need a different kind of path that relates sets of states where each set contains a state of each state machine of the model. We refer to the former as *trigger-based path* and to the latter as *path*.

3.3.1 Trigger Consistency

A trigger-based path considers only the triggers of a state machine.

Definition 3.3.1 (Trigger-based Path)**.** Given a state machine $M = (S, \iota, A^{tr}, A^{\mathit{eff}}, T)$, a trigger-based path in M is a sequence $[a_1, a_2, \ldots, a_n]$ of trigger symbols with $a_i \in A^{tr}$ for $i \in [1..n]$ such that there exists a sequence $[(s_1, a_1, A_1, s_2), (s_2, a_2, A_2, s_3), \ldots, (s_n, a_n, A_n, s_{n+1})]$ of transitions in M. △

Based on the definition of a trigger-based path we define trigger consistency as follows.

Definition 3.3.2 (Trigger Consistency of a Lifeline)**.** Let

- $D = (\mathcal{L}, \mathcal{N})$ be a well-formed and time-consistent event-based sequence diagram,
- $L = (l, M, E^{sd}, E^{rv}, >) \in \mathcal{L}$ be an event-based lifeline of D,
- $M = (S, \iota, A^{tr}, A^{\mathit{eff}}, T)$ be a state machine modeling the behavior of L, and
- $[e_1, \ldots, e_n]$ be the sequence of events where for all i, j with $1 \leq i, j \leq n$ it holds that $e_i, e_j \in E^{rv}$ and $e_i > e_j$ if and only if $i > j$.

Then the lifeline L is *trigger-consistent* with M if and only if there exists a trigger-based path $A = [a_1, a_2, \ldots, a_{n+m}]$ in M containing m occurrences of ϵ such that for the path $B = [b_1, b_2, \ldots, b_n]$ corresponding to A with all occurrences of ϵ removed it holds that $b_i = \mathsf{symb}(e_i)$ for $i \in [1..n]$. △

Example 3.3.1. In Figure 3.2.1, consider lifeline cm. The state machine instantiated by cm is CoffeeMachine (cf. Figure 3.1.2). The sequence of receive events on this lifeline is [orderCoffee1Rcv, doneRcv, orderCoffee2Rcv]. The sequence [orderCoffee, done, orderCoffee] of symbols from the messages connected to these events exists as path of triggers in CoffeeMachine connecting the states idle, preparing, error, maintenance, idle, preparing in this order. The lifeline cm is therefore trigger-consistent with its state machine. If message m4 was swapped with message m5, then cm would not be trigger-consistent, as from the only state that can be reached by a transition triggered by orderCoffee there is no outgoing transition triggered by orderCoffee. ◊

Definition 3.3.3 (Trigger Consistency)**.** A sequence diagram D is *trigger-consistent* if and only if

- D is well-formed,
- D is time-consistent, and
- all lifelines of D are trigger-consistent with respect to their state machine (cf. Definition 3.3.2.

△

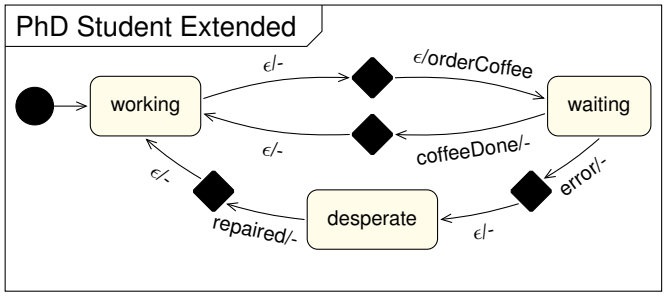

Figure 3.3.1: Extended state machine corresponding to the state machine PhD Student of Figure 3.1.2.

3.3.2 Full Consistency

Other than trigger consistency, *full consistency* of a sequence diagram also takes into account the communication between lifelines and therefore both the triggers and effects of the respective state machines. It verifies whether a sequence of messages can be sent and received by the set of state machines instantiated by the sender and receiver lifelines of the messages (state machines are duplicated when instantiated more than once). Such a sequence of messages moves along a *path* that connects *global states*, where each global state is a set of states containing one state for each instantiation of a state machine.

We first introduce the notion of an *extended state machine*. An extended state machine introduces an additional state for each transition in order to separate the events of receiving a symbol (receive event) and sending one or more symbols (send events). This facilitates a more intuitive description of the communication between state machines.

Definition 3.3.4 (Extended State Machine)**.** Given a state machine $M = (S, \iota, A^{tr}, A^{\mathit{eff}}, T)$, the *extended state machine* M^* is a state machine $(S \cup S^*, \iota, A^{tr}, A^{\mathit{eff}}, T^*)$ where

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

△

An extended state machine introduces an *intermediate state* s_t^* for each transition t of the original state machine. An intermediate state has exactly one incoming transition, which is triggered by the trigger of t and has no effect. It also has exactly one outgoing transition, which leads to the target state of t with no trigger and with the effects from t. An intermediate state can be identified by the source state, the target state, the trigger, and the effects of the transition. We call S the *original states* and S^* the *intermediate states* of a state machine. Any state machine can be translated to exactly one extended state machine and vice versa. We often refer to the extended state machine corresponding to a state machine M by M^*.

The extended state machine helps to distinguish between the event of having received the trigger and the event of being able to send the effects. Other than a non-extended state machine,

it can contain transitions which have both ϵ as trigger and the empty set as effects. These transitions connect intermediate states to original states when the corresponding transition of the non-extended state machine has the empty set as effect.

Figure 3.3.1 depicts the extended state machine of the state machine PhD Student. In concrete syntax, we represent the intermediate states by black diamonds with rounded corners.

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

Definition 3.3.5 (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 $1 \leq i \leq l$. △

Example 3.3.1 (Cont. from p. 24). A global state of the three extended state machines in Figure 3.1.2 instantiated by the lifelines of the sequence diagram in Figure 3.2.1 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. ◊

The communication between lifelines takes place through symbols received as triggers and sent as effects on their transitions. A transition t of a state machine M can only be executed when the trigger of t is received by M and all effects of t are received by state machines other than M. To capture this semantics, we define *multimessages* and their *admissibility* and *application*.

Definition 3.3.6 (Multimessage)**.** Given a set $\mathcal{M}^* = \{M_1^*, \ldots, M_l^*\}$ of extended state machines, the empty sender ε, and the set $\mathcal{A}^A$ of symbols, a *multimessage* over $\mathcal{M}^*$ is a pair $(\sigma, \{(a_1, \rho_1), \ldots, (a_k, \rho_k)\})$ where either

(1)
- $\sigma = M_d^*$ for some $d \in [1..l]$
- $\{(a_1, \rho_1), \ldots, (a_k, \rho_k)\} \in \mathcal{P}(A_d^{\mathit{eff}} \times \mathcal{M}^* \setminus \{\sigma\})$ such that for $1 \leq i \leq k$ all ρ_i are pairwise distinct, or

(2)
- $\sigma = \varepsilon$ and
- $\{(a_1, \rho_1), \ldots, (a_k, \rho_k)\} \in \mathcal{P}(\{\epsilon\} \times \mathcal{M}^*)$.

A message (σ, a, ρ) equals the multimessage $(\sigma, \{(a, \rho)\})$. △

A multimessage is a set of messages where either (1) symbols are sent from one state machine to a set of different state machines or (2) the empty symbol is received as trigger by one or more state machines. Case (1) corresponds to the set of effects on a transition of a state machine received as triggers by other state machines. Case (2) describes the "receipts" of the empty symbol as messages sent from empty senders in order to initiate transitions with ϵ as trigger. In the sequel we call this an empty multimessage.

In a global state of a collection of state machines, there exists a (possibly empty) set of multimessages that contains messages with symbols that occur as triggers and effects on the outgoing transitions of the particular states or with the symbol ϵ from the empty state machine ε. We call

a multimessage of this set *admissible* with respect to a global state. If this set is empty, then it means that no more communication between the state machines is possible. In the following definitions, it is important to separate the receipt and the sending of messages. They therefore refer to extended state machines rather than to regular state machines.

Definition 3.3.7 (Admissibility of a Multimessage)**.** Given a set $\mathcal{M}^* = \{M_1^*, \ldots, M_l^*\}$ of extended state machines with $M_i^* = (S_i, \iota_i, A_i^{tr}, A_i^{\mathit{eff}}, T_i)$ for all $i \in [1..l]$, and a global state $\hat{s} = (s_1, \ldots, s_l) \in S_1 \times \cdots \times S_l$, a multimessage $\mathcal{N} = (M_d^*, \{(a_1, \rho_1), \ldots, (a_k, \rho_k)\})$ over $\mathcal{M}^*$ is *admissible* in $\hat{s}$ if the following holds.

- If $M_d^* \neq \varepsilon$ then

  (1) $(s_d, \epsilon, \{a_1, \ldots, a_k\}, s_d') \in T_d$, and

  (2) there exists a set $\mathcal{R} \subseteq \{1, \ldots, l\} \setminus \{d\}$ and a bijective function $\mathsf{rec} : \{1, \ldots, k\} \to \mathcal{R}$ such that for $i \in [1..k]$ it holds that $\rho_j = M^*_{\mathsf{rec}(j)}$ and $(s_{\mathsf{rec}(j)}, a_j, \emptyset, s'_{\mathsf{rec}(j)}) \in T_{\mathsf{rec}(j)}$.

- Otherwise, if $M_d^* = \varepsilon$ then there exists a set $\mathcal{R} \subseteq \{1, \ldots, l\} \setminus \{d\}$ and a bijective function $\mathsf{rec} : \{1, \ldots, k\} \to \mathcal{R}$ such that for $j \in [1..k]$ it holds that $\rho_j = M^*_{\mathsf{rec}(j)}$ and $(s_{\mathsf{rec}(j)}, \epsilon, \emptyset, s'_{\mathsf{rec}(j)}) \in T_{\mathsf{rec}(j)}$.

△

There are two requirements for a multimessage with a sender other than ε to be admissible in a global state: (1) the sender's state in the global state is an extended state with an outgoing transition containing the set $\{a_1, \ldots, a_k\}$ of effects, and (2) each receiver's state in the global state has an outgoing transition triggered by the respective symbol from the multimessage. Note that we are dealing with extended state machines, which means that a transition cannot carry a trigger symbol other than ϵ together with a non-empty set of effects. Therefore it can never happen that a receiver state machine ρ_i sends any effects while executing the transition triggered by some symbol a_i. In order for an empty multimessage to be admissible, all receivers' states have to be in an intermediate or original state with an outgoing transition containing ϵ as trigger.

After *applying* an admissible multimessage, that is, after sending and receiving the symbols of the multimessage, a global state $\hat{s}'$ is reached as follows.

Definition 3.3.8 (Application of a Multimessage)**.** Given a set $\mathcal{M}^* = \{M_1^*, \ldots, M_l^*\}$ of extended state machines with $M_i^* = (S_i, \iota_i, A_i^{tr}, A_i^{\mathit{eff}}, T_i)$ for all $i \in [1..l]$, a global state $\hat{s} = (s_1, \ldots, s_l) \in S_1 \times \cdots \times S_l$, and a multimessage $\mathcal{N} = (M_d^*, \{(a_1, \rho_1), \ldots, (a_k, \rho_k)\})$ over $\mathcal{M}^*$ that is admissible in $\hat{s}$, a *global successor state* $\hat{s}'$ of $\hat{s}$ after *applying* $\mathcal{N}$ is given by $\hat{s}' = (\mathsf{next}(s_1), \ldots, \mathsf{next}(s_l))$ where for $i \in [1..l]$

(1) $\mathsf{next}(s_i) = s_d'$ if $M_d^* \neq \varepsilon$ and $i = d$,

(2) $\mathsf{next}(s_i) = s_i'$ if $i \in \mathcal{R}$, and

(3) $\mathsf{next}(s_i) = s_i$ otherwise.

△

The global successor state $\hat{s}'$ is reached from $\hat{s}$ by applying a multimessage. It can differ from $\hat{s}$ in the sender's state (unless the sender is ε) and the receivers' states reached by transitions that carry as effects or as trigger, respectively, a symbol of the applied multimessage. Case (1) defines that unless the sender is ε, the sender's state changes from an extended state to its only successor state, case (2) that the receivers' states change according to the received symbol into an intermediate state, and case (3) that all other state machines remain in their current state.

Example 3.3.1 (Cont. from p. 24). Consider the three lifelines $(\mathsf{alice}, \mathsf{PhD})$, $(\mathsf{cm}, \mathsf{CM})$, and $(\mathsf{m}, \mathsf{Maintenance})$ instantiating the state machines in Figure 3.1.2, and the global state $\hat{s} = (\mathsf{desperate}, \text{<maintenance/done/repaired/idle>}, \mathsf{idle})$ of their extended state machines. The set of admissible messages from $\hat{s}$ contains only the message $\{((\mathsf{cm}, \mathsf{CM}), \mathsf{repaired}, (\mathsf{alice}, \mathsf{PhD}))\}$. There are three outgoing transitions from states contained in $\hat{s}$, and only those from the states in the state machines instantiated by cm and by alice can send or receive a symbol. The symbol received by the outgoing transition from the state idle in Maintenance is not sent by any outgoing transition of another state in $\hat{s}$. Applying N to the global state $\hat{s}$ reaches the global successor state $(\text{<desperate/repaired/}\epsilon\text{/working>}, \mathsf{idle}, \mathsf{idle})$. ◊

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

Definition 3.3.9 (Transaction)**.** A *transaction* is a nonempty set $\{\mathcal{N}_1, \ldots, \mathcal{N}_l\}$ of multimessages with $\mathcal{N}_i = (\sigma_i, \{(a_{i,1}, \rho_{i,1}), \ldots, (a_{i,k_i}, \rho_{i,k_i})\})$ such that for all $i \in [1..l]$ and $j \in [1..k_i]$ it holds that $\sigma_i \neq \rho_{i,j}$, i.e., all state machines occurring in the multimessages are distinct. △

A transaction is admissible if all its multimessages are admissible. The global state reached by applying a transaction is the global state reached by applying each of the transaction's multimessages.

Along a *path* we can step through global states of a set of state machines.

Definition 3.3.10 (Path)**.** A *path μ from a global state $\hat{s}_0$ to a global state $\hat{s}_k$* is a sequence $\mu = [\mathcal{C}_1, \ldots, \mathcal{C}_k]$ of transactions such that there exists a sequence $[\hat{s}_0, \ldots, \hat{s}_k]$ of global states where for all $i \in [1..k]$, $\mathcal{C}_i$ is admissible in state $\hat{s}_{i-1}$ and $\hat{s}_i$ is the global successor state of $\hat{s}_{i-1}$ after applying $\mathcal{C}_i$. △

A path is a sequence of transactions connecting global states. The *length* of a path is the number of its transactions. Along a path, a global state is *reachable* from another global state.

Definition 3.3.11 (Reachability)**.** A global state $\hat{s}_j$ *is reachable from* $\hat{s}_i$ if there is a path from $\hat{s}_i$ to $\hat{s}_j$. △

Example 3.3.1 (Cont. from p. 24). For the state machines Coffee, PhD Student, and Maintenance of Figure 3.1.2, the global state $(\mathsf{waiting}, \text{<idle/orderCoffee/}\emptyset\text{/preparing>}, \mathsf{idle})$ is reachable by the path $[(\varepsilon, \{(\epsilon, \mathsf{PhD})\}), (\mathsf{PhD}, \{(\mathsf{orderCoffee}, \mathsf{CM})\})]$ from the global state $(\mathsf{waiting}, \mathsf{idle}, \mathsf{idle})$. ◊

A sequence diagram can be understood to model a path in a set of instantiations of state machines. In this case a message of the sequence diagram is interpreted as a multimessage with

a singleton set of effects. We connect the two views by defining k-*consistency* and *full consistency* between them. Intuitively, a sequence diagram is consistent to a set of state machines, if the sequence of messages occurs somewhere along a path between the global states of the instantiations of the state machines such that the beginning of this path is reachable from the global initial state, which contains each state machine's initial state. The views are k-*consistent* if this holds for a path of length at most k.

Definition 3.3.12 (k-Consistency)**.** Given a set $\mathcal{M}$ of extended state machines and a sequence diagram $D = (\mathcal{L}, \mu)$ over $\mathcal{M}$ with $\mathcal{L} = \{L_1, \ldots, L_l\}$ and $\pi_2(L_i) = (S_i, \iota_i, A_i^{tr}, A_i^{\mathit{eff}}, T_i)$ for $i \in [1..l]$ it holds that D 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 only each message of the sequence μ in the order of the sequence and zero or more empty messages in between the messages. △

If the path can be of arbitrary length, then the two views are fully consistent.

Definition 3.3.13 (Full Consistency)**.** Given a set $\mathcal{M}$ of extended state machines and a sequence diagram $D = (\mathcal{L}, \mu)$ over $\mathcal{M}$ with $\mathcal{L} = \{L_1, \ldots, L_l\}$ and $\pi_2(L_i) = (S_i, \iota_i, A_i^{tr}, A_i^{\mathit{eff}}, T_i)$ for $i \in [1..l]$ it holds that D and $\mathcal{M}$ are *fully consistent* if there exists a path 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 only each message of the sequence μ in the order of the sequence and zero or more empty messages in between the messages. △

The following example illustrates this notion of consistency.

Example 3.3.1 (Cont. from p. 24). The sequence diagram of Figure 3.1.4 is inconsistent with the state machines of Figure 3.1.2. There is no path in instantiations of the state machines that contains the sequence of messages modeled in the sequence diagram. However, the sequence diagram containing only the first four messages of those modeled in Figure 3.1.4 is k-consistent for $k = 9$ (and therefore also fully consistent) with the state machines. They model the path

$[(\varepsilon, \{(\epsilon, \mathsf{PhD})\}),$
$(\mathsf{PhD}, \{(\mathsf{orderCoffee}, \mathsf{CM})\}),$
$(\varepsilon, \{(\epsilon, \mathsf{CM})\}),$
$(\varepsilon, \{(\epsilon, \mathsf{CM})\}),$
$(\mathsf{CM}, \{(\mathsf{error}, \mathsf{PhD})\}),$
$(\varepsilon, \{(\epsilon, \mathsf{CM}), (\epsilon, \mathsf{PhD})\}),$
$(\mathsf{CM}, \{(\mathsf{repair}, \mathsf{Maintenance})\}),$
$(\varepsilon, \{(\epsilon, \mathsf{CM}), (\epsilon, \mathsf{Maintenance})\}),$
$(\mathsf{Maintenance}, \{(\mathsf{done}, \mathsf{CM})\})]$

containing nine transactions. ◊

With these formal definitions at hand, in the next chapter we will formally describe three consistency problems related to trigger consistency and k-consistency.

Chapter 4

Verification Problems in Software Modeling

A major difference between traditional software engineering and model-driven software engineering (MDE) 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 these models are expressed in multi-view modeling languages like the UML. The goal of MDE is to leverage the abstraction power of these models in order to deal with the complexity of modern software systems [13], and to further exploit the models to automatically generate executable code with little or no intervention of a developer [109].

With this increasing valorization of software models, stronger requirements on their correctness come along. At the same time, in their role as core artifacts, the software models are increasingly sensitive to the impact of evolution [52]. Model management tasks such as synchronization, versioning, or co-evolution can involve changes in one view of the model that result in inconsistencies with another view of the model [89], which due to their multi-view nature and the size of the software, are often hard to spot for a human developer. However, in particular when the models are employed in automatic code generation, such inconsistencies propagate to the executable system and can result in serious errors. Therefore, automated methods are required to alleviate the developer from responsibilities. Such methods can be applied in different evolution scenarios, e.g., they can directly support necessary model management tasks or verify the result of a (possibly manually) performed task.

In this chapter, we propose automated methods that tackle three different problems, one directly supporting a model management task and two for verification purposes that can be applied in different evolution scenarios. First, the *Sequence Diagram Merging (*SDMERGE*) Problem* occurs in the management task of model versioning. In an evolution scenario where two versions of a sequence diagram have been created by different developers this task requires the two versions to be merged into a new sequence diagram which considers the changes of both developers

and which is trigger-consistent with the state machines its lifelines instantiate (cf. Section 3.3, Definition 3.3.3). Second and third, we consider two verification problems which rely on the semantics of full consistency and k-consistency (cf. Section 3.3, Definitions 3.3.13 and 3.3.12), the *State Machine Reachability (*SMREACH*) Problem*, and the *Sequence Diagram Model Checking (*SDCHECK*) Problem*. The SMREACH problem asks whether a certain configuration of state machines, i.e., a combination of at most one state per state machine, is reachable from some global state. We call such a configuration a *partial global state*. An instance of the SMREACH problem takes as input a set of state machines and a partial global state over this set. The SDCHECK problem applies the semantics of full consistency to ask whether a sequence of messages described in a sequence diagram can be executed from some global state of the instantiations of the state machines such that this global state is reachable from the *global initial state*. The global initial state is the global state where all state machines are in their initial state.

We bound both of these problems by the parameter k. The k-SMREACH problem asks whether a partial global state over a set of state machines is reachable *by a path of length at most* k from some global state, and the k-SDCHECK problem asks whether a sequence diagram can be executed after a path of length at most k from the global initial state.

In order to solve these problems, we propose encodings to the satisfiability problem of propositional logic (SAT). Over the last years, propositional logic has proven to be a powerful host language for a wide range of real-life problems like verification and planning, particularly because of the availability of efficient and stable solvers [102]. The formula encoding the problem can then be handed to an off-the-shelf solver. If a solution exists, the solver returns a logical model which can be translated into a concrete solution to the problem. Otherwise, the solver reports the formula to be unsatisfiable, which means that no solution to the encoded problem exists. In addition to help finding a solution to the problem, the SAT encodings provide a formal definition of the semantics of the two properties trigger consistency and k-consistency.

The semantic differences between the SDMERGE problem and the other two problems result in differences between the encoding of the former problem and the encodings of the latter two problems. On the other hand, the semantic similarities between the k-SMREACH problem and the k-SDCHECK problem result in many similarities of the encodings of the two problems. Indeed, only few adaptations are needed to convert the encoding of the k-SMREACH problem into an encoding of the k-SDCHECK problem.

We first discuss work related to the three problems. Then, for each problem, we give a problem definition accompanied by a small example, describe an encoding to propositional logic, and prove its computational complexity. We devote a separate chapter, Chapter 5, to an in-depth evaluation of our method.

4.1 Related Work

Work related to the problems tackled in this chapter stems from two fields of research. In the field of *model versioning* we are dealing with results regarding the detection of differences between independently changed artifacts and regarding the consolidation of the changes. The field of *model verification* covers many kinds of syntactic and semantic consistency checks within one or more views of a software model. In both fields the notion of "consistency" is central,

but its definitions are highly heterogeneous, depending on the type of diagrams under consideration and covering only syntactic aspects in some cases, and semantic aspects in others. It is therefore often difficult or impossible to compare the different approaches with respect to their performance.

4.1.1 Model Versioning

The inherent complexity of the software development process [100] has resulted in a variety of tools supporting team work and change management [35]. In particular, *version control systems* (VCS) provide a powerful means to assist the evolution of software by automated merging and tracking of changes introduced by multiple developers. Versioning systems that manage parallel modifications on a software artifact are called *optimistic versioning systems*. Often, they provide sophisticated means of conflict resolution by comparing and merging the independently evolved versions with a common ancestor. In traditional software engineering, VCS are applied to textual artifacts such as source code. MDE, however, puts software models into a more central position and therefore creates a need to version control them too. However, mainly due to the graph-based nature of models, the requirements of model versioning systems strongly diverge from the requirements of traditional versioning systems for text-based artifacts [2, 7, 9].

In the past decade, several model versioning systems based on different approaches have been proposed. In particular, exploitation of the graph-based nature of the models and analysis of composite changes in order to detect and automatically resolve conflicts has shown to be successful [24, 30]. Most approaches, however, mainly target the syntactic part of a model and mostly neglect the semantics such as inter-view consistency. In the following, we give a short overview on model versioning approaches.

Westfechtel [121] discusses the merge of ordered features in models of the Eclipse Modeling Framework by aggregating elements into linearly ordered clusters. The order within a cluster is determined either at random or by a user. However, the merge is performed on the metamodel level in order to keep the approach generic, and therefore the information available within the model cannot be used for merging. Gerth et al. [55] provide dedicated merge support for business process models ensuring a consistent outcome. They formalize process models as process terms and use a term rewriting system to detect and interactively resolve merge conflicts. However, there is no support to compute all valid merge solutions. Cicchetti et al. [30] propose to define conflict patterns which can be tailored towards the application on sequence diagrams. Such a conflict pattern can be equipped with a reconciliation strategy for resolving the conflict.

Nejati et al. [96] present an approach to merge two state machines. Their approach exploits syntactical as well as semantic information provided by the models in order to compare variants and perform consistency checks.

Reiter et al. [103] suggest *semantic views* constructed via a manual normalization process. Two different revisions of a model are normalized according to this process and compared to detect conflicts. Semantics are considered in the normalization process, but the later comparison only considers syntactic features. Maoz et al. [92] define semantic equivalence, which can be independent of syntactic differences. Their work focuses on the task of differencing, which is one part of merging process, but they do not cover the whole merging process. Further, an in-depth survey on model versioning has been published by Brosch et al. [23].

To the best of our knowledge, no previous work has considered inter-view consistency between sequence diagrams and state machines for the merging process. We consider such an approach in Section 4.2 of this work.

4.1.2 Model Verification

Besides approaches tackling problems of specific model management tasks such as model versioning, many model verification techniques can be employed to support various model management tasks. For example, the question whether a sequence diagram is still consistent with a set of state machines can be asked after any evolution step that contains any kind of changes to either of the views.

Related work verifying the consistency of a software model covers a very broad field, possibly due to the lack of common consent on the meaning of the term "consistency". It can be roughly categorized along two dimensions. On the one hand, there are approaches performing only syntax checks, for example in works by Egyed [44] and by Mens et al. [94] and such taking semantics or behavioral aspects into account. On the other hand some approaches focus on one single view and others consider multi-view consistency. In this section, we focus on related work with respect to semantics and behavior for both single and multi-view consistency problems.

In the category of single-view consistency checking, Cabot et al. [29] verify the behavioral aspects of UML class diagrams annotated with *operation contracts*, which are declarative descriptions of operations specified as pre- and postconditions in the Object Constraint Language (OCL), an extension of the UML to define syntactic components for which the UML metamodel is not expressive enough. Other single-view approaches tackle the state machine or the sequence diagram view by applying model checking [33]. Alur and Yannakakis [3] present theoretical results on model checking of hierarchical state machines. In particular, they establish that reachability analysis and checking of linear properties, which are both applications of model checking, can be performed without flattening the state machines. Lilius and Porres [86] formalize UML 1.0 state machines in order to employ the SPIN model checker [67] to check for various properties of the model. A formal verification technique for UML 2.0 sequence diagrams employing linear temporal logic (LTL) formulas and the SPIN model checker to reason about the occurrences of events is introduced by Lima et al. [87]. Ter Beek et al. [112] present their own model checking framework to check for event-based logic constraints in state machines and so do Zhang and Liu [125] to check for safety properties in UML state machines. Symbolic encodings for the model checker nuSMV [31] have been proposed for hierarchical state machines [40] and activity diagrams [46].

Concerning multi-view consistency checking, Diskin et al. [39] present a framework based on category theory for consistency checking between views. They integrate the relevant parts of the models into one global metamodel such that all instance models become instance models thereof. These instance models can then be checked for inconsistencies. Van Der Straeten et al. [118] use the SAT-based constraint solver *Kodkod* to detect and resolve inconsistencies between class and sequence diagrams. Egyed [43] applies instant consistency validation by rules formulated in OCL which shows to be very efficient on large models. Sabetzadeh et al. [107] present an approach to check consistency between a set of different, but overlapping models by merging this set of models into one model. Tsiolakis [115] suggests to collect constraints dis-

tributed over views like the class diagram or the state machine and integrate them in a sequence diagram in terms of state invariants yielding pre- and postconditions for individual messages.

Approaches considering inter-view consistency between state machines and sequence diagrams include the following. Lam and Vitus [82] present an algebraic approach to express the consistency checking problem in the π-calculus, but they do not discuss the practical realization of their approach. Van der Straeten et al. [117] propose to use description logics to formally describe the consistency between class diagrams, sequence diagrams, and state machines. Bernardi et al. propose to use Petri nets for checking the consistency between different diagrams [12]. Communication, however, is only considered at the class level and not at the instance level. Engels et al. [45] propose to check consistency by evaluating dedicated consistency constraints represented in form of collaborations. Graaf and Van Deursen [58] and Whittle and Schumann [122] suggest to synthesize a state machine from a given sequence diagram and then compare the automatically generated state machine to the given state machine. Therefore, they realize normalization, transformation, and comparison steps, respectively. In [58], however, the comparison requires manual intervention. Further, Feng and Vangheluwe propose to use a simulation-based approach for consistency checking [48].

In particular for consistency checking between the state machine and the sequence diagram views, the use of *model checking* [33] is very popular. Usually, the model checkers provide languages to describe finite state automata, which are also the conceptual basis of state machines. Inverardi et al. [68] present their own definition of a multi-view modeling, simulation, and verification environment, the software architecture (SA), describing the static and behavioral structures of systems with component, state transition, and sequence diagrams. They propose to employ the model checker SPIN [67] to check for different inter-view consistency properties. The tools HUGO [108] and CHARMY [101] also employ the model checker SPIN [67]. HUGO verifies whether the interactions of a UML 1.0 collaboration diagram are consistent with a set of state machines. The tool automatically translates the state machine diagrams to PROMELA, the input language of SPIN, and generates "never claims" from the collaboration diagrams, which are then verified by SPIN. Another version of HUGO [79, 80] is based on the model checker UPPAAL [84]. Other than HUGO, which uses a UML based definition of collaboration diagrams and state machines, CHARMY builds on the previously introduced SA [68]. CHARMY also translates the modeled artifacts to PROMELA and calls SPIN to either locate deadlocks and unreachable states in the state machines, or to verify temporal properties of the system. Finally, we also proposed to verify inter-view consistency between sequence diagrams and state machines [21, 22].

Most of these tools have been discontinued with the introduction of UML 2.0 or are unavailable. In our related works [21,22] we experienced that one of the major challenges of approaches based on expressing state machines in an input language of model checkers is to overcome semantic heterogeneities between the two representations. However, other than having to deal with this intermediate step, the state machine model could be encoded in a more low-level language such as propositional logic in order to avoid such semantic restrictions. Indeed, this approach is proposed and shown to be more efficient than translations to standard model checkers by Niewiadomski et al. [97]. In Sections 4.3 and 4.4, we follow this idea of symbolic encodings, but we propose an alternative encoding inspired by works on solving planning problems [105].

4.2 Guided Merging of Sequence Diagrams – the SDMERGE Problem

This section presents the *Sequence Diagram Merging (*SDMERGE*) Problem*. The objective of solving this problem is to compute a sequence diagram that considers (possibly conflicting) changes applied by two different developers to the same initial sequence diagram and that is trigger-consistent with the state machines it instantiates. In our problem formulation, we consider as changes only additions of messages and lifelines to a sequence diagram. A formulation that also considers deletions could result in a more complex problem.

We first give an intuition of the problem with a small example based on three state machines that implement a simplified version of an email protocol. We then give a formal problem definition followed by an encoding of the problem to the satisfiability problem of propositional logic (SAT). The encoding expresses the semantics of the property of trigger consistency (cf. Section 3.3, Definition 3.3.3), and therefore forms parts of the formal semantics of the *tMVML*. Finally, we show that the SDMERGE problem is solvable in time polynomial in the input size. Given this result, a SAT solver may not seem to be the appropriate tool to tackle this problem. However, apart from solving the SDMERGE problem, our SAT encoding serves to define the semantics of trigger consistency and, given the rather basic problem formulation, it will later serve as a foundation to encode more complex problem formulations.

4.2.1 A Motivating Example

Figure 4.2.1 shows three state machines implementing a simplified variant of the *Simple Mail Transfer Protocol* (SMTP). The state machine Client starts in state Idle and waits until it receives uCon, which triggers its transition to state conPend. During the execution of the transition it sends the symbol sCon, which is received by the state machine Server and triggers its transition from state waiting to accepting. During the execution of this transition Server sends the symbol ok, which is again received by Client, triggering the transition to state connected, and so on.

The state machine User represents the (human) interaction with the state machine Client. It has only one state from which it can send any of a set of commands to Client.

Figure 4.2.2 shows three sequence diagrams D_o, D_α, and D_β. A sequence diagram is trigger-consistent with the state machines instantiated by its lifelines if for each lifeline the sequence of received messages is a trigger-based path in the corresponding state machine (cf. Section 3.3, Definitions 3.3.1 and 3.3.2). For the uppermost diagram in Figure 4.2.2, D_o, the sequence of received messages for lifeline c:Client corresponds to the sequence $[\mathsf{uCon}, \mathsf{ok}, \mathsf{ok}]$ of triggers. This sequence can be found as a trigger-based path in the state machine Client connecting states idle $\rightarrow$ conPend $\rightarrow$ connected $\rightarrow$ identified. A similar argument holds for the lifeline s:Server and the sequence $[\mathsf{sCon}, \mathsf{sHello}]$ of triggers. The lifeline u:User instantiates the state machine User, which never receives, but only sends symbols. Therefore, this lifeline is also consistent.

Based on the sequence diagram D_o, the following evolution scenario starts. Two modelers, Alice and Bob, independently perform some modifications. Alice extends the scenario with a logout message resulting in the revised sequence diagram D_α, and Bob adds a communication

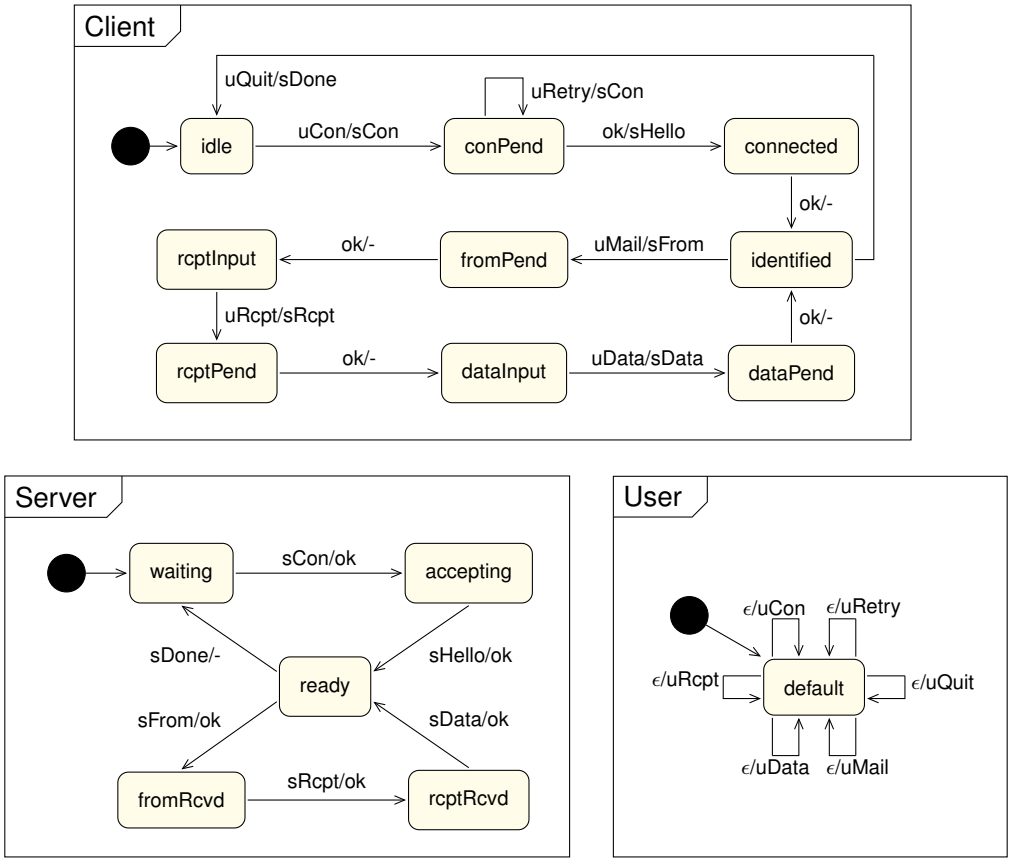

Figure 4.2.1: State machines of an email client, an email server, and a user.

to send an email, resulting in D_β. Trying to merge the modifications of both modelers without any additional information, it cannot be decided in a straightforward manner in which order the added messages from both revisions should be arranged. Hence, we have a *merge conflict*.

Several time-consistent merges of the sequence diagram are possible. In particular, all permutations of the two concatenated message sequences are time-consistent. We further assume that the modelers wish to preserve the relative order of their added messages and therefore, out of the time-consistent merges, we consider only such merges that fulfill this constraint. However, many of such merges are trigger-inconsistent with the state machines. For example, the sequence

$$[\mathsf{uCon}, \mathsf{sCon}, \mathsf{ok}, \mathsf{sHello}, \mathsf{ok}, \mathsf{uQuit}, \mathsf{uMail}, \mathsf{sFrom}, \mathsf{ok}, \mathsf{uRcpt}, \mathsf{sRcpt}, \mathsf{ok}, \mathsf{uData}, \mathsf{sData}, \mathsf{sDone}, \quad \mathsf{ok}]$$

is a permutation of all original and all added messages that preserves the relative order of the additions, but it is not trigger-consistent with the instantiated state machines.

Indeed, Alice's changes have to be appended after Bob's changes, as in any other possible merging scenario the resulting sequence diagram models a scenario which is forbidden by the state machines.

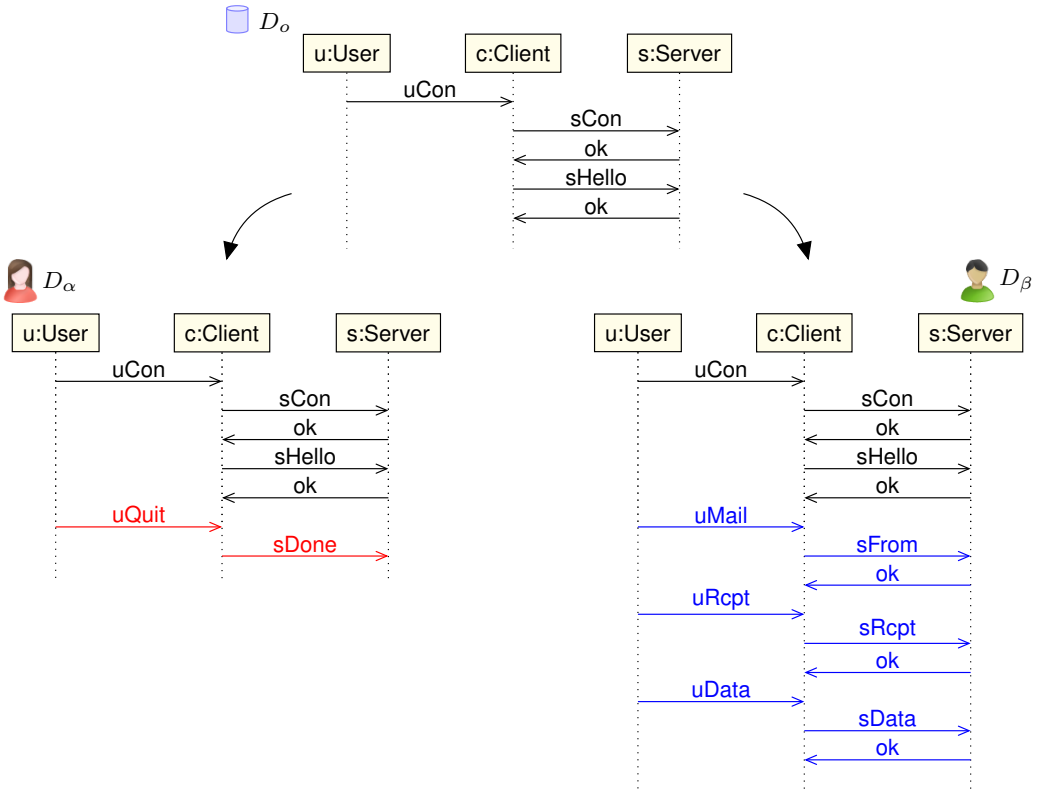

Figure 4.2.2: Evolution of a sequence diagram.

4.2.2 Problem Definition

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

In this section, we work only with the event-based definitions of a sequence diagram, that is, with Definition 3.2.2 of an event-based lifeline, Definition 3.2.3 of an event-based message, and Definition 3.2.4 of an event-based sequence diagram, all of which occur in Section 3.2. Hence, in the current section, when we write "lifeline", "message", or "sequence diagram" we mean the event-based versions according to the above definitions.

First, we define a *revision* of a sequence diagram as an extension of a sequence diagram by messages and lifelines.

Definition 4.2.1 (Revision)**.** A sequence diagram $D_\alpha = (\mathcal{L}_\alpha, \mathcal{N}_\alpha)$ defined over the universe $\mathcal{A}$ is a *revision* of a trigger-consistent sequence diagram $D_o = (\mathcal{L}_o, \mathcal{N}_o)$ also defined over $\mathcal{A}$ if and only if $\mathcal{L}_o \subseteq \mathcal{L}_\alpha, \mathcal{N}_o \subseteq \mathcal{N}_\alpha$, and D_α is trigger-consistent. △

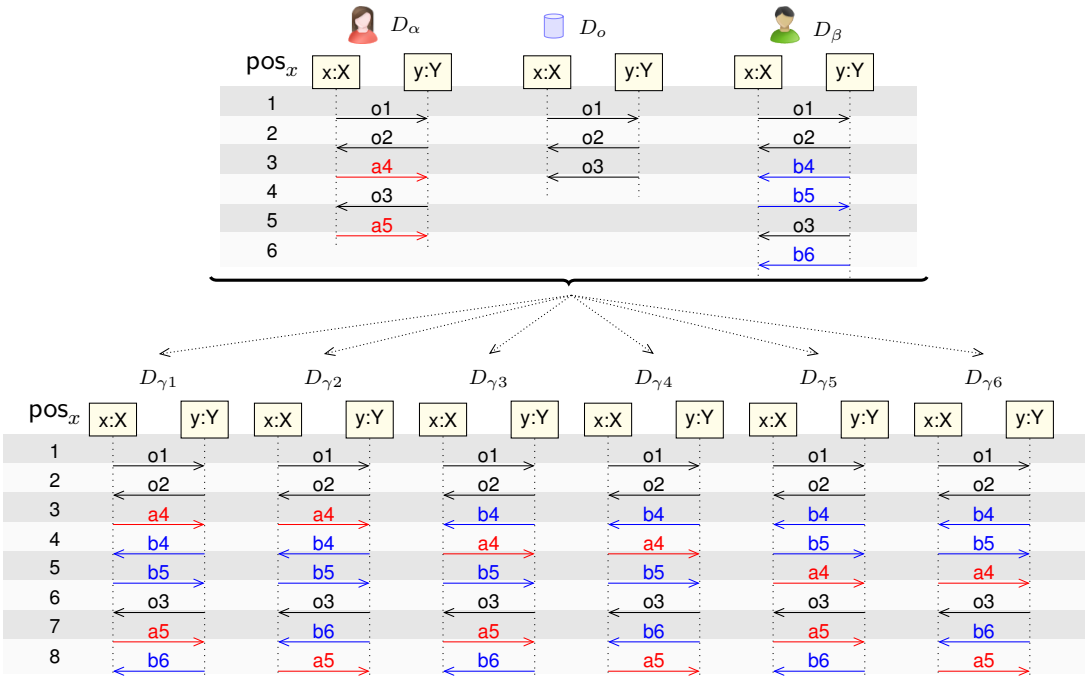

Figure 4.2.3: Sequence diagram D_o and two revisions D_α and D_β (top) with their six time-consistent, but not necessarily trigger-consistent merges (bottom) and the values of the function pos_x for $x \in \{o, \alpha, \beta, \gamma1, \ldots, \gamma6\}$.

Example 4.2.1. In Figure 4.2.2, the sequence diagrams D_α and D_β are revisions of the sequence diagram D_o. In D_α, a sequence of messages representing a cancellation of the connection, and in D_β, a sequence of messages representing the receipt of an email have been added. ◊

A consolidated version of a sequence diagram and two of its revisions is a trigger-consistent sequence diagram that contains the messages and lifelines of the original sequence diagram and all added messages and lifelines from the revisions. The order of messages relative to the original diagram and to the revisions is maintained.

In the rest of this section, we use the position function pos to refer to the position of a message in a sequence diagram.

Definition 4.2.2 (Position of a Message)**.** Given a time-consistent sequence diagram $D = (\mathcal{L}, \mathcal{N})$, the position function $\mathsf{pos} : \mathcal{N} \to \{1, \ldots, |\mathcal{N}|\}$ maps a message to an integer such that for all $N, N' \in \mathcal{N}$ it holds that $\mathsf{pos}(N) = \mathsf{pos}(N')$ if and only if $N = N'$ and $\mathsf{pos}(N) > \mathsf{pos}(N')$ if and only if $N \succ N'$. △

In a consolidated version, each of the messages added to one of the revisions can be placed on one of a set of positions. This set of positions maintains the relative order of the messages in the revisions. We return this set of positions for each message by the function allow.

Definition 4.2.3 (Allowed Positions)**.** Given three trigger-consistent sequence diagrams $D_x = (\mathcal{L}_x, \mathcal{N}_x)$, for $x \in \{o, \alpha, \beta\}$, where D_α and D_β are revisions of D_o, and $\mathsf{pos}_x : \mathcal{N}_x \to \{1, \ldots, |\mathcal{N}_x|\}$ with $x \in \{o, \alpha, \beta\}$ are the position functions, let $\mathcal{N} = \mathcal{N}_o \cup \mathcal{N}_\alpha \cup \mathcal{N}_\beta$ and $I = \{1, 2, \ldots, |\mathcal{N}|\}$. Then $\mathsf{allow} : \mathcal{N} \to \mathcal{P}(I)$ assigns to each message N a set of positions, such that

- if $N \in \mathcal{N}_o$ and $\mathsf{pos}_o(N) = \mathsf{pos}_\alpha(N) = \mathsf{pos}_\beta(N)$ then $\mathsf{allow}(N) = \{\mathsf{pos}_o(N)\}$ (N remains at the same position),
- if $N \in \mathcal{N}_o$ and $\mathsf{pos}_o(N) \neq \mathsf{pos}_\alpha(N)$ or $\mathsf{pos}_o \neq \mathsf{pos}_\beta(N)$, then for $\mathcal{N}' = \{M \in \mathcal{N}_\alpha \mid \mathsf{pos}_\alpha(M) < \mathsf{pos}_\alpha(N)\} \cup \{M \in \mathcal{N}_\beta \mid \mathsf{pos}_\beta(M) < \mathsf{pos}_\beta(N)\}$, it holds that $\mathsf{allow}(N) = \{\mathsf{pos}_o(N) + |\mathcal{N}'|\}$ (N is shifted to a position such that messages which have been inserted into revisions before N can be placed on positions before N in the consolidated version),
- if $N \notin \mathcal{N}_o$ and $N \in \mathcal{N}_\alpha$ then for
  - $N' \in \mathcal{N}_o$ such that $\mathsf{pos}_\alpha(N') = \max_{N \in \mathcal{N}_o \mid \mathsf{pos}_\alpha(n) < \mathsf{pos}_\alpha(N)} \mathsf{pos}(N)$,
  - $N'' \in \mathcal{N}_o$ such that $\mathsf{pos}_o(N'') = \mathsf{pos}_o(N) + 1$,
  - $\mathcal{N}' = \begin{cases} \{N \in N_\beta \mid \mathsf{pos}_\beta(N) < \mathsf{pos}_\beta(N')\} & \text{if } \mathsf{pos}_\alpha(N') > 1 \\ \emptyset & \text{otherwise} \end{cases}$
    ($\mathcal{N}'$ contains all messages that were added to the other revision before the closest predecessor of N from $\mathcal{N}_o$), and
  - $\mathcal{N}'' = \begin{cases} \{N \in \mathcal{N}_\beta \mid \mathsf{pos}_\beta(N) < \mathsf{pos}_\beta(N'')\} & \text{if } \mathsf{pos}_\alpha(N'') < |\mathcal{N}_\alpha| \\ \mathcal{N}_\beta \setminus \mathcal{N}_o & \text{otherwise} \end{cases}$
    ($\mathcal{N}''$ contains all messages that were added to the other revision before the closest successor of N from $\mathcal{N}_o$),

  it holds that $\mathsf{allow}(N) = \{i \in I \mid \mathsf{pos}_\alpha(N) + |\mathcal{N}'| \leq i \leq \mathsf{pos}_\alpha(N) + |\mathcal{N}''|\}$, and
- if $N \notin \mathcal{N}_o$ and $N \in \mathcal{N}_\beta$ then allow is defined as for the case $N \notin \mathcal{N}_o$ and $N \in \mathcal{N}_\alpha$ but with α and β exchanged in the whole definition.

△

Example 4.2.2. Consider the sequence diagrams D_o, D_α and D_β shown in the upper part of Figure 4.2.3, where D_α and D_β are revisions of D_o. In D_α, the message a4, and in D_β the messages b4 and b5 are added between the original messages o2 and o3. In a merged sequence diagram, each of a4, b4 and b5 must again be placed between o2 and o3. Also, in order to maintain their order from the revisions, b5 must be placed after b4. Similar conditions are given for the messages a5 and b6 inserted after o3. Table 4.2.1 shows the values of the pos and allow functions for each message N in the sequence diagrams D_o, D_α, and D_β of Figure 4.2.3. ◊

If in the merged sequence diagram each message N is placed on one of the positions defined in $\mathsf{allow}(N)$ and exactly one message has been placed at each position, then the merged sequence diagram is time-consistent (cf. Section 3.2, Definitions 3.2.5, 3.2.6, 3.2.7). However, in order for the merged sequence diagram to be trigger-consistent, the messages have to be placed such that the lifelines are trigger-consistent with respect to their state machines. If this is also the case, then the merged diagram is a *consolidated version*, defined as follows.

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

Table 4.2.1: Allowed positions for each message of Figure 4.2.3.

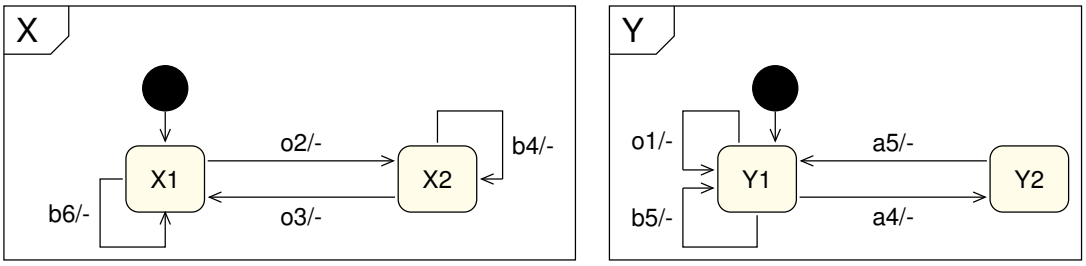

Figure 4.2.4: The state machines modeling the behavior of the lifelines in Figure 4.2.3.

Definition 4.2.4 (Consolidated Version)**.** Given the trigger-consistent sequence diagrams $D_o = (\mathcal{L}_o, \mathcal{N}_o)$, $D_\alpha = (\mathcal{L}_\alpha, \mathcal{N}_\alpha)$, and $D_\beta = (\mathcal{L}_\beta, \mathcal{N}_\beta)$, where D_α and D_β are revisions of D_o, a consolidated version $D_\gamma = (L_\gamma, \mathcal{N}_\gamma)$ is a sequence diagram where

- $\mathcal{L}_\gamma = \mathcal{L}_\alpha \cup \mathcal{L}_\beta$,
- $\mathcal{N}_\gamma = \mathcal{N}_\alpha \cup \mathcal{N}_\beta$,
- for each $N \in \mathcal{N}_\gamma$ it holds that $\mathsf{pos}(N) \in \mathsf{allow}(N)$, and
- S_γ is trigger-consistent.

△

Example 4.2.2 (Cont. from p. 40). Consider the example shown in Figures 4.2.3 and 4.2.4. The upper part of Figure 4.2.3 depicts the original sequence diagram D_o and two of its revisions D_α and D_β with the values of the respective pos_x function. The revised diagrams contain added messages between messages o2 and o3 and at the end of the diagram. The lower part of the figure depicts six different time-consistent merged diagrams $D_{\gamma 1}$ to $D_{\gamma 6}$. Figure 4.2.4 shows two state machines describing the behavior of the lifelines. The two merged diagrams $D_{\gamma 5}$ and $D_{\gamma 6}$ are also consolidated versions, i.e. they are trigger-consistent with respect to the state machines

depicted in Figure 4.2.4. It can be verified that the sequence of message symbols received by lifeline y of the rightmost diagram, i.e., $[\mathsf{o1}, \mathsf{b5}, \mathsf{a4}, \mathsf{a5}]$ occurs as trigger-based path in state machine Y and so does the sequence of lifeline x, i.e., $[\mathsf{o2}, \mathsf{b4}, \mathsf{o3}, \mathsf{b6}]$. However, the other merged diagrams $D_{\gamma 1}$ to $D_{\gamma 4}$ are not trigger-consistent. For example, the sequence $[\mathsf{o1}, \mathsf{a4}, \mathsf{b5}, \mathsf{a5}]$ of lifeline y cannot be found as trigger-based path in state machine Y. ◊

Finally, the Sequence Diagram Merging Problem is defined as follows.

Definition 4.2.5 (*Sequence Diagram Merging (*SDMERGE*) Problem*)**.**
Instance: The universe $\mathcal{A}$, a set $\mathcal{M}$ of state machines over $\mathcal{A}$, and three trigger-consistent sequence diagrams $D_o = (\mathcal{L}_o, \mathcal{N}_o)$, $D_\alpha = (\mathcal{L}_\alpha, \mathcal{N}_\alpha)$, and $D_\beta = (\mathcal{L}_\beta, \mathcal{N}_\beta)$ over $\mathcal{A}$ and $\mathcal{M}$, where D_α and D_β are revisions of D_o.
Question: Does there exist a consolidated version of D_o, D_α, D_β?

△

4.2.3 Encoding to SAT

We propose to translate the SDMERGE problem to the satisfiability problem of propositional logic (SAT) (cf. Section 2.2). In order to define a propositional encoding for a problem like the SDMERGE problem, first some basic components of the problem that can be represented by Boolean variables have to be identified. Then these components are connected by operators of propositional logic such that they convey the semantic properties of the problem. A positive solution of the resulting formula is expressed by a logical model of the formula, i.e., a mapping of all variables to *true* or *false* (cf. Section 2.2.2). The assigned meaning of the variables evaluated to *true* then describes the solution of the encoded problem.

We first present the set of variables the encoding of the SDMERGE problem is based on. Using these variables we define the semantic properties of well-formedness (cf. Section 3.2, Definition 3.2.5), time consistency (cf. Section 3.2, Definition 3.2.7), and trigger consistency (cf. Section 3.2, Definition 3.3.3), which make up the property of trigger consistency required by a solution of the SDMERGE problem. Then, based on these formulas, we present a set of formulas whose conjunction encodes the SDMERGE problem.

A solution of an instance of the SDMERGE problem is a sequence of messages where each message causes the state machine of its receiver to change to a certain state. This allows us to make statements like "if message N is the first message in the sequence, then at first the state machine M that receives the message is in a state s that has an outgoing transition with N as trigger", and "next, M changes to a state that has an incoming transition with N as trigger and is connected to s with this transition".

The basic components that can be represented by propositional variables in order to express such statements in propositional logic are for example "message N is at position i of the message sequence" or "state machine M is in state s at position i of the message sequence". Therefore, we define the set $\mathcal{V}$ of variables that encode the SDMERGE problem as follows.

Given a set $\mathcal{M} = M_1, \ldots, M_l$ of state machines with $M_i = (S_i, \iota_i, A_i^{tr}, A_i^{\mathit{eff}}, T_i)$ for $i \in [1..l]$, and an event-based sequence diagram $D = (\mathcal{L}, \mathcal{N})$, let $n = |\mathcal{N}|$ and let $\mathcal{S} = \bigcup_{1 \leq i \leq l} S_i$. Then $\mathcal{V}$ is the union of the following sets:

Position	Message Symbol	Variable in va	Source State	Variable in vsrc	Target State	Variable in vtgt
1	o1	$o1_1$	Y1	c_1^{Y1}	Y1	vt_1^{Y1}
2	o2	$o2_2$	X1	c_2^{X1}	X2	vt_2^{X2}
3	b4	$b4_3$	X2	c_3^{X2}	X2	vt_3^{X2}
4	b5	$b5_4$	Y1	c_4^{Y1}	Y1	vt_4^{Y1}
5	a4	$a4_5$	Y1	c_5^{Y1}	Y2	vt_5^{Y2}
6	o3	$o3_6$	X2	c_6^{X2}	X1	vt_6^{X1}
7	b6	$b6_7$	X1	c_7^{X1}	X1	vt_7^{X1}
8	a5	$a5_8$	Y2	c_8^{Y2}	Y1	vt_8^{Y1}

Table 4.2.2: Variables encoding messages of a sequence diagram and source and target states of state machines for each position according to the rightmost merge of Figure 4.2.3.

- va $= \{N_i \mid N \in \mathcal{N} \wedge i \in [1..n]\}$ is a set of variables that encode the placement of each message symbol to a position. If N_i evaluates to true, it means that message N is placed at position i.

- vsrc $= \{c_i^s \mid 1 \leq i \leq n, s \in \mathcal{S})\}$ is a set of variables that encode states that are acting as a source state of a transition being triggered by a message at a certain position. If c_i^s evaluates to true, it means that s is the source state of the transition triggered by the symbol of the message placed on i.

- vtgt $= \{vt_i^s \mid 1 \leq i \leq n, s \in \mathcal{S}\}$ is a set of variables that encode states that are acting as a target state of a transition being triggered by a message at a certain position. If vt_i^s evaluates to true, it means that s is the target state of the transition triggered by the symbol of the message placed on i.

Example 4.2.3. Columns 3, 5, and 7 of Table 4.2.2 show the variables evaluating to *true* for the rightmost merge in Figure 4.2.3. By receiving message o1, state machine Y moves from state Y1 to Y1 on position 1, by receiving message o2, state machine X moves from state X1 to X2 on position 2, and so on. This behavior is determined by the evaluation of the respective variables to *true*. Other variables, for example variable $o1_2$ meaning that message o1 is placed on position 2, evaluate to *false*. ◊

Using the sets of Boolean variables described above we express the properties of an event-based sequence diagram. First, the following formula defines well-formedness (cf. Section 3.2, Definition 3.2.5) of the sequence diagram by requiring a total relation over the messages.

$$\bigwedge_{N\in\mathcal{N}}\left(\bigvee_{i=1}^{n} N_i\right) \wedge \bigwedge_{N\in\mathcal{N}}\bigwedge_{\substack{i,j\in[1..n]\\ i\neq j}}\left(\overline{N}_i \vee \overline{N}_j\right) \qquad \text{(Well-Formedness)}$$

The second formula defines time consistency (cf. Section 3.2, Definition 3.2.7).

$$\bigwedge_{N\in\mathcal{N}}\bigwedge_{i=1}^{n}\left[N_i \rightarrow \bigvee_{\substack{N'\in\mathcal{N},\\ N'\succ N}}\ \bigvee_{i<j\leq n} N'_j\right] \qquad \text{(Time Consistency)}$$

Finally, in order to be trigger-consistent (cf. Section 3.3, Definition 3.3.3) the sequence of messages imposed by their positioning must correspond to a trigger-based path for each of the state machines receiving the messages.

Example 4.2.3 (Cont. from p. 43). The rightmost merge in Figure 4.2.3 with its solution expressed as variables evaluating to *true* in Table 4.2.2 is a trigger-consistent solution because the sequence of messages corresponds to paths in the state machines receiving them. State machine X moves along a path through states X1 → X2 → X2 → X1 → X1. State machine Y moves along a path through states Y1 → Y1 → Y1 → Y2 → Y1. ◊

We split the definition of trigger consistency into three formulas in order to be more intuitive. Other than the previous two formulas, the formulas defining trigger consistency have to take into account not only the positioning of messages, but also the semantics of state machines and their interplay with the message sequence.

The first two formulas towards the definition of trigger consistency concern only state variables, i.e., variables from the sets vsrc and vtgt. The first formula defines that to each position of the message sequence, exactly one source state and exactly one target state must be assigned.

$$\bigwedge_{i=1}^{n}\left[\left(\bigvee_{c_i^s\in\mathsf{vsrc}} c_i^s\right)\wedge\left(\bigvee_{vt_i^s\in\mathsf{vtgt}} vt_i^s\right)\wedge\bigwedge_{s\in\mathcal{S}}\bigwedge_{r\in\mathcal{S}\setminus s}\left((\overline{c}_i^s\vee\overline{c}_i^r)\wedge\left(\overline{vt}_i^s\vee\overline{vt}_i^r\right)\right)\right]$$

(State Machines – State Positions)

The second formula defines the requirement that the sequence of states implied for each state machine by the positions of its states must be connected by transitions. This is a requirement to establish a trigger-based path along the states with respect to their positions. This means that the target state of a state machine at some position i must be the same as the next source state of the same state machine at a position j with $j > i$. States of other state machines can be placed at any position k with $i < k < j$.

$$\bigwedge_{i=1}^{n}\bigwedge_{M\in\mathcal{M}}\bigwedge_{s\in\pi_1(M)}\left[\left(vt_i^s\rightarrow\bigwedge_{r\in\pi_1(M)\setminus s}\overline{c}_{i+1}^r\right)\wedge\left(\bigwedge_{j=1}^{i}\left(vt_i^s\wedge\bigwedge_{l=1}^{j}\overline{c}_l^s\rightarrow\bigwedge_{r\in\pi_1(M)\setminus s}\overline{c}_{j+1}^r\right)\right)\right]$$

(State Machines – Paths)

Example 4.2.3 (Cont. from p. 43). The sequence of message symbols in the second column of Table 4.2.2 implies the sequence $[\mathsf{Y1}, \mathsf{Y1}, \mathsf{Y1}, \mathsf{Y2}, \mathsf{Y1}]$ of states for state machine Y, and the sequence $[\mathsf{X1}, \mathsf{X2}, \mathsf{X2}, \mathsf{X1}, \mathsf{X1}]$ of states for state machine X. These sequences are determined for each state machine by the first line containing one of its states in the column "Source State" followed by its states in the following lines in the column "Target State". ◊

The third formula defines the essence of trigger consistency. It ensures that the source and target states at each position are actually connected by a transition that carries as trigger symbol the message symbol placed at the same position. This formula uses the function trans returning all transitions of the state machine instantiated by the lifeline receiving a message N that carry as trigger the same symbol as the message, i.e., $\mathsf{trans}(N) = \{t \mid t \in \pi_4(\pi_2(\mathsf{rcv}(N)))$ and $\pi_2(N) = \pi_2(t)\}$.

$$\bigwedge_{N \in \mathcal{N}} \bigwedge_{i=1}^{n} \left[N_i \rightarrow \bigvee_{t \in \mathsf{trans}(N)} \left(c_i^{\pi_1(t)} \wedge t_i^{\pi_4(t)} \right) \right] \qquad \text{(Trigger Consistency)}$$

The encoding of the SDMERGE problem is based on the above definitions of the properties of a trigger-consistent model. Recall that the SDMERGE problem asks whether for a set of state machines, a sequence diagram, and two revisions of the sequence diagram, a consolidated version exists (cf. Definition 4.2.4). This consolidated version contains all messages of the original version and all added messages of its revisions, respects relative order of the messages, and is trigger-consistent. Hence the consolidated version must fulfill all the properties expressed in the above formulas and additionally take into account the added messages and their possible positions defined by the allow function (cf. Definition 4.2.3).

Given an instance of the SDMERGE problem consisting of the sequence diagrams $D_x = (\mathcal{L}_x, \mathcal{N}_x)$ with $x \in \{o, \alpha, \beta\}$, let $\mathcal{N} = \mathcal{N}_\alpha \cup \mathcal{N}_\beta$ be the set of all messages (note that the messages of D_o are already contained in both D_α and D_β as no deletions are allowed). The full encoding of the SDMERGE problem is expressed by a conjunction of the following formulas (4.2.1) to (4.2.4).

The first formula is based on the formula (Well-Formedness). The set $\mathcal{N}$ over which the conjunction iterates contains all messages that are to be contained in the consolidated version, i.e., the messages of the original sequence diagram and those of the two revisions. The positions of the messages that have to be considered are restricted to those precalculated by the allow function. Therefore, other than the formula (Well-Formedness), which iterates over all positions, formula (4.2.1) iterates only over those that are returned by the allow function. The formula encodes that each message must be placed on exactly one of the positions returned by its allow function and no messages can be placed on the same position, which ensures well-formedness of the merged sequence diagram.

$$\bigwedge_{N \in \mathcal{N}} \left(\bigvee_{i \in \mathsf{allow}(N)} N_i \right) \wedge \bigwedge_{\substack{i,j \in \mathsf{allow}(N) \\ i \neq j}} \left(\overline{N}_i \vee \overline{N}_j \right) \qquad (4.2.1)$$

The next formula is based on the above formula (Time Consistency). Similarly to formula (4.2.1), it also differs in that it restricts the positioning of messages to those positions that are returned by the allow function.

$$\bigwedge_{N\in\mathcal{N}} \bigwedge_{i\in\mathsf{allow}(N)} \left[N_i \rightarrow \bigvee_{\substack{N'\in\mathcal{N},\\ N'\succ N}} \bigvee_{\substack{j>i,\\ j\in\mathsf{allow}(N')}} N'_j \right] \tag{4.2.2}$$

Finally, among the three formulas concerning trigger consistency, only the third formula has to consider the allow function since the first two encode only properties of the state machines. Hence formula (4.2.3) is the same as formula (State Machines – State Positions), formula (4.2.4) is the same as formula (State Machines – Paths), and formula (4.2.5) restricts the range of positions of formula (Trigger Consistency) by those returned by the allow function. Formula (4.2.4) uses the function trans returning all transitions of the state machine instantiated by the lifeline that receives a message N and carries as trigger the same symbol as the message, i.e., $\mathsf{trans}(N) = \{t \mid t \in \pi_4(\pi_2(\mathsf{rcv}(N)))$ and $\pi_2(N) = \pi_2(t)\}$.

$$\bigwedge_{i=1}^{n} \left[\left(\bigvee_{c_i^s\in\mathsf{vsrc}} c_i^s \right) \wedge \left(\bigvee_{vt_i^s\in\mathsf{vtgt}} vt_i^s \right) \wedge \bigwedge_{s\in\mathcal{S}} \bigwedge_{r\in\mathcal{S}\setminus s} \left(\left(\overline{c}_i^s \vee \overline{c}_i^r \right) \wedge \left(\overline{vt}_i^s \vee \overline{vt}_i^r \right) \right) \right] \tag{4.2.3}$$

$$\bigwedge_{i=1}^{n-1} \bigwedge_{M\in\mathcal{M}} \bigwedge_{s\in\pi_1(M)} \left[\left(vt_i^s \rightarrow \bigwedge_{r\in\pi_1(M)\setminus s} \overline{c}_{i+1}^r \right) \wedge \bigwedge_{j=1}^{i} \left(\left(vt_i^s \wedge \bigwedge_{l=1}^{j} \overline{c}_l^s \right) \rightarrow \left(\bigwedge_{r\in\pi_1(M)\setminus s} \overline{c}_{j+1}^r \right) \right) \right] \tag{4.2.4}$$

$$\bigwedge_{N\in\mathcal{N}} \bigwedge_{i\in\mathsf{allow}(N)} \left[N_i \rightarrow \bigvee_{t\in\mathsf{trans}(N)} \left(c_i^{\pi_1(t)} \wedge t_i^{\pi_4(t)} \right) \right] \tag{4.2.5}$$

Given an instance $\mathcal{M}$ of the SDMERGE problem, its encoding as a conjunction of the formulas (4.2.1) to (4.2.4) is of size polynomial with respect to the size of $\mathcal{M}$. This is the case because any of the formulas (4.2.1) to (4.2.4) contains at most four nested iterations over sets contained in the input.

4.2.4 Computational Complexity

We show that the SDMERGE problem is in the complexity class P. To this end, we show how the problem can be solved with an algorithm based on dynamic programming [8] running in polynomial time with respect to the size of the instance. The design of this algorithm is based on the following two observations regarding event-based time-consistent sequence diagrams (cf. Definitions 3.2.4 and 3.2.7). Recall that time consistency imposes a total order over the set of

messages in the sequence diagram, making this set a sequence. To enhance the presentation, in the remainder of this section we denote a message only by its message symbol when the context is clear.

Observation 4.2.1 (Revision Fragments)**.** Two revisions $D_\alpha = (\mathcal{L}_\alpha, \mathcal{N}_\alpha)$ and $D_\beta = (\mathcal{L}_\beta, \mathcal{N}_\beta)$ of an event-based time-consistent sequence diagram $D_o = (\mathcal{L}_o, \mathcal{N}_o)$ can be divided into fragments as follows. Let o_i and o_j be two messages of D_o at positions i and j such that $\mathsf{pos}_o(o_j) = \mathsf{pos}_o(o_i) - 1$. Then $F_x^{(i,j)} = [N \in \mathcal{N}_x \mid \mathsf{pos}_x(o_i) < \mathsf{pos}_x(N) < \mathsf{pos}_x(o_j)]$ for $x \in \{\alpha, \beta\}$ are the sequences of messages inserted in the revisions D_α and D_β respectively, between o_i and o_j. Further, let o_1 be the first message and o_e be the last message of D_o, and the sequences $F_x^{(\bullet,1)} = [N \in \mathcal{N}_x \mid \mathsf{pos}_x(N) < \mathsf{pos}_x(o_1)]$ and $F_x^{(e,\bullet)} = [N \in \mathcal{N}_x \mid \mathsf{pos}_x(N) > \mathsf{pos}_x(o_e)]$ for $x \in \{\alpha, \beta\}$ be the fragments containing the sequences added at the beginning and at the end of D_o respectively. According to the allow function the consolidated version contains between the messages o_i and o_j, before message o_1, and after message o_e a sequence of messages that contains only the messages of fragments $F_\alpha^{(i,j)}$ and $F_\beta^{(i,j)}$, $F_\alpha^{(\bullet,1)}$ and $F_\beta^{(\bullet,1)}$, and $F_\alpha^{(e,\bullet)}$ and $F_\beta^{(e,\bullet)}$ respectively, and maintains their relative order.

Example 4.2.2 (Cont. from p. 40). The problem instance depicted in Figure 4.2.3 contains the fragments $F_\alpha^{(2,3)} = [\mathsf{a4}]$, $F_\beta^{(2,3)} = [\mathsf{b4}, \mathsf{b5}]$, $F_\alpha^{(e,\bullet)} = [\mathsf{a5}]$, $F_\beta^{(e,\bullet)} = [\mathsf{b6}]$. ◊

Observation 4.2.2 (Independence of Lifelines)**.** Trigger consistency (other than full consistency) depends only on sequences of symbols that are received by a lifeline and that occur as triggers in the respective state machine, thereby fully ignoring the sent symbols occurring as effects in the respective state machine. Further, for a sequence diagram $D = (\mathcal{L}, \mathcal{N})$ to be trigger-consistent, each of its lifelines $L \in \mathcal{L}$ must be trigger-consistent, *but the trigger consistency of one lifeline does not depend on the trigger consistency of any other lifeline* (cf. Definition 3.3.3). Therefore, given a sequence diagram $D_o = (\mathcal{L}_o, \mathcal{N}_o)$ and two revisions $D_\alpha = (\mathcal{L}_\alpha, \mathcal{N}_\alpha)$ and $D_\beta = (\mathcal{L}_\beta, \mathcal{N}_\beta)$ of D_o, in order to merge two fragments $F_\alpha^{(x,y)}$ and $F_\beta^{(x,y)}$ for $(x, y) \in \{(\bullet, 1), (e, \bullet)\} \cup \{(i, j) \mid 1 \leq i, j \leq |\mathcal{N}_o|, j = i - 1\}$ we have to merge for each lifeline $L \in \mathcal{L}_\alpha \cup \mathcal{L}_\beta$ each $F_{\alpha,L}^{(x,y)}$ with $F_{\beta,L}^{(x,y)}$. Each of these sequences contains only messages that are received by the same lifeline L and are contained in the fragment (x, y) of D_α or D_β. Based on these sequences, we then have to determine whether any sequence $F_{\gamma,L}^{(i,j)}$ merging $F_{\alpha,L}^{(i,j)}$ and $F_{\beta,L}^{(i,j)}$ is trigger-consistent with the state machine instantiated by L. If this is the case, then any sequence $F_\gamma^{(i,j)}$ that contains all messages of $F_{\gamma,L}^{(i,j)}$ for all $L \in \mathcal{L}$ and maintains their relative order is trigger-consistent.

Example 4.2.2 (Cont. from p. 40). In Figure 4.2.3 consider the fragments $F_\alpha^{(2,3)} = [\mathsf{a4}]$ and $F_\beta^{(2,3)} = [\mathsf{b4}, \mathsf{b5}]$ of revisions D_α and D_β respectively. The sequences $F_{\alpha,y}^{(2,3)} = [\mathsf{a4}]$, $F_{\beta,x}^{(2,3)} = [\mathsf{b4}]$, and $F_{\beta,y}^{(2,3)} = [\mathsf{b5}]$ separate these fragments by their receiver lifelines x and y. A trigger-consistent merge of $F_\alpha^{(2,3)}$ and $F_\beta^{(2,3)}$ with respect to the state machines in Figure 4.2.4 consists of a trigger-consistent merge of $F_{\alpha,y}^{(2,3)}$ and $F_{\beta,y}^{(2,3)}$ with the message of $F_{\beta,x}^{(2,3)}$ added before, after, or at an arbitrary position in between the merge. ◊

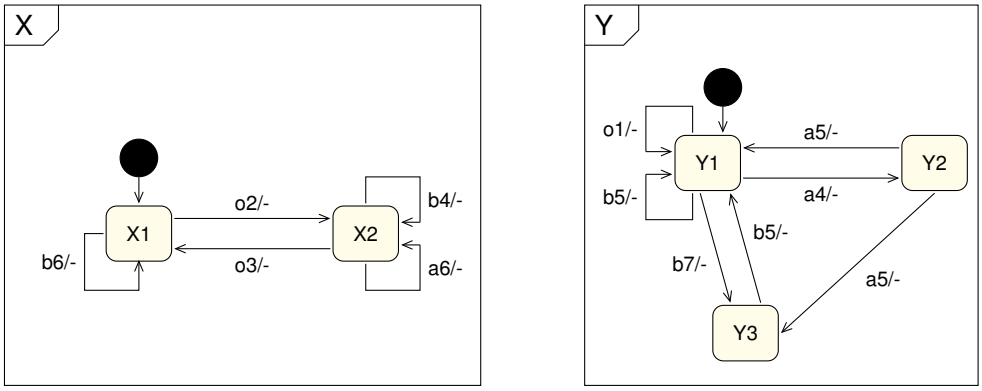

Figure 4.2.5: The state machines modeling the behavior of the lifelines in Figure 4.2.6.

From Observations 4.2.1 and 4.2.2 it follows that the difficulty of solving this problem lies in merging for each lifeline $L \in \mathcal{L}_\alpha \cup \mathcal{L}_\beta$ and for each $(x, y) \in \{(\bullet, 1), (e, \bullet)\} \cup \{(i, j) \mid 1 \leq i, j \leq |\mathcal{N}_o|, j = i - 1\}$ the fragments $F^{(x,y)}_{\alpha,L}$ and $F^{(x,y)}_{\beta,L}$ of message sequences. Computing the fragments as discussed in Observation 4.2.1 is trivial, as is picking an arbitrary interleaving of the merged sequences $F^{(x,y)}_{\gamma,L}$ for each L as discussed in Observation 4.2.2.

To merge two message sequences P and Q where all messages are received by the same lifeline into a sequence that is trigger-consistent with the state machine instantiated by the lifeline, we propose an approach based on dynamic programming [8]. Dynamic programming is a well-known method for solving such problems that can be broken down into overlapping subproblems. Dynamic programming algorithms for problems like the longest common subsequence problem are discussed in standard literature on algorithms [36].

In our case, the problem is to find an interleaving between P and Q such that the result represents a path of triggers in a state machine. This problem contains the subproblems of merging any of the subsequences of P that start with the first message of P to any of the subsequences of Q that start with the first message of Q. To compute such a trigger-consistent subsequence, we only need the following information:

- The indices of the last messages of subsequences of P and Q such that these subsequences can be trigger-consistently merged, and
- the set of transitions reached by a path consisting of some interleaving of these two subsequences.

The result of each trigger-consistent merge of two subsequences can then be used to compute the following subsequence containing one more message from either of the sequences P or Q.

Example 4.2.4. Consider the sequence diagram D_o and its two revisions D_α and D_β in Figure 4.2.6 and the two state machines in Figure 4.2.5, which the lifelines of the sequence diagrams instantiate. The revisions contain four lifeline-related fragments; $F^{(e,\bullet)}_{\alpha,x} = [\mathsf{a6}]$ and $F^{(e,\bullet)}_{\beta,x} = [\mathsf{b4}]$ are received by lifeline x, and $F^{(e,\bullet)}_{\alpha,y} = [\mathsf{a4}, \mathsf{a5}]$ and $F^{(e,\bullet)}_{\beta,y} = [\mathsf{b5}, \mathsf{b7}, \mathsf{b5}]$ are received by life-

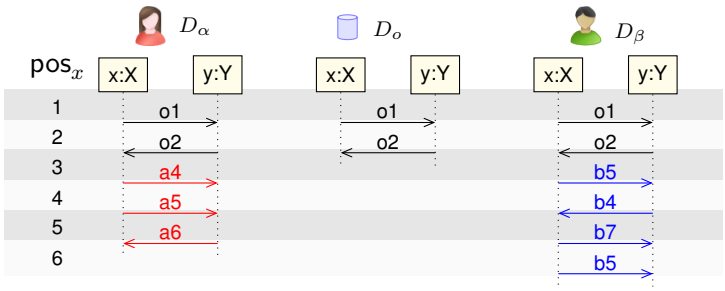

Figure 4.2.6: A sequence diagram and two of its revisions trigger-consistent with respect to the state machines depicted in Figure 4.2.5.

line y. Now consider the two fragments $F^{(e,\bullet)}_{\alpha,y}$ and $F^{(e,\bullet)}_{\beta,y}$ containing messages received by lifeline y. The problem of trigger-consistently merging these two sequences has the overlapping subproblems of trigger-consistently merging the subsequences [a4] and [b5], [a4] and [b5, b7], [a4, a5] and [b5], [a4, a5] and [b5, b7], etc. Computing the set of transitions reached by interleaving the two subsequences [a4] and [b5, b7] can be based on the set of transitions stored for interleaving the subsequences [a4] and [b5]. Similarly, computing the set of transitions for the two subsequences [a4, a5] and [b5] can also be based on the set of transitions stored for [a4] and [b5]. Then, computing the set of transitions reached by interleaving the subsequences [a4, a5] and [b5, b7] can be based on the set of transitions stored for interleaving the subsequences [a4, a5] and [b5] and the subsequences [a4] and [b5, b7], etc.

◊

The following recurrence relation reach shows that given the sequences P and Q and a state machine $M = (S, \iota, A^{tr}, A^{\mathit{eff}}, T)$ the set of transitions reached for each subproblem can be computed in time $O(|P| \cdot |Q| \cdot |T|)$.

Definition 4.2.6 (Reachable Transitions)**.** Given a state machine $M = (S, \iota, A^{tr}, A^{\mathit{eff}}, T)$, the function succ : $\mathcal{P}(T) \to \mathcal{P}(T)$ maps a set $T' \subseteq T$ of transitions to the set of transitions succeeding T', i.e., $\mathsf{succ}(T') = \{(s', a, s'') \in T \mid \exists (s, b, s') \in T'\}$. Further, given two sequences P and Q of symbols contained in A^{tr}, the function reach : $\{1..|P|\} \times \{1..|Q|\} \to \mathcal{P}(T)$ returns the set of transitions that are reached by a sequence of triggers containing the first i symbols of sequence P and the first j symbols of sequence Q for $i \in [1..|P|]$ and $j \in [1..|Q|]$ as follows.

$$
\begin{aligned}
\mathsf{reach}(0,0) = \;& T', \\
\mathsf{reach}(i,0) = \;& \{(s,a,s') \in \mathsf{succ}(\mathsf{reach}(i-1,0)) \mid a = P[i]\}, \\
\mathsf{reach}(0,j) = \;& \{(s,a,s') \in \mathsf{succ}(\mathsf{reach}(0,j-1)) \mid a = Q[j]\}, \\
\mathsf{reach}(i,j) = \;& \{(s,a,s') \in \mathsf{succ}(\mathsf{reach}(i-1,j)) \mid a = P[i]\} \;\cup \\
& \{(s,a,s') \in \mathsf{succ}(\mathsf{reach}(i,j-1)) \mid a = Q[j]\}.
\end{aligned}
$$

△

(i,j)	$(P[i],P[j])$	$\mathsf{reach}(i,j)$
$(0,0)$	$(-,-)$	$\{(\mathsf{Y1},\mathsf{o1},\mathsf{Y1})\}$
$(0,1)$	$(-,\mathsf{b5})$	$\{(\mathsf{Y1},\mathsf{b5},\mathsf{Y1})\}$
$(1,0)$	$(\mathsf{a4},-)$	$\{(\mathsf{Y1},\mathsf{a4},\mathsf{Y2})\}$
$(1,1)$	$(\mathsf{a4},\mathsf{b5})$	$\{(\mathsf{Y1},\mathsf{a4},\mathsf{Y2})\} \cup \emptyset$
$(0,2)$	$(-,\mathsf{b7})$	$\{(\mathsf{Y1},\mathsf{b7},\mathsf{Y3})\}$
$(2,0)$	$(\mathsf{a5},-)$	$\{(\mathsf{Y2},\mathsf{a5},\mathsf{Y1}),(\mathsf{Y2},\mathsf{a5},\mathsf{Y3})\}$
$(1,2)$	$(\mathsf{a4},\mathsf{b7})$	$\emptyset \cup \emptyset$
$(2,1)$	$(\mathsf{a5},\mathsf{b5})$	$\{(\mathsf{Y2},\mathsf{a5},\mathsf{Y1}),(\mathsf{Y2},\mathsf{a5},\mathsf{Y3})\} \cup \{(\mathsf{Y1},\mathsf{b5},\mathsf{Y1}),(\mathsf{Y3},\mathsf{b5},\mathsf{Y1})\}$
$(2,2)$	$(\mathsf{a5},\mathsf{b7})$	$\emptyset \cup \{(\mathsf{Y1},\mathsf{b7},\mathsf{Y3})\}$
$(0,3)$	$(-,\mathsf{b5})$	$\{(\mathsf{Y3},\mathsf{b5},\mathsf{Y1})\}$
$(1,3)$	$(\mathsf{a4},\mathsf{b5})$	$\{(\mathsf{Y1},\mathsf{a4},\mathsf{Y2})\} \cup \emptyset$
$(2,3)$	$(\mathsf{a5},\mathsf{b5})$	$\{(\mathsf{Y2},\mathsf{a5},\mathsf{Y1}),(\mathsf{Y2},\mathsf{a5},\mathsf{Y3})\} \cup \{(\mathsf{Y3},\mathsf{b5},\mathsf{Y1})\}$

Table 4.2.3: The computation of the reach function for the instance of the SDMERGE problem depicted in Figures 4.2.5 and 4.2.6

When applied to an instance of the SDMERGE problem, the set T' of transitions in the above definition contains the set of transitions reached by applying the previous sequence of messages if one exists, and otherwise $T' = T$.

Example 4.2.5. Table 4.2.3 shows the sets of transitions reached by subsequences of the two fragments $F^{(e,\bullet)}_{\alpha,y} = [\mathsf{a4},\mathsf{a5}]$ and $F^{(e,\bullet)}_{\beta,y} = [\mathsf{b5},\mathsf{b7},\mathsf{b5}]$ containing the messages received by lifeline y of the fragments in diagrams D_α and D_α of Figure 4.2.6. The first line refers to the set of transitions reached without applying any of the messages in $F^{(e,\bullet)}_{\alpha,y}$ or $F^{(e,\bullet)}_{\beta,y}$. This is the set of transitions reached by applying the previous sequence of messages received by lifeline y. The following lines show the sets of transitions reached by applying the first i messages of sequence P and the first j messages of sequence Q. Since the last line does not contain the empty set, there exists a trigger-consistent interleaving of P and Q. Such an interleaving can be found by picking some transition, for example $(\mathsf{Y3},\mathsf{b5},\mathsf{Y1})$, in the last row of the table, checking its predecessors $(1,3)$ and $(2,2)$ for transitions with target state Y3, picking $(\mathsf{Y1},\mathsf{b7},\mathsf{Y3})$ from $(2,2)$, then picking $(\mathsf{Y2},\mathsf{a5},\mathsf{Y1})$ from $(2,1)$, then picking $(\mathsf{Y1},\mathsf{a4},\mathsf{Y2})$ from $(1,1)$, and finally picking $(\mathsf{Y1},\mathsf{b5},\mathsf{Y1})$ from $(0,1)$, resulting in the sequence $[\mathsf{b5},\mathsf{a4},\mathsf{a5},\mathsf{b7},\mathsf{b5}]$. Depending on which transitions are picked, other solutions include the sequences $[\mathsf{a4},\mathsf{a5},\mathsf{b5},\mathsf{b7},\mathsf{b5}]$ and $[\mathsf{b5},\mathsf{b7},\mathsf{b5},\mathsf{a4},\mathsf{a5}]$. ◊

If the fragments $F^{(i,j)}_{\alpha,L}$ and $F^{(i,j)}_{\beta,L}$ for a lifeline L are followed in sequence diagrams D_α and D_β by a message sequence $F_{o,L}$ containing a sequence of messages received by L in the original sequence diagram, then from at least one target state reached by a transition in $\mathsf{reach}(|F^{(i,j)}_{\alpha,L}|,|F^{(i,j)}_{\beta,L}|)$ there must exist a path corresponding to the sequence $F_{o,L}$ for the merge to be trigger-consistent. The set T' reached by paths from these target states is then used to compute the subsequent merge from fragments $F^{(i',j')}_{\alpha,L}$ and $F^{(i',j')}_{\beta,L}$ for $i' > i$ if such fragments exist.

Corollary 4.2.1. The SDMERGE problem is in P.

We have shown that some solution of the SDMERGE problem can be found in polynomial time, however, it should be noted that the number of solutions can be exponential, as explained by the following observation.

Observation 4.2.3 (Number of Solutions)**.** Let $D_\alpha = (\mathcal{L}_\alpha, \mathcal{N}_\alpha)$ and $D_\beta = (\mathcal{L}_\beta, \mathcal{N}_\beta)$ be two revisions of the sequence diagram $D_o = (\mathcal{L}_o, \mathcal{N}_o)$, and let F_α and F_β be two fragments in D_α and D_β. Then the number of time-consistent merges of F_α and F_β is $\frac{(|F_\alpha|+|F_\beta|)!}{|F_\alpha|!|F_\beta|!}$. For the total number of merges we have to consider all fragments $F_\alpha^{(x,y)}$ and $F_\beta^{(x,y)}$ for $(x, y) \in \{(\bullet, 1), (e, \bullet)\} \cup \{(i, j) \mid 1 \leq i, j \leq |\mathcal{N}_o|, j = i - 1\}$ inserted between two messages of the original sequence diagram and build the product $\prod_{(x,y)\in\{(\bullet,1),(e,\bullet)\}\cup\{(i,j)|1\leq i,j\leq|\mathcal{N}_o|,j=i-1\}} \frac{(|F_\alpha^{(x,y)}|+|F_\beta^{(x,y)}|)!}{|F_\alpha^{(x,y)}|!|F_\beta^{(x,y)}|!}$. The number of time-consistent merges is the maximal number of trigger-consistent merges, which can be reached if any trigger can be consumed from any state.

4.3 Reachability – the k-SMREACH Problem

This section presents the bounded formulation k-SMREACH of the *State Machine Reachability Problem*. Other than the SDMERGE problem discussed in the previous section, this problem relies on the definition of k-consistency (cf. Section 3.3, Definitions 3.3.13 and 3.3.12), which, in contrast to trigger consistency, also consider sending of messages in sequence diagrams and effects on transitions in state machine. This problem asks whether in a set of state machines a *partial global state* is reachable from some global state (cf. Definition 3.3.5). on a path with a size of at most k transactions. In practical applications this global state is likely to be represented by the set of states containing the initial state of each state machine. However, since for the problem definition and solving it does not make a difference where to start, we use a generalized formulation. The partial global state can be seen as a partially specified global state; it contains at most one state of each state machine.

For positive instances of this problem, that is, for which such a sequence of messages exists, a sequence diagram can be generated which is k-consistent with the state machines for $k = 1$ (one empty message may be necessary to reach an intermediate state before the first message of the sequence diagram can be sent).

In this section we refer to state machines instead of instances of state machines where the context is clear. It does not make a difference for the problem formulation, because two instances of the same state machine can be represented by two state machines implementing the same behavior.

We first motivate the problem by presenting an intuitive example on the interaction between a PhD student, a coffee machine, and a payment unit. We then give a formal problem definition followed by an encoding of the problem to SAT. The encoding expresses the semantics of the property of k-consistency and therefore forms a part of the semantics of the *tMVML*. Finally, we show that the SMREACH problem is NP-complete.

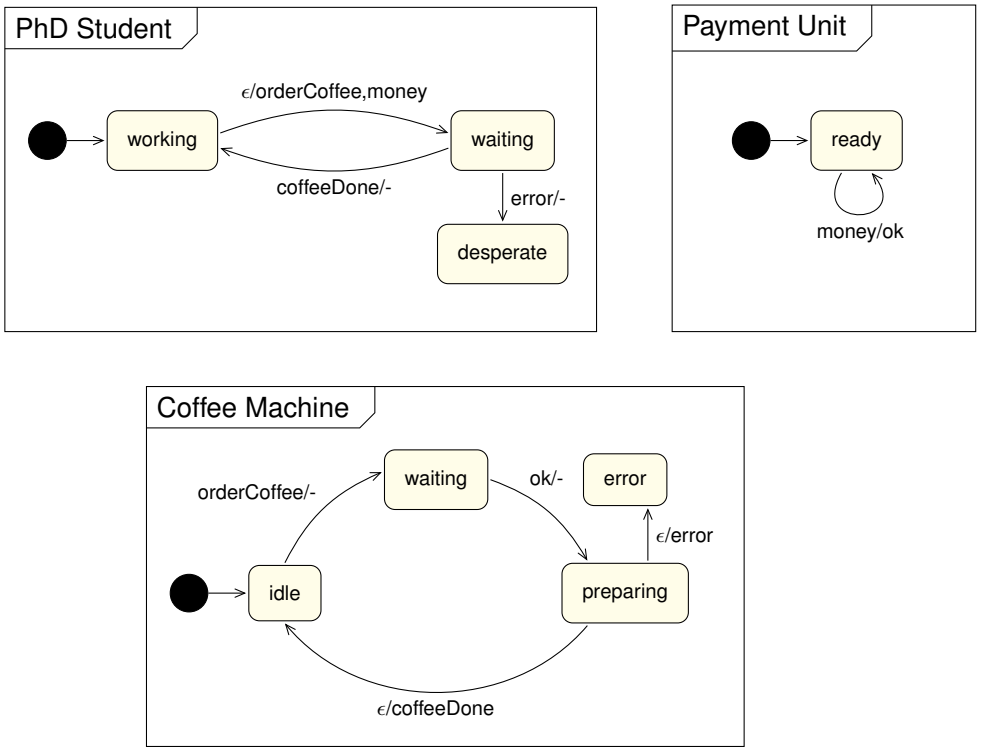

Figure 4.3.1: State machines of a student, a coffee machine, and a payment unit.

4.3.1 A Motivating Example

Figure 4.3.1 shows the state machines of a student and a coffee machine implementing a workflow of a student's interaction with a coffee machine and the payment unit of the coffee machine. Recall that each transition carries a label consisting of a symbol called *trigger* to the left, and a set of symbols called *effects* to the right of the symbol "/". The empty trigger ϵ indicates that the transition can be executed without receiving any trigger symbol. The symbol "–" represents the empty set of effects. The receipt of the trigger symbol causes the state machine to attempt an execution of the transition changing its current state from the source state to the target state of the transition. The symbols in the set of effects are sent during the execution of the transition. The execution of a transition is only finished if each of its effects is received as a trigger by a different state machine in its current state.

A possible instance of the k-SMREACH problem are the state machines depicted in Figure 4.3.1 and the question whether a partial global state containing the states desperate from the state machine PhD Student and error from the state machine Coffee Machine is reachable from the global state containing the states working from PhD Student, idle from Coffee Machine, and ready from Payment Unit by a path of length at most 5. In this case, the answer is positive. The sequence diagram representing such a path by its message sequence is depicted in Figure 4.3.2.

On the other hand, a partial global state containing the states waiting from PhD Student and

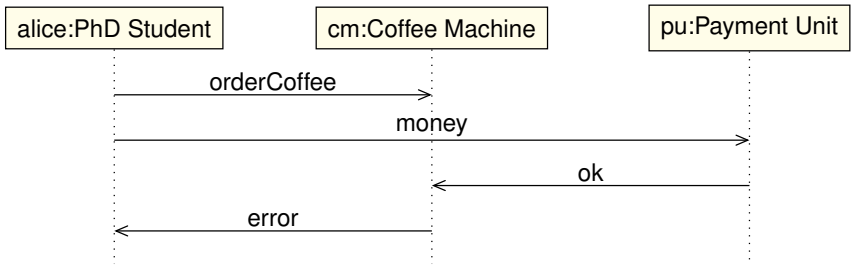

Figure 4.3.2: Sequence diagram depicting a sequence leading the two state machines of Figure 4.3.1 to the partial global state $(\mathsf{desperate}, \mathsf{error})$.

error from Coffee Machine is not reachable from the global state $(\mathsf{working}, \mathsf{idle}, \mathsf{ready})$ by a path of length at most 5.

4.3.2 Problem Definition

The k-SMREACH problem deals with the existence of a path to a set of states in state machines of a state machine view. It is based on the Definitions 3.3.4 to 3.3.12 regarding k-consistency in Section 3.3. The set of states to be reached contains at most one state of each state machine. Hence it does not necessarily specify a complete global state as defined in Definition 3.3.5. We therefore define a partial global state as a tuple containing for each state machine either one of its states or the fresh symbol "ς " as follows.

Definition 4.3.1 (Partial Global State)**.** Given a set $\mathcal{M} = \{M_1, \ldots, M_l\}$ of (extended) state machines with $M_i = (S_i, \iota_i, A_i^{tr}, A_i^{\mathit{eff}}, T_i)$ for $i \in [1..l]$, a *partial global state* is an l-tuple $\hat{s}_p \in S_1 \cup \{\varsigma\} \times S_2 \cup \{\varsigma\} \times \cdots \times S_l \cup \{\varsigma\}$, where ς is a new symbol not contained in any S_i. $\triangle$

An extended state machine as defined in Definition 3.3.4 of Section 3.3 helps to distinguish between the event of having received the trigger and the event of being able to send the effect, and turned out to be very convenient when finding solving methods for consistency problems with state machines. Other than a non-extended state machine, an extended state machine can contain transitions which have ϵ as trigger and the empty set as effects (cf. Definitions 3.2.1 and 3.3.4 in Sections 3.2 and 3.3). Such transitions connect intermediate states to original states when the corresponding transition of the non-extended state machine has the empty set 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. The environment is necessary to translate a solving method based on an extended state machine back and forth to the corresponding non-extended state machine.

Definition 4.3.2 (Environment)**.** Given a state machine $M = (S, \iota, A^{tr}, A^{\mathit{eff}}, T)$, its extended state machine $M^* = (S \cup S^*, \iota, A^{tr}, A^{\mathit{eff}}, T^*)$, and a state $s \in S$, the *environment* of s is given by the function $\mathsf{env} : S \to \mathcal{P}(S^*)$ such that $\mathsf{env}(s) = \{s^* \mid (s^*, \epsilon, \emptyset, s) \in T^*\}$. $\triangle$

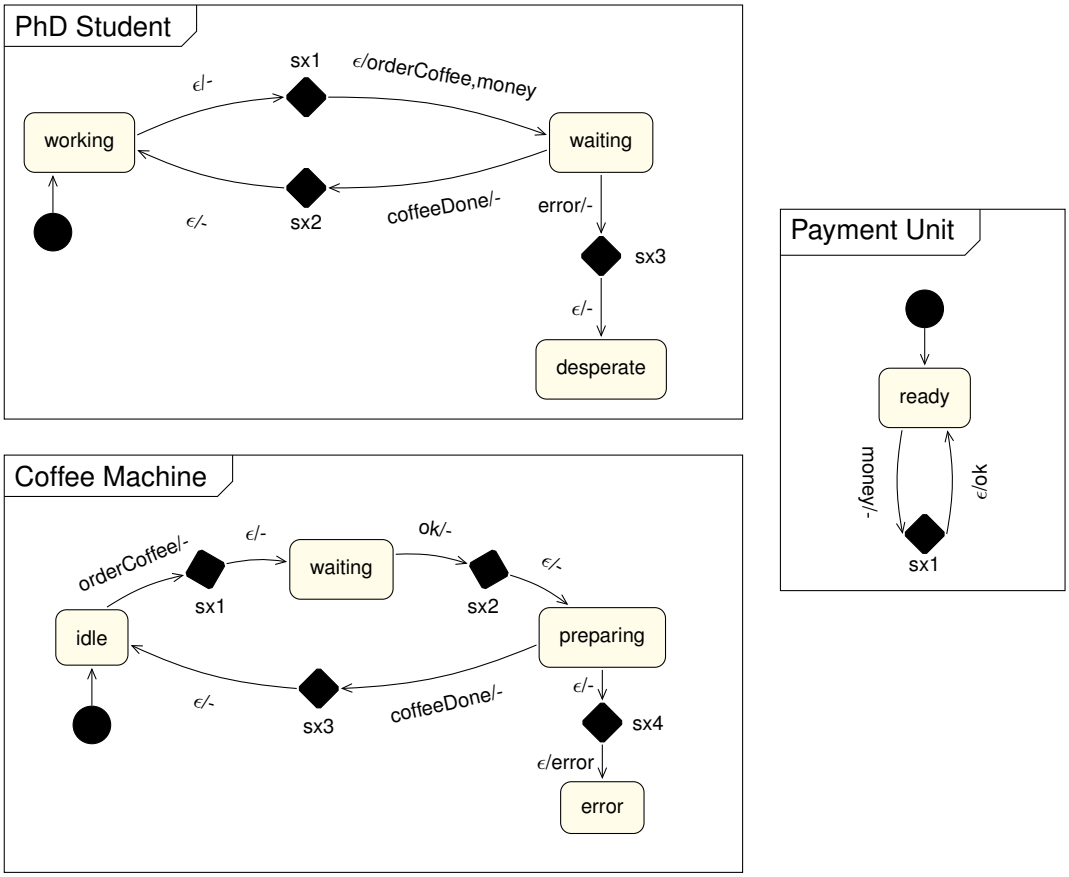

Figure 4.3.3: Extended state machines corresponding to state machines in Figure 4.3.1.

Example 4.3.1. Figure 4.3.3 depicts the extended state machines for the state machines shown in Figure 4.3.1. The extended states are labeled. The environment of state working in state machine PhD Student is $\{\mathsf{sx1}, \mathsf{sx2}\}$, i.e., the two intermediate states on its incoming transitions that contain the empty set as effect. ◊

A partial global state subsumes one or more global states. Such global states contain each state of the partial global state. We refer to this relation as *matching*. If the partial global state refers to a non-extended state machine and the global state to an extended state machine, then the environment (cf. Definition 4.3.2) has to be taken into account.

Definition 4.3.3 (Matching)**.** Let $\mathcal{M}^* = \{M_1^*, \ldots, M_l^*\}$ be a set of extended state machines with $M_i^* = (S_i \cup S^*, \iota_i, A_i^{tr}, A_i^{\mathit{eff}}, T_i)$ for $i \in [1..l]$, let $\hat{s} = (s_1, \ldots, s_l)$ be a global state with $s_i \in S_i \cup S_i^*$ for $i \in [1..l]$, and let $\hat{s}_p = (p_1, \ldots, p_l)$ be a partial state with $p_i \in S_i$ for $i \in [1..l]$. Then $\hat{s}$ *matches* $\hat{s}_p$ if for all $i \in [1..l]$ with $p_i \neq \varsigma$ it holds that if $s_i \in S$ then $s_i = p_i$ and if $s_i \in S^*$ then $s_i \in \mathsf{env}(p_i)$. △

Finally, based on the definitions of a path (cf. Section 3.3, Definition 3.3.10) connecting global states and that of reachability (cf. Section 3.3, Definition 3.3.11) of a global state, we define the k-SMREACH problem as follows.

Definition 4.3.4 (k-SMREACH Problem)**.**

Instance: A set $\mathcal{M} = \{M_1, \ldots, M_l\}$ of state machines with $M_i = (S_i, \iota_i, A_i^{tr}, A_i^{eff}, T_i)$ for $i \in [1..l]$ where for all $i, j \in [1..l]$ with $i \neq j$ it holds that S_i, S_j are disjoint and T_i, T_j are disjoint, a global state $\hat{s}$ over $\mathcal{M}$, a partial global state $\hat{s}_p$ over $\mathcal{M}$, and a positive integer $k = \mathsf{p}(l)$ for some polynomial 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 partial global state $\hat{s}_p$ as *goal*. We define the parameter k as a polynomial in the number of state machines for complexity theoretical reasons. The encoding presented below would be of exponential size with respect to the input if k is exponential in the number of state machines.

4.3.3 Encoding to SAT

In order to find solutions to the k-SMREACH problem, we propose to encode it to the satisfiability problem of propositional logic (SAT). To this end, we build a propositional formula representing an instance of the k-SMREACH problem and hand the formula to a SAT solver. The solver returns `SAT` and a logical model if a global state subsumed by the partial global state can be reached in at most k steps. Otherwise, it returns `UNSAT`. The logical model can be translated back into a path leading to such a global state.

Similarly to Section 4.3 describing the SDMERGE problem, we first describe the set of variables used to encode the k-SMREACH problem along with their meaning, and then present a set of formulas that make up the encoding of the problem.

A solution to an instance of the k-SMREACH problem consists of a sequence of transactions with length at most k such that applying these transactions to the state machines starting in the given global state reaches a global state that matches the given partial state. Therefore, similarly to the previous problem SDMERGE in Section 4.3, we can base the encoding on variables representing relevant components like messages, states, or transitions, of the diagrams being placed at specific positions with respect to the message sequence.

However, other than for the SDMERGE problem, where only trigger consistency was considered, we are now dealing with k-consistency, which also takes into account the sending of messages and effects of transitions. This property has been defined in Section 3.3 based on extended state machines rather than state machines (cf. Definitions 3.3.4 and 3.3.12). Recall that any state machine can be converted to an extended state machine by adding an extra state onto each transition and separating the trigger symbol from the effect symbols. The encoding of the k-SMREACH problem is based on extended state machines.

The set $\mathcal{V}$ of variables in the formula encoding the k-SMREACH problem is based on the set of all symbols, all transitions, and all extended and regular states inside the problem definition. For an instance of the k-SMREACH problem containing a set $\mathcal{M} = \{M_1, \ldots, M_l\}$

of state machines, a global state $\hat{s} = (x_1, \ldots, x_l)$, and a partial state $\hat{s}_p = (g_1, \ldots, g_l)$, let $M_i = (S_i, \iota_i, A_i^{tr}, A_i^{\mathit{eff}}, T_i)$ and let the corresponding extended state machine be $M_i^* = (S_i \cup S_i^*, \iota_i, A_i^{tr}, A_i^{\mathit{eff}}, T_i^*)$ for each $i \in [1..l]$. The following sets collect transitions, symbols, and states over all involved state machines.

- $\mathcal{T} = \bigcup_{1 \leq i \leq l} T_i$, the set of all transitions,
- $A = \bigcup_{1 \leq i \leq l} \left(A_i^{tr} \cup A_i^{\mathit{eff}} \right)$, the set of all symbols,
- $\mathcal{S} = \bigcup_{1 \leq i \leq l} S_i$, the set of all states, and
- $\mathcal{S}^* = \bigcup_{1 \leq i \leq l} S_i^*$, the set of all extended states over the extended versions of the state machines.

The set $\mathcal{V}$ 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.

- $\mathsf{vs} = \{s^i \mid s \in \mathcal{S}, 0 \leq i \leq k\}$ is a set of variables that encode states at positions. If a state variable s^i is set to true, then the extended state machine to which s belongs is in state s at position i.
- $\mathsf{vx} = \{sx^i \mid sx \in \mathcal{S}^*, 0 \leq i \leq k\}$ is a set of variables that encode intermediate states at positions. If a state variable sx^i is set to true, then the extended state machine to which sx belongs is in state sx at position i.
- $\mathsf{vt} = \{t^i \mid t \in \mathcal{T}, 0 \leq i \leq k\}$ is a set of variables that encode transitions triggered at a position due to a message placed at that position. If a transition variable t^i is set to true, then the transition t is being triggered at position i.
- $\mathsf{va} = \{a^i \mid a \in A, 0 \leq i \leq k\}$ is a set of variables that encode whether a message symbol is available to be consumed by another machine at a certain position. If a variable a^i is set to true, then some extended state machine tries to send a at position i for $i > 0$, i.e., a transition has received a trigger and is waiting for a to be consumed by a different extended state machine in order to complete the transition. When a^j for $j > i$ is set to false, then the symbol is consumed at position j.

Note that we have defined a set of variables for the set $\mathcal{S}^*$ of extended states, but we have no set of variables encoding the additional transitions of the extended state machines. This is not necessary as there is exactly one additional transition for each extended state (cf Definition 3.3.4) and therefore it suffices to have a variable for each extended state.

In order to obtain a solution of a positive instance of the k-SMREACH problem, it suffices to obtain the variables evaluated to *true* in the logical model returned by the SAT solver. The following example describes how a solution is retrieved from a logical model.

Position	Symbols (va)	(Extended) States (vs ∪ vx)		
		PhD Student	Coffee Machine	Payment Unit
0		working^0	idle^0	ready^0
1	orderCoffee^1,money^1	sx1^1	idle^1	ready^1
2		waiting^2	sx1^2	sx1^2
3	ok^3	waiting^3	waiting^3	sx1^3
4		waiting^4	sx2^4	ready^4
5		waiting^5	preparing^5	ready^5
6		waiting^6	sx4^6	ready^6
7	error^7	waiting^7	error^7	ready^7
8		sx4^8	error^8	ready^8
9		sx4^9	error^9	ready^9
10		sx4^{10}	error^{10}	ready^{10}
11		desperate^{11}	error^{11}	ready^{11}

Table 4.3.1: Parts of a solution to reach the partial state (desperate, error) for the state machines depicted in Figures 4.3.1 and 4.3.3, showing variables of the sets va, vs, and vx.

Example 4.3.2. Table 4.3.1 shows a positive solution to the question whether the partial global state (desperate, error) can be reached by a path of length $k \leq 11$ for the three state machines depicted in Figures 4.3.1 and 4.3.3. Each word in the fields of columns "Symbols", "PhD Student", "Coffee Machine", and "Payment Unit" represents a variable evaluating to *true* in the logical model of the SAT encoding. All other variables of these sets are evaluating to *false*. The variable working^0 in the first row means that at the zeroth position, the state machine containing the state working is in that state. The field in column "Symbols" of this row is empty because no state machine is trying to send a symbol, so all variables in va with index 0 are evaluating to *false*.

At position 1, two symbols, orderCoffee^1 and money^1 are evaluating to *true* because a transition in state machine PhD Student has fired, and this state machine changes to the corresponding intermediate state. At position 2, these two symbols have disappeared, i.e., they are evaluating to *false*, because they are being consumed by state machines Coffee Machine and Payment Unit, which in turn change their states to an intermediate state. The communications continue, until at position 11, a global state matching the required partial state is reached. At positions 9 and 10 no changes occur, therefore the number of necessary steps to the matching goal state is $11 - 2 = 9$. ◊

To enhance the presentation of the formulas, we use the functions src, int, trg, eff, and tgt, which are defined as follows. Let $t = (s, trg, \mathit{eff}, s') \in \mathcal{T}$ be a transition of a state machine corresponding to the two transitions $(s, trg, \emptyset, s_t^*)$ and $(s_t^*, \epsilon, \mathit{eff}, s')$ in the respective extended state machine. Then $\mathsf{src}(t) = s$, $\mathsf{int}(t) = s_t^*$, $\mathsf{trg}(t) = trg$, $\mathsf{eff}(t) = \mathit{eff}$, and $\mathsf{tgt}(t) = s'$.

The first formula (Init) initializes the path by setting the states contained in the global state $\hat{s}$ with index 0 to *true*, and all other states and all symbols at index 0 to false. This means that at position 0, all state machines are in their state as specified by $\hat{s}$ and no symbol is waiting to be consumed.

$$\bigwedge_{j=1}^{l} \Bigg(\bigwedge_{s \in S_j, s = x_j} s^0 \wedge \bigwedge_{s \in S_j \cup S_j^*, s \neq x_j} \overline{s}^0 \Bigg) \wedge \bigwedge_{a \in A} \overline{a}^0 \tag{Init}$$

For a position $i > 0$, a variable from va can occur in its positive polarity. This means that the symbol has been made available as effect by a transition at position $j \in [1..i]$ and is waiting to be consumed by some transition as a trigger in a transaction on a position greater than i. When the symbol is consumed at a later position, the symbol variable occurs in its negative polarity with that index. Formula (4.3.1) ensures that whenever a transition is executed at some position i, then the state machine changes from the transition's source state at position i to its target state at position $i + 1$, the trigger symbol is set its negative polarity, and the effect symbols are set to their positive polarity. We use the expression $\mathsf{trg}(t) \neq \epsilon$ for the presentation of the formula. It is replaced by the logical constants $\top$ and $\bot$ during the generation of the formula.

$$\bigwedge_{i=0}^{k-1} \bigwedge_{t \in \mathcal{T}} \Bigg[t^i \rightarrow \Bigg(\mathsf{src}(t)^i \wedge \mathsf{int}(t)^{i+1} \wedge \Big(\mathsf{trg}(t) \neq \epsilon \rightarrow \Big(\mathsf{trg}(t)^i \wedge \overline{\mathsf{trg}(t)}^{i+1} \Big) \Big) \wedge \bigwedge_{\mathit{eff} \in \mathsf{eff}(t)} \Big(\overline{\mathit{eff}}^i \wedge \mathit{eff}^{i+1} \Big) \Bigg) \Bigg] \tag{4.3.1}$$

The two formulas (4.3.2) and (4.3.3) take care of the polarity of the effect symbols. Formula (4.3.2) ensures that if a state machine does not leave its intermediate state, then the effect symbols remain positive and formula (4.3.3) ensures that if it leaves its intermediate state, then all effect symbols are set to false at the following position.

$$\bigwedge_{i=0}^{k-1} \bigwedge_{\substack{t \in \mathcal{T} \\ \mathsf{eff}(t) \neq \emptyset}} \Big[\mathsf{int}(t)^i \wedge \mathsf{int}(t)^{i+1} \rightarrow \bigwedge_{\mathit{eff} \in \mathsf{eff}(t)} \mathit{eff}^{i+1} \Big] \tag{4.3.2}$$

$$\bigwedge_{i=0}^{k-1} \bigwedge_{\substack{t \in \mathcal{T} \\ \mathsf{eff}(t) \neq \emptyset}} \Big[\mathsf{int}(t)^i \wedge \overline{\mathsf{int}(t)}^{i+1} \rightarrow \bigwedge_{\mathit{eff} \in \mathsf{eff}(t)} \overline{\mathit{eff}}^{i+1} \Big] \tag{4.3.3}$$

Formula (4.3.4) ensures that if the state machine is in an intermediate state at position i and all effects have been consumed at position $i + 1$, then at position $i + 1$ the state machine leaves the intermediate state and changes into the target state of the transition.

$$\bigwedge_{i=0}^{k-1} \bigwedge_{t \in \mathcal{T}} \left[\left(\mathsf{int}(t)^i \wedge \bigwedge_{\mathit{eff} \in \mathsf{eff}(t)} \overline{\mathit{eff}}^{i+1} \right) \rightarrow \left(\overline{\mathsf{int}(t)}^{i+1} \wedge \mathsf{tgt}(t)^{i+1} \right) \right] \tag{4.3.4}$$

The three formulas (4.3.5), (4.3.6), and (4.3.7) are called *frame axioms*. They ensure that there is always a transition when changes of symbols (formulas (4.3.5) and (4.3.6)) or changes of states (formula (4.3.7)) occur.

$$\bigwedge_{i=0}^{k-1} \bigwedge_{\mathit{trg} \in A} \left[\mathit{trg}^i \wedge \overline{\mathit{trg}}^{i+1} \rightarrow \left(\left(\bigvee_{\substack{t \in \mathcal{T}, \\ \mathsf{trg}(t) = \mathit{trg}}} t^i \right) \wedge \bigwedge_{\substack{t_1, t_2 \in \mathcal{T}, \\ \mathsf{trg}(t_1) = \mathsf{trg}(t_2) = \mathit{trg}}} \left(\overline{t_1}^i \vee \overline{t_2}^i \right) \right) \right] \tag{4.3.5}$$

$$\bigwedge_{i=0}^{k-1} \bigwedge_{\mathit{eff} \in A} \left[\overline{\mathit{eff}}^i \wedge \mathit{eff}^{i+1} \rightarrow \left(\left(\bigvee_{\substack{t \in \mathcal{T}, \\ \mathit{eff} \in \mathsf{eff}(t)}} t^i \right) \wedge \bigwedge_{\substack{t_1, t_2 \in \mathcal{T}, \\ \mathsf{eff}(t_1) = \mathsf{eff}(t_2), \\ \mathsf{eff}(t_1), \mathsf{eff}(t_2) \in \mathit{eff}}} \left(\overline{t_1}^i \vee \overline{t_2}^i \right) \right) \right] \tag{4.3.6}$$

$$\bigwedge_{i=0}^{k-1} \bigwedge_{s \in S} \left[s^i \wedge \overline{s}^{i+1} \rightarrow \bigvee_{t \in \mathcal{T}, s = \mathsf{src}(t)} t^i \right] \tag{4.3.7}$$

Formula (4.3.8) expresses that each state machine is in exactly one state at each position.

$$\bigwedge_{i=0}^{k-1} \bigwedge_{j=1}^{l} \left[\left(\bigvee_{s \in (S_j \cup S_j^*)} s^i \right) \wedge \bigwedge_{s_1, s_2 \in (S_j \cup S_j^*), s_1 \neq s_2} \left(\overline{s_1}^i \vee \overline{s_2}^i \right) \right] \tag{4.3.8}$$

Finally, formula (Goal) encodes the goal and the (extended) states of its environment for index k. Recall that the goal $\hat{s}_p = (g_1, \ldots, g_l)$ contains the symbol ς when for a state machine no state is defined for the goal.

$$\bigwedge_{i=1, g_i \neq \varsigma}^{l} \left(g_i^k \vee \bigvee_{s \in \mathsf{env}(g_i)} s^k \right) \tag{Goal}$$

The formulas are converted to CNF (cf. Section 2.2), the input format of most SAT solvers. To this end, we apply the Tseitin transformation [114] where necessary.

The encoding allows that nothing happens, i. e., no transaction takes place at some position. In this case, the frame axioms ensure that the global state remains the same. This relaxation implicitly encodes the "at most k" formulation of the problem: If at n positions nothing changes and the goal is reached at index k for $k \geq n$, it means that the length of the path is $k - n$.

Example 4.3.2 (Cont. from p. 57). In the solution shown in Table 4.3.1 at position 2, the two symbols orderCoffee and money are consumed (they disappear as variables evaluating to *true* at that position) and therefore the states of all state machines change. In particular, the state machine PhD Student changes into an original state because the transition's symbols orderCoffee and money are consumed, and the state machines Coffee Machine and Payment Unit change into an intermediate state because they consume these symbols. At positions where states change without symbols being consumed, as at positions 1, 3, 5, 6, 7, and 11, the state changes are triggered by the empty symbol ϵ. At positions 9 and 10, nothing changes, so these two positions can be removed. Then, the goal state is reached in 9 steps, which is less then k. $\Diamond$

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 (i.e., variables from the sets vs, vx, and vt) we obtain the path of at most k steps leading to the goal state. If the length of the path is less than k, then for some consecutive indices the state variables represent identical states.

In order to simplify the encoding we assume that at each position, each symbol can be consumable only once. 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 [110].

It can be verified that this encoding is of size polynomial with respect to the size of the instance of k-SMREACH.

Observation 4.3.1. Given an instance $\mathcal{R}$ of the k-SMREACH problem, its encoding as a conjunction of the formulas (Init), (Goal), and (4.3.1) to (4.3.8) is of size polynomial with respect to the size of $\mathcal{R}$ because any of the formulas (Init), (Goal), and (4.3.1) to (4.3.8) contains at most two nested iterations over sets contained in the input.

4.3.4 Computational Complexity

We show that the k-SMREACH problem is NP-complete. Its membership in NP follows directly from Observation 4.3.1, i.e., from the fact that the k-SMREACH problem can be encoded in polynomial time with respect to its input size into a formula in propositional logic.

Corollary 4.3.1. The k-SMREACH problem is in NP.

In order to prove NP-hardness of the k-SMREACH problem, we reduce a variation of the 3-satisfiability problem [54, Appendix A9.1, L02], the 3X3SAT problem, to the k-SMREACH problem. The definition of the 3X3SAT problem allows each variable to occur at most three times and each literal to occur at most twice in a propositional CNF formula where each clause contains at most three literals. The 3X3SAT problem has been shown to be NP-complete [99, Proposition 9.3].

Definition 4.3.5 (3X3SAT)**.**

Instance: A set $\mathcal{V}$ of variables and a collection $\mathcal{C}$ of clauses over $\mathcal{V}$ such that for each clause $C \in \mathcal{C}$ it holds that $|C| = 3$, each variable occurs at most three times in $\mathcal{C}$, and each literal occurs at most twice in $\mathcal{C}$.

Question: Is $\mathcal{C}$ satisfiable?

$\triangle$

In our reduction we construct two sets $\mathcal{M}_{vars}$ and $\mathcal{M}_{clauses}$ of state machines, where $\mathcal{M}_{vars}$ contains a state machine for each variable in $\mathcal{V}$, and $\mathcal{M}_{clauses}$ contains a state machine for each clause in $\mathcal{C}$. The alphabet of effects of the state machines in $\mathcal{M}_{vars}$ and the alphabet of triggers of the state machines in $\mathcal{M}_{clauses}$ contain a different symbol for each occurrence of each variable in $\mathcal{C}$.

Each state machine M_v in $\mathcal{M}_{vars}$ contains one state and one transition for each occurrence of a variable v in $\mathcal{C}$, and an initial state. This state machine does not receive any symbols, it only sends symbols. Thus, all triggers on all its transitions are ϵ. The states are aligned along two branches of transitions. Transitions on one branch refer to the occurrences of v in clauses as positive literals and transitions on the second branch to the occurrences of v as negative literals. This way, a state machine in M_v cannot send symbols representing opposite literals of a variable in the same run.

Each state machine M_C for clause C in $\mathcal{M}_{clauses}$ contains two states, an initial state and a sink state, and for each literal contained in clause C, M_C contains one transition connecting the two states and one transition looping the sink state. Opposite to those in $\mathcal{M}_{vars}$, state machines in $\mathcal{M}_{clauses}$ only receive, but never send symbols. Their transitions therefore contain a trigger other than ϵ and the empty set as effects. Each trigger on a transition connecting the two states and on a transition looping the sink state in M_C corresponds to a literal in C and vice versa. The global state of the reduced instance is the set of all initial states in $\mathcal{M}_{vars}$ and $\mathcal{M}_{clauses}$ and the goal state is the set of sink states in $\mathcal{M}_{clauses}$.

With this construction, each state machine in $\mathcal{M}_{vars}$ sends symbols representing literals that evaluate to *true* under one of the interpretations *true* or *false* of its variable to state machines in $\mathcal{M}_{clauses}$. Since there is no link between the two branches of a state machine in $\mathcal{M}_{vars}$, only one branch can be contained in a path and hence only one interpretation be constructed for the clauses in $\mathcal{C}$. When a state machine M_C in $\mathcal{M}_{clauses}$ reaches its sink state, then the clause C is satisfied. Hence, when all state machines in $\mathcal{M}_{clauses}$ reach their sink state, then the propositional formula is satisfiable. If they cannot reach their sink state, then the propositional formula is unsatisfiable. The loops on the sink states prevent the state machines in $\mathcal{M}_{vars}$ from getting stuck in a state when the sink state of a state machine in $\mathcal{M}_{clauses}$ has already been reached by receiving a symbol from a different state machine in $\mathcal{M}_{vars}$.

We set k to $4|\mathcal{V}|$ for the following reasons. A path along one branch of a state machine in $\mathcal{M}_{vars}$ can contain at most two transitions (recall that the 3X3SAT problem allows a literal to occur at most twice) and the conversion into an extended state machine doubles the number of transitions. Further, all variables may be necessary to satisfy the formula and therefore, all state machines in $\mathcal{M}_{vars}$ may be required to reach their sink state. Also, in the worst case, one transaction can only contain one message, requiring one transaction for each transition in the extended state machines for $\mathcal{M}_{vars}$.

Example 4.3.3. Table 4.3.2 shows a positive instance $\mathcal{S} = (\mathcal{V}, \mathcal{C})$ of 3X3SAT with four clauses and three variables. Figure 4.3.4 depicts the set $\mathcal{M}$ of state machines of the k-SMREACH instance $\mathcal{R} = (\mathcal{M}, \hat{s}, \hat{s}_p, k)$ reduced from $\mathcal{S}$. The set $\mathcal{M}_{vars}$ contains three state machines, one for each variable and $\mathcal{M}_{clauses}$ contains four state machines, one for each clause. The initial global state in $\mathcal{R}$ is $\hat{s} = (x_\iota, y_\iota, z_\iota, \iota_1, \iota_2, \iota_3, \iota_4)$, the goal state is $\hat{s}_p = (g_1, g_2, g_3, g_4)$, and k is $4|\mathcal{M}_{vars}| = 12$.

Clause	Literals
1	$\{x, \overline{y}\}$
2	$\{x, y\}$
3	$\{z\}$
4	$\{\overline{x}, \overline{y}, \overline{z}\}$

Table 4.3.2: A satisfiable instance $\mathcal{S}$ of 3X3SAT.

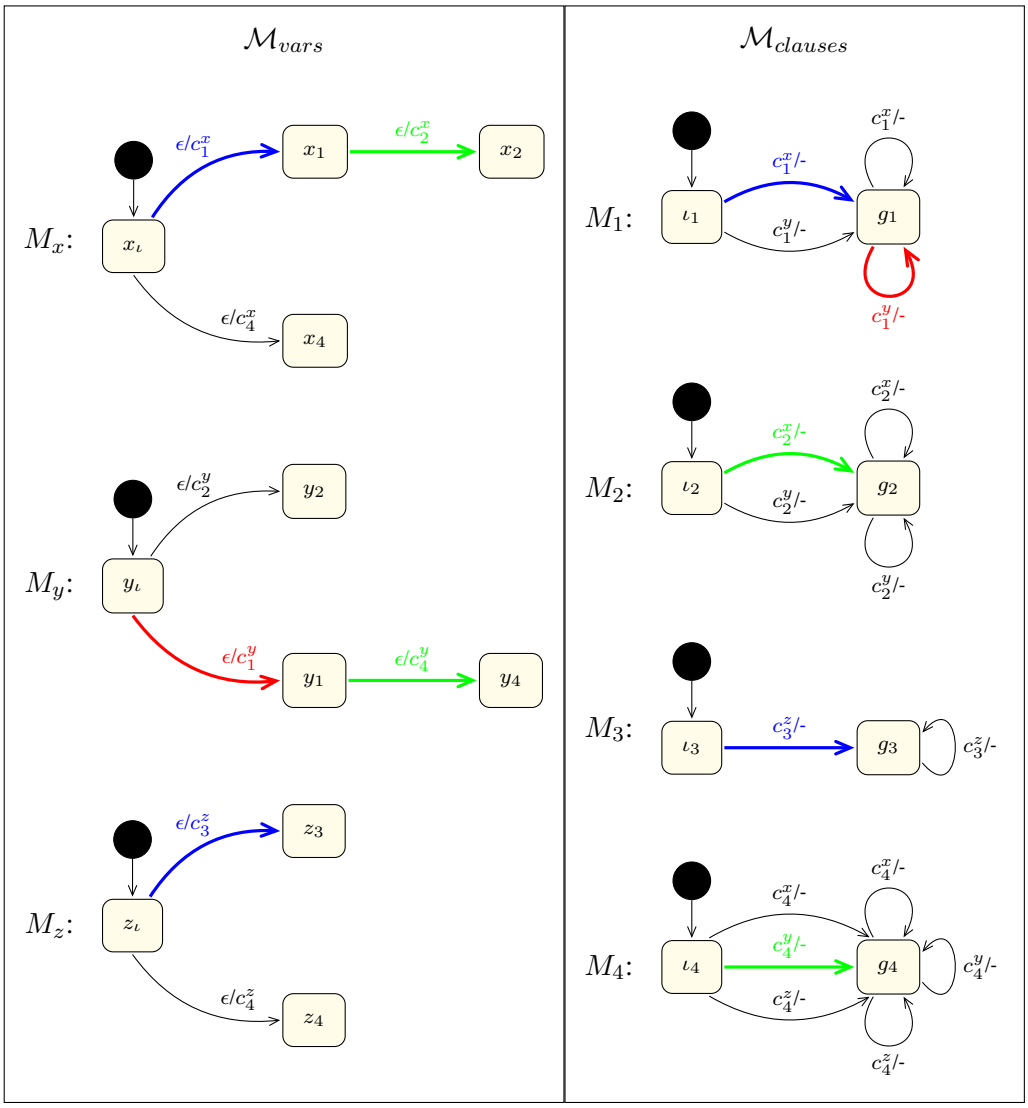

Figure 4.3.4: State machines reduced from the 3X3SAT instance in Table 4.3.2. Highlighted transitions represent a path leading to the goal state. The transitions highlighted in blue belong to the first transaction, those highlighted in red to the second transaction, and those in green to the third transaction.

The instance $\mathcal{S}$ is satisfied by the assignment $x \mapsto$ *true*, $y \mapsto$ *false*, $z \mapsto$ *true*. Under this assignment, the occurrences of x evaluate to *true* in the clause C_1 and the clause C_2, and to *false* in the clause C_4, therefore satisfying C_1 and C_2. This evaluation corresponds to the upper branch of the state machine M_x, which sends the symbols c_1^x and c_2^x leading state machines M_1 and M_2 to their goal states. Similarly, the literal $\overline{y}$ evaluates to *true* in clause C_1 and in clause C_4, and the literal y to *false* in clause C_2. The two evaluations of $\overline{y}$ to *true* correspond to the lower branch of the state machine M_y, leading state machine M_4 to its goal state and looping the sink state of M_1 (which has already reached its goal by the evaluation of x). The evaluation of z leads M_3 to its goal in a similar way. Therefore, $\mathcal{R}$ is a positive instance of k-SMREACH.

This example shows the role of the looping transitions on the sink state of the state machines in $\mathcal{M}_{clauses}$: Without the looping transition receiving c_1^y in M_1, M_y could not send the symbol c_4^y to M_4 and therefore, M_4 would not reach its sink state. The communication of the symbols can be found as the path

$$[\{(M_x, c_1^x, M_1), (M_z, c_3^z, M_3)\},\\ \{(M_y, c_1^y, M_1)\},\\ \{(M_x, c_2^x, M_2), (M_y, c_4^y, M_4)\}]$$

of length 3 (some brackets are omitted as the multimessages are single messages). ◊

Lemma 4.3.1. The k-SMREACH problem is NP-hard.

Proof. Given an instance $\mathcal{S} = (\mathcal{V}, \mathcal{C})$ of the 3X3SAT problem, we construct an instance $\mathcal{R} = (\mathcal{M}, \hat{s}, \hat{s}_p, k)$ of the k-SMREACH problem as follows.

$\mathcal{M}$ is the union of two sets $\mathcal{M}_{vars}$ and $\mathcal{M}_{clauses}$ of state machines constructed as follows. For each variable v in $\mathcal{V}$, let $\mathcal{C}_v = \{C \mid v \in \mathsf{vars}(C)\}$ be the set of clauses containing a literal of v. Then the set $\mathcal{M}_{vars}$ contains for each variable v in $\mathcal{V}$ a state machine $M_v = (S_v, \iota_v, A_v^{tr}, A_v^{\mathit{eff}}, T_v)$ with

- $S_v = \{v_\iota\} \cup \{v_C \mid C \in \mathcal{C}_v\}$,
- $\iota_v = v_\iota$,
- $A_v^{tr} = \{\epsilon\}$,
- $A_v^{\mathit{eff}} = \{c_C^v \mid C \in \mathcal{C}_v\}$, and
- $$T_v = \begin{cases} \{(v_\iota, \epsilon, \{c_A^v\}, v_A), (v_A, \epsilon, \{c_B^v\}, v_B), (v_\iota, \epsilon, \{c_C^v\}, v_C) \\ \quad \mid \ell \in A, \ell \in B, \overline{\ell} \in C, \mathsf{var}(\ell) = v, A, B, C \in \mathcal{C}_v\} & \text{if } |\mathcal{C}_v| = 3 \\ \{(v_\iota, \epsilon, \{c_A^v\}, v_A), (v_A, \epsilon, \{c_B^v\}, v_B) \\ \quad \mid \ell \in A, \ell \in B, \mathsf{var}(\ell) = v, A, B \in \mathcal{C}_v\} \ \cup \\ \quad \{(v_\iota, \epsilon, \{c_A^v\}, v_A), (v_\iota, \epsilon, \{c_B^v\}, v_B) \\ \quad\quad \mid \ell \in A, \overline{\ell} \in B, \mathsf{var}(\ell) = v, A, B \in \mathcal{C}_v\} & \text{if } |\mathcal{C}_v| = 2 \\ \{(v_\iota, \epsilon, \{c_A^v\}, v_A) \mid \ell \in A, \mathsf{var}(\ell) = v, A \in \mathcal{C}_v\} & \text{if } |\mathcal{C}_v| = 1. \end{cases}$$

The set $\mathcal{M}_{clauses}$ contains a state machine $M_C = (S_C, \iota_C, A_C^{tr}, A_C^{\mathit{eff}}, T_C)$ for each clause C in $\mathcal{C}$ with

- $S_C = \{\iota_C, g_C\}$,
- $\iota_C = \iota_C$,
- $A_C^{tr} = \{c_C^v \mid v \in \mathsf{vars}(C)\}$,
- $A_C^{\mathit{eff}} = \emptyset$, and
- $T_C = \{(\iota_C, c_C^v, \emptyset, g_C), (g_C, c_C^v, \emptyset, g_C) \mid v \in \mathsf{vars}(C)\}$.

Further, we set the global state $\hat{s} = \{\iota_v \mid M_v \in \mathcal{M}_{vars}\} \cup \{\iota_C \mid M_C \in \mathcal{M}_{clauses}\}$ such that it contains all initial states of the state machines in $\mathcal{M}$, the goal state $\hat{s}_p = \{g_C \mid M_C \in \mathcal{M}_{clauses}\}$ such that it contains the sink states of the state machines in $\mathcal{M}_{clauses}$, and $k = 4|\mathcal{M}_{vars}|$.

The size of the reduced instance $\mathcal{R}$ of k-SMREACH is linear with respect to the size of the instance $\mathcal{S}$ of 3X3SAT. In particular, it contains for each variable one state machine with at most four states and three transitions, and for each clause one state machine with two states and at most six transitions. We proceed with showing that $\mathcal{S}$ is a positive instance of 3X3SAT if and only if $\mathcal{R}$ is a positive instance of k-SMREACH.

($\Rightarrow$) If $\mathcal{S}$ is a positive instance of 3X3SAT, then there exists an interpretation σ for each variable in $\mathcal{V}$ under which the formula evaluates to *true*. This means that in each clause, at least one literal evaluates to *true* (cf. Section 2.2). By construction, a state machine M_v in $\mathcal{M}_{vars}$ can send one of two sequences of symbols representing the clauses satisfied due to an assignment of v to *true* or to *false* respectively, or one sequence if the literal only occurs in one polarity in the formula. The state machines in $\mathcal{M}_{clauses}$ can receive the symbols representing each clause's literals. Receiving these symbols leads the state machine's initial state to the sink state or makes the state machine stay in the sink state. It is always possible to execute all transitions along a branch of a state machine in $\mathcal{M}_{vars}$ because the symbols can always be received by some state machine in $\mathcal{M}_{clauses}$, either by a transition between source and sink state or by a transition looping the sink state. A branch of a state machine in $\mathcal{M}_{vars}$ contains at most two transitions, which correspond to four transitions in the extended state machine. Assuming the case where each transaction in the path contains only one message, we need $k = 4|\mathcal{M}_{vars}|$ transactions to reach the goal state. Since under σ, at least one literal of each clause evaluates to *true* and exactly one state machine in $\mathcal{M}_{clauses}$ corresponds to one clause, all state machines in $\mathcal{M}_{clauses}$ reach their sink state. This makes $\mathcal{R}$ a positive instance of k-SMREACH.

($\Leftarrow$) If $\mathcal{R}$ is a positive instance of SMREACH, then a path exists such that a partial global state containing all sink states of the state machines in $\mathcal{M}_{clauses}$ can be reached from the global state containing all state machines' initial states. This path contains for each state machine M_C in $\mathcal{M}_{clauses}$ a transaction containing one of the symbols occurring as trigger on one of the transitions between the initial state and the sink state of M_C. The senders of these symbols are state machines in $\mathcal{M}_{vars}$, each along one of its branches. Since each branch corresponds to an assignment to a variable in $\mathcal{S}$ and each state machine in $\mathcal{M}_{clauses}$ corresponds to a clause in $\mathcal{S}$, an assignment satisfying the clauses in $\mathcal{S}$ can be retrieved from the branches. Note that, since

exactly one branch of each state machine in $\mathcal{M}_{vars}$ is contained in the path, it is ensured that a variable is assigned exactly one truth value. Therefore, $\mathcal{S}$ is a positive instance of 3X3SAT. □

Theorem 4.3.1. The k-SMREACH problem is NP-complete.

Proof. The theorem follows from Corollary 4.3.1 and Lemma 4.3.1 □

4.4 Model Checking – the k-SDCHECK Problem

In this section we deal with the k-SDCHECK problem, which concerns k-consistency (cf. Section 3.3, Definition 3.3.12) between state machines and sequence diagrams. The k-SDCHECK problem asks whether the communication described by a sequence diagram can be executed by a set of state machines after at most k steps from the global state containing all initial states. Similarly to a logical model for a satisfiable formula in propositional logic, for this problem a witness can be returned for positive instances. This witness consists of a concrete communication trace leading to a global state from which the sequence diagram can be executed.

Otherwise, if the instance is negative and therefore the message sequence cannot be executed, then the first message of the sequence diagram that cannot be applied can be computed. To this end, we systematically remove messages from the sequence diagram and solve these new instances until a positive instance is found. The obtained information on up to which message the sequence diagram can be executed can be used for debugging purposes. On this basis, inconsistencies introduced during the evolution of a model cannot only be discovered easily, but also be immediately corrected.

The scenarios modeled by the sequence diagram view of a software model can be interpreted as either required or forbidden sequences of message exchange. A positive instance of this problem, that is, a sequence diagram that can be executed by the set of state machines, can therefore be interpreted as a proof of the existence of a required behavior or as a proof of an error in the system. This way, sequence diagrams can be used as test cases in different testing scenarios. Similarly to the neg fragment used in UML state machines, we mark unwanted scenarios using this notation and regard unmarked scenarios as wanted. As in the previous section, we refer to state machines instead of instances of state machines whenever the meaning is clear.

We first give an intuition of this problem by a similar example as in Section 4.3 and then describe the problem formally. To solve this consistency checking problem, we propose to use a similar encoding as for solving the k-SMREACH problem (cf. Section 4.3). We finally show that the problem is NP-complete by a proof similar to the proof in Section 4.3.

4.4.1 A Motivating Example

Figure 4.4.1 shows three state machines that describe the behaviors of a PhD student, a coffee machine, and a maintenance unit for the coffee machine, Figure 4.4.2 shows the extended state machines for the state machines in Figure 4.4.1, and Figure 4.4.3 shows two sequence diagrams that describe communication scenarios between instances of the state machines in Figure 4.4.1.

A state machine is instantiated by one or more lifelines. In order to be *consistent* with the state machines, the message sequence of a sequence diagram must be executable from some

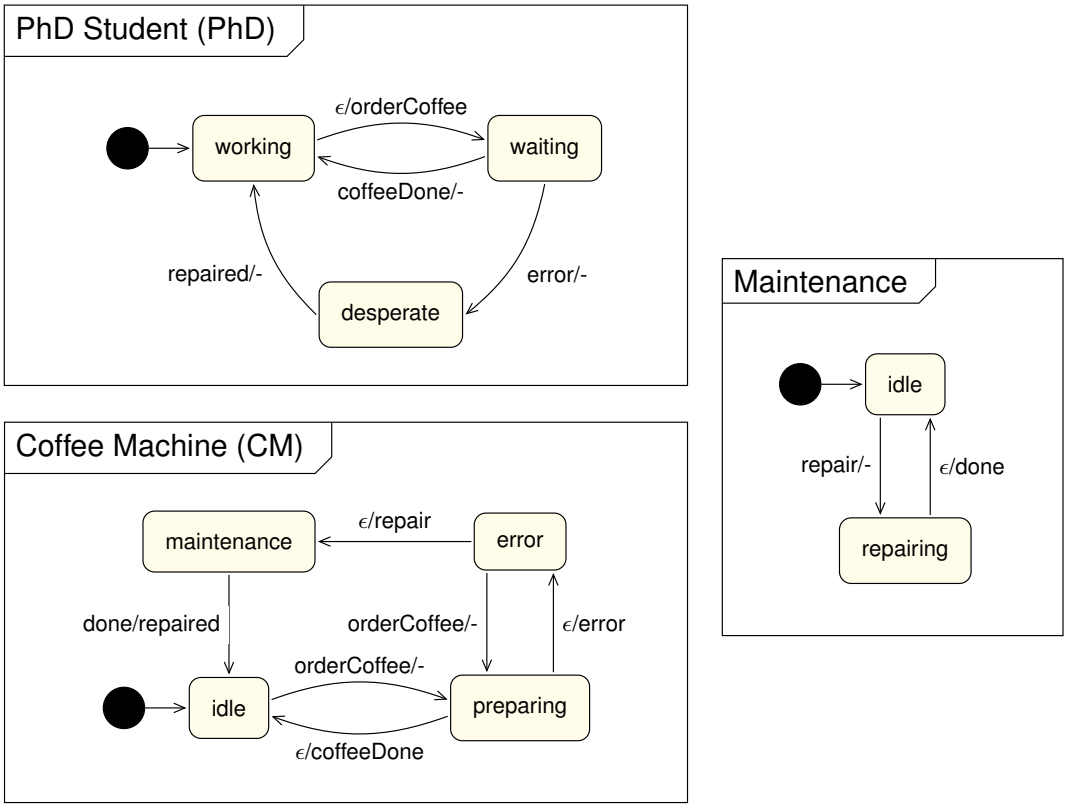

Figure 4.4.1: Three state machines modeling a PhD student, a coffee machine, and a maintenance unit.

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. In between the messages of the sequence diagram only empty messages (which trigger transitions with the symbol ϵ as trigger) can be sent.

We distinguish two possible application scenarios for this problem. First, the scenario depicted in the sequence diagram can be desired. 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. Second, the scenario depicted in the sequence diagram can be undesired. In this case, if the 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 countertrace, namely a sequence of global states which follows from the application of the message sequence.

An example for each scenario is depicted in Figure 4.4.3. The left sequence diagram shows a desired scenario. However, it is inconsistent with the state machines for the following reason.

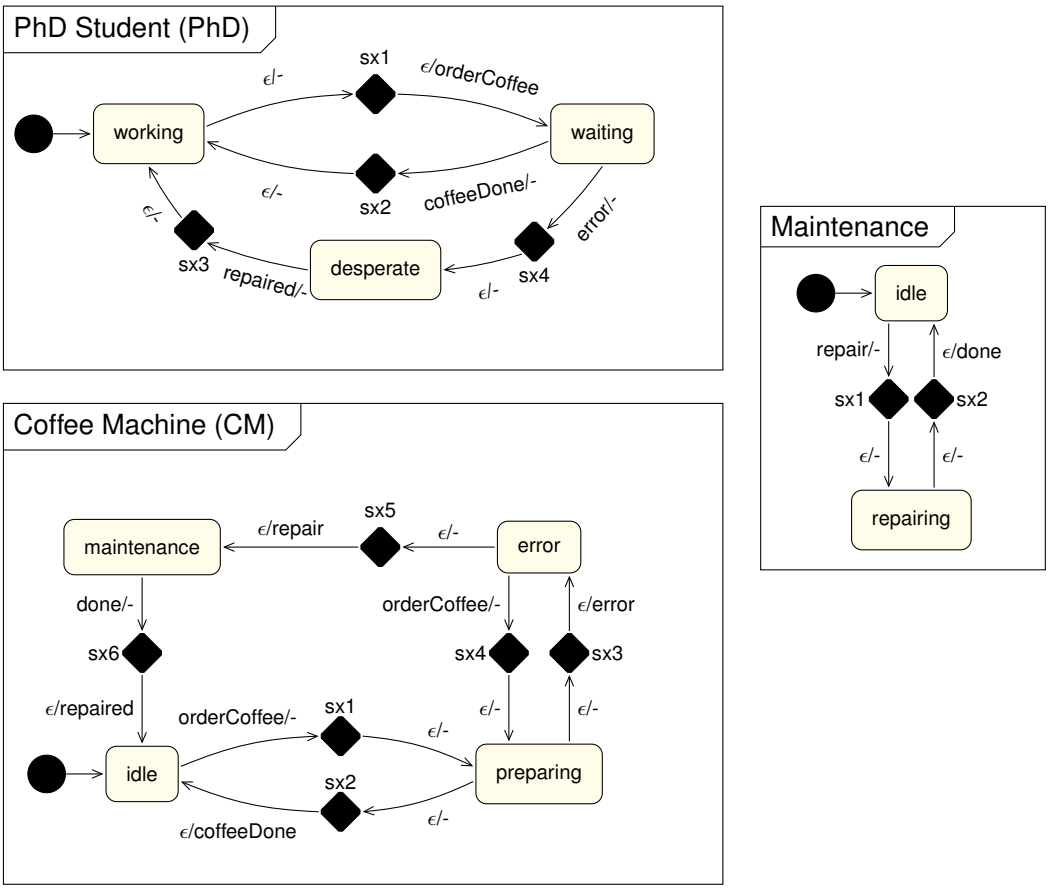

Figure 4.4.2: The extended state machines corresponding to those in Figure 4.4.1.

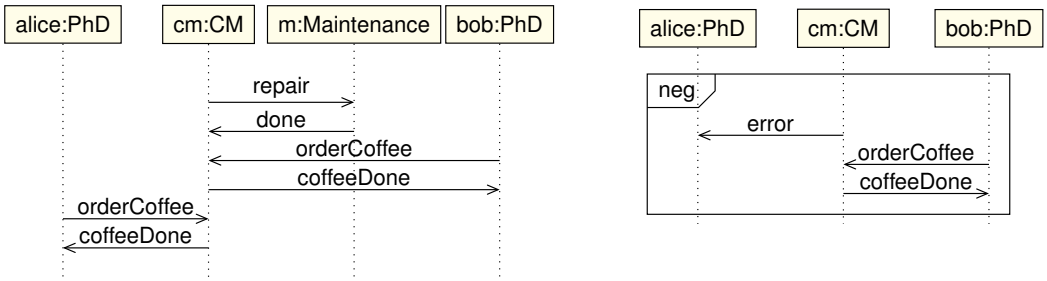

Figure 4.4.3: (Left) A sequence diagram depicting a desired scenario that is inconsistent with the state machines of Figure 4.4.1. The state machines have to be changed in order to allow the scenario. (Right) A sequence diagram depicting a forbidden scenario that is consistent with the state machines of Figure 4.4.1. The state machines have to be changed in order to forbid the scenario.

The first message of the sequence diagram, repair, can only be sent from the intermediate state sx5 of the instance cm of Coffee Machine. This state can be reached, and then the message repair can be sent by cm and consumed by the instance m of the state machine Maintenance. Also, the following message done can be sent at the intermediate state between the states repairing and idle of state machine Maintenance and received at state maintenance of state machine Coffee Machine. After receiving done, the instance cm is in intermediate state sx6, from which it can continue only after the symbol repaired is received by some other state machine. However, this never happens in the sequence diagram and so the coffee machine never returns to state idle and therefore cannot receive the symbol orderCoffee from PhD student bob.

We can now automatically discover the sequence of messages up to the message that cannot be sent or received by systematically removing messages from the sequence diagram. In the current example, the sequence diagram can be executed up to and including the message 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.

The second diagram shows an unwanted scenario. It allows the coffee machine to prepare coffee after receiving the error signal, but without receiving the repair signal. This scenario is implemented in the state machines, so this also indicates a bug. The path to the global state from which the sequence can be executed contains an empty message received by the instance alice of PhD Student followed by sending orderCoffee from alice to instance cm of Coffee Machine and another empty message received by cm. Then, cm is in the intermediate state on the transition from preparing to error from where it can send a message containing the symbol error back to alice and also the rest of the messages of the sequence diagram can be executed. The path to the global state $(\mathsf{waiting}, \mathsf{sx3}, \mathsf{idle})$ from where the sequence diagram can be executed hence consists of an empty message, the message $(\mathsf{alice}, \mathsf{orderCoffee}, \mathsf{cm})$, and two more empty messages.

4.4.2 Problem Definition

Given a sequence diagram and a set of communicating state machines modeling the behavior of the lifelines in the sequence diagram, the *Multiview Sequence Consistency (k-SDCHECK) Problem* asks whether, from some global state that is reachable in k steps from the global initial state, there is a path representing the sequence of messages described in the sequence diagram. The sequence of messages in a sequence diagram is interpreted as path that contains only singleton transactions of single messages and may contain empty messages in between them.

If such a path exists, then we call the two views *k-consistent* (cf. Section 3.3, Definition 3.3.12). The desired outcome of a wanted scenario (no neg label) depicted in a sequence diagram is to be consistent with the state machine view, which means that the desired scenario is indeed implemented in the state machines. The desired outcome of an unwanted scenario (neg label) is to be inconsistent with the state machine view, which means that the state machines do not implement the undesired trace.

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

Definition 4.4.1 (*k-Multiview Sequence Consistency (k-*SDCHECK*) Problem*)**.**

Instance: A set $\mathcal{M} = \{M_1, \ldots, M_l\}$ of state machines over the universe $\mathcal{A}$ with $M_i = (S_i, \iota_i, A_i^{tr}, A_i^{\mathit{eff}}, T_i)$ for $i \in [1..l]$ where for all $i, j \in [1..l]$ with $i \neq j$ it holds that S_i, S_j are disjoint and T_i, T_j are disjoint, a sequence diagram $D = (\mathcal{L}, \mu)$ over $\mathcal{M}$ and $\mathcal{A}$, and a positive integer $k = \mathsf{p}(l)$ for some polynomial p.

Question: Are $\mathcal{M}$ and D k-consistent?

△

As in Section 4.3, we define the parameter k as polynomial in the number of state machines for complexity theoretical reasons. The encoding presented below would be of exponential size with respect to the input if k is exponential in the number of state machines.

4.4.3 Encoding to SAT

To solve the k-SDCHECK problem we propose to encode it to the satisfiability problem of propositional logic (SAT). To this end, we build a propositional formula representing an instance of the k-SDCHECK problem and hand it to a SAT solver. The solver returns `SAT` and a logical model if the sequence diagram of the k-SDCHECK problem instance can be executed after at most k transactions between state machines instantiated by the lifelines. Otherwise, it returns `UNSAT`. The logical model can then be translated back into a concrete sequence of transactions between the lifelines and to the transitions triggered by the application of these transactions. The sequence of transactions represents the path from the global initial state to the global state from which the sequence diagram can be executed and the sequence of messages of the sequence diagram. The solver returns `UNSAT` if the sequence diagram cannot be executed by the state machines after at most k message exchanges.

As in the previous sections, we first describe the sets of variables used in the encoding and then present the encoding as a set of formulas. The encoding is a modification of the encoding discussed in Section 4.3 where we determined the reachability of a partial global state regardless of a particular message sequence. As in Section 4.3, we deal with k-consistency (cf. Section 3.3, Definition 3.3.12) defined over extended state machines (cf. Section 3.3, Definition 3.3.4).

We encode an instance of the k-SDCHECK problem as a propositional formula over a set $\mathcal{V}$ of variables representing original states, intermediate states, transitions, and alphabet symbols.

Let $\mathcal{M} = \{M_1, \ldots, M_l\}$ be a set of state machines over the universe $\mathcal{A}$. For $i \in [1..l]$, let $M_i = (S_i, \iota_i, A_i^{tr}, A_i^{\mathit{eff}}, T_i)$ and $M_i^* = (S_i \cup S_i^*, \iota_i, A_i^{tr}, A_i^{\mathit{eff}}, T_i^*)$, i.e., a state machine and its extended state machine. Further, let $D = (\mathcal{L}, \mu)$ be a sequence diagram over $\mathcal{M}$ with the set $\mathcal{L} = \{L_1, \ldots, L_l\}$ of lifelines and the sequence $\mu = [N_1, \ldots, N_m]$ of messages. Then the following sets collect transitions, symbols, and states over all involved state machines:

- $\mathcal{T} = \bigcup_{1 \leq i \leq l} T_i$, the set of all transitions,
- $A = \bigcup_{1 \leq i \leq l} \left(A_i^{tr} \cup A_i^{\mathit{eff}} \right)$, the set of all symbols,
- $\mathcal{S} = \bigcup_{1 \leq i \leq l} S_i$, the set of all states, and
- $\mathcal{S}^* = \bigcup_{1 \leq i \leq l} S_i^*$, the set of all extended states over the extended versions of the state machines.

Recall that k is an integer defining the maximum length of the path leading to a global state from which the message sequence in D is executed. However, we need variables for positions greater than k in order to ensure the correct execution of the sequence diagram after k steps. In particular, for a sequence diagram with n messages, we need $4n$ additional positions. The factor 4 leaves room for three empty messages between the messages of the sequence diagram in order to execute transitions with ϵ as trigger or the empty set as effects. It can never happen that more than three additional messages are necessary for the state machines to send and receive the next message in the sequence diagram because each transition in a state machine has a trigger, a non-empty set of effects, or both. This way, in the corresponding extended state machine, at most three positions are necessary to process empty messages between (intermediate) states from where non-empty messages are sent or received. Therefore, the variables are indexed up to $k' = k + 4n$ and also most formulas of the encoding count up to k'.

As in Section 4.3, the set $\mathcal{V}$ of variables is the union of the following sets.

- $\mathsf{vs} = \{s^i \mid s \in \mathcal{S}, 0 \leq i \leq k'\}$ is a set of variables that encode states at positions. If a state variable s^i is set to true, then the extended state machine to which s belongs is in state s at position i.

- $\mathsf{vx} = \{sx^i \mid sx \in \mathcal{S}^*, 0 \leq i \leq k'\}$ is a set of variables that encode intermediate states at positions. If a state variable sx^i is set to true, then the extended state machine to which sx belongs is in state sx at position i.

- $\mathsf{vt} = \{t^i \mid t \in \mathcal{T}, 0 \leq i \leq k'\}$ is a set of variables that encode transitions triggered at a position due to a message placed at that position. If a transition variable t^i is set to true, then the transition t is being triggered at position i.

- $\mathsf{va} = \{a^i \mid a \in A, 0 \leq i \leq k'\}$ is a set of variables that encode whether a message symbol is available to be consumed by another machine at a certain position. If a variable a^i is set to true, then some extended state machine tries to send a at position i for $i > 0$, i.e., a transition has received a trigger and is waiting for a to be consumed by a different extended state machine in order to complete the transition. When a^j for $j > i$ is set to false, then the symbol is consumed at position j.

Note that we have defined a set of variables for the set $\mathcal{S}^*$ of extended states, but we have no set of variables encoding the additional transitions of the extended state machines. This is not necessary as there is exactly one additional transition for each extended state (cf Definition 3.3.4) and therefore it suffices to have a variable for each extended state.

Similarly as for the k-SMREACH problem in Section 4.3, a solution of a positive instance of the k-SMREACH problem can be retrieved by obtaining the meaning of the variables evaluating to *true* in the logical model returned by the SAT solver. The following example describes how a solution is retrieved from a logical model.

Example 4.4.1. Table 4.4.1 shows a positive witness to the question whether the sequence diagram on the right in Figure 3.1.4 is k-consistent for $k = 4$ with the set of state machines depicted in Figure 4.4.1 and 4.4.2. Each word in the fields of columns “Symbols”, “alice:PhD”,

Position	Symbols (va)	(Extended) States (vs ∪ vx)		
		alice:PhD	cm:CM	bob:PhD
0		working^0	idle^0	working^0
1	orderCoffee^1	sx1^1	idle^1	working^1
2		waiting^2	sx1^2	working^2
3	orderCoffee^3	waiting^3	preparing^3	sx1^3
4	error^4,orderCoffee^4	waiting^4	sx3^4	sx1^4
5	orderCoffee^5	sx4^5	error^5	sx1^5
6		desperate^6	sx4^6	waiting^6
7		desperate^7	preparing^7	waiting^7
8		desperate^8	preparing^8	waiting^8
9		desperate^9	preparing^9	waiting^9
10	coffeeDone^{10}	desperate^{10}	sx2^{10}	waiting^{10}
11		desperate^{11}	idle^{11}	sx2^{11}
12		desperate^{12}	idle^{12}	sx2^{12}

Table 4.4.1: A solution to executing the right-hand sequence diagram of Figure 4.4.3 after a path of length 4 in the state machines of Figures 4.4.1 and 4.4.2, showing variables of the sets va, vs, and vx.

"cm:CM", and "bob:PhD" represents a variable evaluating to *true* in the logical model of the SAT encoding. All other variables of these sets are evaluating to *false*. The variable working^0 in the first row means that at the zeroth position, the state machine containing the state working is in the state working. The field in column "Symbols" of this row is empty because no state machine is trying to send a symbol, so all symbol variables are evaluating to *false*. At position 1, the symbol orderCoffee is evaluating to *true* because a transition in the instance alice of state machine PhD has fired, and alice changes into the corresponding intermediate state. At position 2, this symbol has disappeared, i.e., is evaluating to *false*, because it is being consumed by instance cm of CM, which in turn changes its state to an intermediate state. At position 4, the symbols error and orderCoffee are being tried to be sent by cm and bob. At position 5, the symbol error is being consumed by alice, therefore the symbol variable evaluates to *false*, alice changes its state to the intermediate state sx4, and cm changes its state to error. This corresponds to the first message in the sequence diagram At position 6, alice changes its state to desperate by an empty message, the evaluation of symbol orderCoffee changes to *false*, and cm, which consumes the symbol orderCoffee, changes its state to the intermediate state sx4. This corresponds to the second message of the sequence diagram. The remaining messages are sent and received similarly. The solution shows how the sequence diagram is executed after a path of length $k = 4$. The encoding counts up to $k' = 12$ in order to accommodate the sequence diagram of length 3 and an upper bound of three additional empty messages before each message. Since less empty messages are necessary in between the messages of the sequence diagram, at positions 8, 9, and 12, nothing changes. ♢

As in Section 4.3, we present the encoding of the problem as a conjunction of different formulas.

We define the following functions to enhance the presentation of the formulas. Let $L = (l, M)$ be a lifeline instantiating the state machine $M = (S, \iota, A^{tr}, A^{\mathit{eff}}, T)$, and let $(s, \mathit{trg}, \epsilon, s_t^*)$ and $(s_t^*, \epsilon, \mathit{eff}, s')$ be transitions of the extended state machine M^* of M. These two transitions correspond to a transition $t = (s, \mathit{trg}, \mathit{eff}, s')$ of M. Further, let $N = (\sigma, a, \rho)$ be a message. Then $\mathsf{trans}(L) = T$, $\mathsf{src}(t) = s$, $\mathsf{int}(t) = s_t^*$, $\mathsf{trg}(t) = \mathit{trg}$, $\mathsf{eff}(t) = \mathit{eff}$, $\mathsf{tgt}(t) = s'$, $\mathsf{snd}(N) = \sigma$, and $\mathsf{symb}(N) = a$.

The first formula (Init) initializes the path by setting the global initial state at position 0 to true, and all other states and all symbols to false. This means that at position 0, all state machines are in their initial state and no symbol is waiting to be consumed. This formula is different to formula (Init) in Section 4.3 in that at position 0 the state machines are in the global initial state other than in some given global state.

$$\bigwedge_{j=1}^{l} \left(\iota_j^0 \wedge \bigwedge_{s \in S_j \cup \mathcal{S}_j^*, s \neq \iota_j} \overline{s}^0 \right) \wedge \bigwedge_{a \in A} \overline{a}^0 \tag{Init}$$

For a position $i > 0$, a variable from va in its positive polarity means that the respective symbol has been made available as effect through a transaction at $j \in [1..i]$ and is waiting to be consumed by some transition as a trigger at a position greater than i. At the position where the symbol is consumed, the symbol variable occurs in its negative polarity.

Formulas (4.4.1) to (4.4.8) are similar to formulas (4.3.1) to (4.3.8). They are different in that they count up to k' rather than up to k in order to accommodate the sequence diagram.

First, formula (4.4.1) ensures that whenever a transition is executed at some position i, then the state machine changes from the transition's source state at position i to its target state at position $i + 1$, the trigger symbol is set its negative polarity and the effect symbols are set to their positive polarity. It corresponds to the formula (4.3.1) in Section 4.3. We use the expression $\mathsf{trg}(t) \neq \epsilon$ for the presentation of the formula. It is replaced by the logical constants $\top$ and $\bot$ during the generation of the formula.

$$\bigwedge_{i=0}^{k'-1} \bigwedge_{t \in \mathcal{T}} \Bigg[t^i \rightarrow \Bigg(\mathsf{src}(t)^i \wedge \mathsf{int}(t)^{i+1} \wedge \left(\mathsf{trg}(t) \neq \epsilon \rightarrow \left(\mathsf{trg}(t)^i \wedge \overline{\mathsf{trg}(t)}^{i+1} \right) \right) \wedge \bigwedge_{\mathit{eff} \in \mathsf{eff}(t)} \left(\overline{\mathit{eff}}^i \wedge \mathit{eff}^{i+1} \right) \Bigg) \Bigg] \tag{4.4.1}$$

The two formulas (4.4.2) and (4.4.3) manage the polarity of the effect symbols and correspond to formulas (4.3.2) and (4.3.3) in Section 4.3. Formula (4.4.2) ensures that if a state machine does not leave its intermediate state, then the effect symbols remain positive and formula (4.4.3) ensures that if it leaves its intermediate state, then all effect symbols are set to false at the following position.

$$\bigwedge_{i=0}^{k'-1} \bigwedge_{\substack{t\in\mathcal{T}\\ \mathsf{eff}(t)\neq\emptyset}} \left[\mathsf{int}(t)^i \wedge \mathsf{int}(t)^{i+1} \rightarrow \bigwedge_{\mathit{eff}\in\mathsf{eff}(t)} \mathit{eff}^{i+1}\right] \tag{4.4.2}$$

$$\bigwedge_{i=0}^{k'-1} \bigwedge_{\substack{t\in\mathcal{T}\\ \mathsf{eff}(t)\neq\emptyset}} \left[\mathsf{int}(t)^i \wedge \overline{\mathsf{int}(t)}^{i+1} \rightarrow \bigwedge_{\mathit{eff}\in\mathsf{eff}(t)} \overline{\mathit{eff}}^{i+1}\right] \tag{4.4.3}$$

Formula (4.4.4) ensures that if the state machine is in an intermediate state at position i and all effects have been consumed at position $i+1$, then at position $i+1$ the state machine leaves the intermediate state and changes into the target state of the transition. It corresponds to formula (4.3.4) in Section 4.3.

$$\bigwedge_{i=0}^{k'-1} \bigwedge_{t\in\mathcal{T}} \left[\left(\mathsf{int}(t)^i \wedge \bigwedge_{\mathit{eff}\in\mathsf{eff}(t)} \overline{\mathit{eff}}^{i+1}\right) \rightarrow \left(\overline{\mathsf{int}(t)}^{i+1} \wedge \mathsf{tgt}(t)^{i+1}\right)\right] \tag{4.4.4}$$

The three formulas (4.4.5), (4.4.6), and (4.4.7) are called *frame axioms*. They ensure that there is always a transition when changes of symbols (formulas (4.4.5) and (4.4.6)) or changes of states (formula (4.4.7)) occur. They correspond to formulas (4.3.5), (4.3.6), and (4.3.7) in Section 4.3.

$$\bigwedge_{i=0}^{k'-1} \bigwedge_{\mathit{trg}\in A} \left[\mathit{trg}^i \wedge \overline{\mathit{trg}}^{i+1} \rightarrow \left(\left(\bigvee_{\substack{t\in\mathcal{T},\\ \mathsf{trg}(t)=\mathit{trg}}} t^i\right) \wedge \bigwedge_{\substack{t_1,t_2\in\mathcal{T},\\ \mathsf{trg}(t_1)=\mathsf{trg}(t_2)=\mathit{trg}}} \left(\overline{t_1}^i \vee \overline{t_2}^i\right)\right)\right] \tag{4.4.5}$$

$$\bigwedge_{i=0}^{k'-1} \bigwedge_{\mathit{eff}\in A} \left[\overline{\mathit{eff}}^i \wedge \mathit{eff}^{i+1} \rightarrow \left(\left(\bigvee_{\substack{t\in\mathcal{T},\\ \mathit{eff}=\mathsf{eff}(t)}} t^i\right) \wedge \bigwedge_{\substack{t_1,t_2\in\mathcal{T},\\ \mathsf{eff}(t_1)=\mathsf{eff}(t_2)=\mathit{eff}}} \left(\overline{t_1}^i \vee \overline{t_2}^i\right)\right)\right] \tag{4.4.6}$$

$$\bigwedge_{i=0}^{k'-1} \bigwedge_{s\in S} \left[s^i \wedge \overline{s}^{i+1} \rightarrow \bigvee_{t\in\mathcal{T},s=\mathsf{src}(t)} t^i\right] \tag{4.4.7}$$

Formula (4.4.8) expresses that each state machine is in exactly one state at each position. This formula corresponds to formula (4.3.8) in Section 4.3.

$$\bigwedge_{i=0}^{k'-1} \bigwedge_{j=1}^{l} \left[\left(\bigvee_{s\in(S_j\cup S_j^*)} s^i\right) \wedge \bigwedge_{s_1,s_2\in(S_j\cup S_j^*),s_1\neq s_2} \left(\overline{s_1}^i \vee \overline{s_2}^i\right)\right] \tag{4.4.8}$$

Finally, formula (Seq) encodes the sequence of messages to be executed after k steps. It sets the symbol of each message first to *true* and in the subsequent position to *false*, then it ensures the state changes of intermediate states, and finally, it ensures that no other message is received during the execution of the sequence diagram. Note that the frame axioms take care of the changes of the transition variables.

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

The formulas are converted to CNF (cf. Section 2.2), the input format of most SAT solvers, during the generation of the encoding. To this end, we apply the Tseitin transformation [114] where necessary.

In the same way as the encoding for the reachability problem of Section 4.3, the encoding of the k-SDCHECK problem allows that nothing happens, i.e., that no transaction takes place at some position. It is ensured by the frame axioms that in this case, the global state remains the same. This relaxation implicitly encodes the "at most k" steps formulation. If at n positions nothing happens and the execution of the message sequence starts at position k for $k \geq n$, it means that the length of the transaction sequence executed before the message sequence of the sequence diagram is of length $k - n$.

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 (i.e., variables from the sets vs, vx, and vt) we obtain the path of at most k steps leading to the execution of the sequence diagram, and the state changes of the state machines 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.

In order to simplify the encoding, we assume that each symbol can be consumable only once at each position. 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 [110].

It can be verified that this encoding is of size polynomial with respect to the size of the instance of k-SDCHECK.

Observation 4.4.1. Given an instance $\mathcal{K}$ of the k-SDCHECK problem, its encoding as a conjunction of the formulas (Init), (Seq), and (4.4.1) to (4.4.8) is of size polynomial with respect to the size of $\mathcal{K}$ because any of the formulas (Init), (Seq), and (4.4.1) to (4.4.8) contains at most two nested iterations over sets contained in the input.

4.4.4 Computational Complexity

We show that the k-SDCHECK problem is NP-complete. Its membership in NP follows directly from Observation 4.4.1, i.e., the fact that the k-SDCHECK problem can be encoded in

polynomial time with respect to its input size into a formula in propositional logic.

Corollary 4.4.1. The k-SDCHECK problem is in NP.

The proof of NP-hardness of the k-SDCHECK problem is based on the proof of NP-hardness of the k-SMREACH problem (cf. Lemma 4.3.1). In a similar way as for the k-SMREACH problem, we reduce the 3X3SAT problem (cf. Definition 4.3.5) to the k-SDCHECK problem. Recall that the 3X3SAT problem is a variation of the 3-satisfiability problem [54, Appendix A9.1, L02] where each variable can occur at most three times and each literal can occur at most twice in a propositional CNF formula that contains at most three literals per clause. This problem has been shown to be NP-complete [99, Proposition 9.3].

In our reduction we construct two sets of state machines in a similar way as in the proof of Lemma 4.3.1 for the k-SMREACH problem. The set $\mathcal{M}_{vars}$ contains a state machine for each variable in $\mathcal{V}$, and the set $\mathcal{M}_{clauses}$ contains a state machine for each clause in $\mathcal{C}$. We additionally build a state machine M_D that is not contained in any of the two sets. This additional state machine is used to execute the sequence diagram.

The alphabet of effects of the state machines in $\mathcal{M}_{vars}$ and the alphabet of triggers of the state machines in $\mathcal{M}_{clauses}$ contain a different symbol for each occurrence of each variable in $\mathcal{C}$. The alphabet of triggers of each state machine in $\mathcal{M}_{clauses}$ contains an additional symbol that does not occur in any of the alphabets of the state machines in $\mathcal{M}_{vars}$. The alphabet of effects of M_D contains all the additional symbols from the alphabets of the state machines in $\mathcal{M}_{clauses}$.

Just as for the reduction to the k-SMREACH problem, each state machine M_v in $\mathcal{M}_{vars}$ contains one state and one transition for each occurrence of a variable v in $\mathcal{C}$, and an initial state. Such a state machine does not receive any symbol, it only sends symbols. Thus, all triggers on all its transitions are ϵ. The states are aligned along two branches of transitions. Transitions on one branch refer to the occurrences of v in clauses as positive literals and transitions on the second branch to the occurrences of v as negative literals. This way, a state machine in M_v cannot send symbols representing opposite literals of a variable in the same run.

Each state machine M_C for clause C in $\mathcal{M}_{clauses}$ contains two states, an initial state and a sink state, and for each literal contained in C, $\mathcal{M}_{clauses}$ contains one transition connecting the two states and one transition looping the sink state. Other than in the reduction to the k-SMREACH problem, M_C contains one additional transition looping the sink state. State machines in $\mathcal{M}_{clauses}$ only receive, but never send symbols. Their transitions contain a trigger other than ϵ and the empty set as effects. Each trigger on a transition connecting the two states and on a transition looping the sink state in M_C corresponds to a literal in C and vice versa. The additional transition looping the sink state is triggered by the additional symbol.

The additional state machine M_D contains only one state and for each clause in $\mathcal{C}$ one transition. Each transition is triggered by ϵ and carries as effect one of the additional symbols of the state machines in $\mathcal{M}_{clauses}$ such that each of the additional symbol occurs on exactly one transition as effect.

We further build a sequence diagram that instantiates each state machine once, i.e., it contains one lifeline for each state machine. The message sequence of the sequence diagram contains exactly one message for each clause in $\mathcal{C}$ sent from the lifeline instantiating M_D and received by the respective lifeline instantiating a state machine M_C in $\mathcal{M}_{clauses}$, and carrying as symbol the additional symbol of M_C. The order of the messages in the sequence can be arbitrary.

Name	Clause
C_1	$\{x, \overline{y}\}$
C_2	$\{x, y\}$
C_3	$\{z\}$
C_4	$\{\overline{x}, \overline{y}, \overline{z}\}$

Table 4.4.2: A satisfiable instance $\mathcal{S}$ of 3X3SAT.

With this construction, each state machine in $\mathcal{M}_{vars}$ sends symbols representing literals that evaluate to *true* under one of the interpretations *true* or *false* of its variable to state machines in $\mathcal{M}_{clauses}$. Since there is no link between the two branches of a state machine in $\mathcal{M}_{vars}$, only one branch can be contained in a path and hence only one interpretation be constructed for the clauses in $\mathcal{C}$. When a state machine M_C in $\mathcal{M}_{clauses}$ reaches its sink state, then the clause C is satisfied. The sequence diagram can be executed if and only if all state machines in $\mathcal{M}_{clauses}$ reach their sink states. When they do so, then the propositional formula is satisfiable because all its clauses are satisfied. If at least one state machine cannot reach their sink state, then the propositional formula is unsatisfiable. The loops on the sink states prevent the state machines in $\mathcal{M}_{vars}$ from getting stuck in a state when the sink state of some state machine in $\mathcal{M}_{clauses}$ has already been reached by receiving a symbol from a different state machine in $\mathcal{M}_{vars}$.

We set k to $4|\mathcal{V}|$ for the following reasons. A path along one branch of a state machine in $\mathcal{M}_{vars}$ can contain at most two transitions (recall that the 3X3SAT problem allows a literal to occur at most twice) and the conversion into an extended state machine doubles the number of transitions. Further, all variables may be necessary to satisfy the formula and therefore, all state machines in $\mathcal{M}_{vars}$ may be required to reach their sink state. Also, in the worst case, one transaction can only contain one message, requiring one transaction for each transition in the extended state machines for $\mathcal{M}_{vars}$.

The following example illustrates the reduction.

Example 4.4.2. Table 4.4.2 shows a positive instance $\mathcal{S} = (\mathcal{V}, \mathcal{C})$ of 3X3SAT with four clauses and three variables. Figure 4.4.4 depicts the set $\mathcal{M}$ of state machines and Figure 4.4.5 depicts the sequence diagram of the k-SDCHECK instance $\mathcal{K} = (\mathcal{M}, D, k)$ reduced from $\mathcal{S}$. The set $\mathcal{M}_{vars}$ contains three state machines, one for each variable, and the set $\mathcal{M}_{clauses}$ contains four state machines, one for each clause. The sequence diagram contains a sequence of four messages, one for each clause. Each message is sent from lifeline m_d and each of the lifelines m_1 to m_4 receives the message containing the symbol that loops the sink state of its state machine and is unique within its transactions. The parameter k is set to $4|\mathcal{M}_{vars}| = 12$.

The instance $\mathcal{S}$ is satisfied by the assignment $x \mapsto$ *true*, $y \mapsto$ *false*, $z \mapsto$ *true*. Under this assignment, the occurrences of x evaluate to *true* in the clause C_1 and the clause C_2, and to *false* in the clause C_4, therefore satisfying C_1 and C_2. This evaluation corresponds to the upper branch of the state machine M_x, which sends the symbols c_1^x and c_2^x leading state machines M_1 and M_2 to their sink states. Similarly, the literal $\overline{y}$ evaluates to *true* in clause C_1 and in clause C_4, and the literal y to *false* in clause C_2. The two evaluations of $\overline{y}$ to *true* correspond to the lower branch of the state machine M_y, leading state machine M_4 to its sink state and looping the sink state of M_1 (which reached its sink state earlier by the evaluation of x). The evaluation of z leads

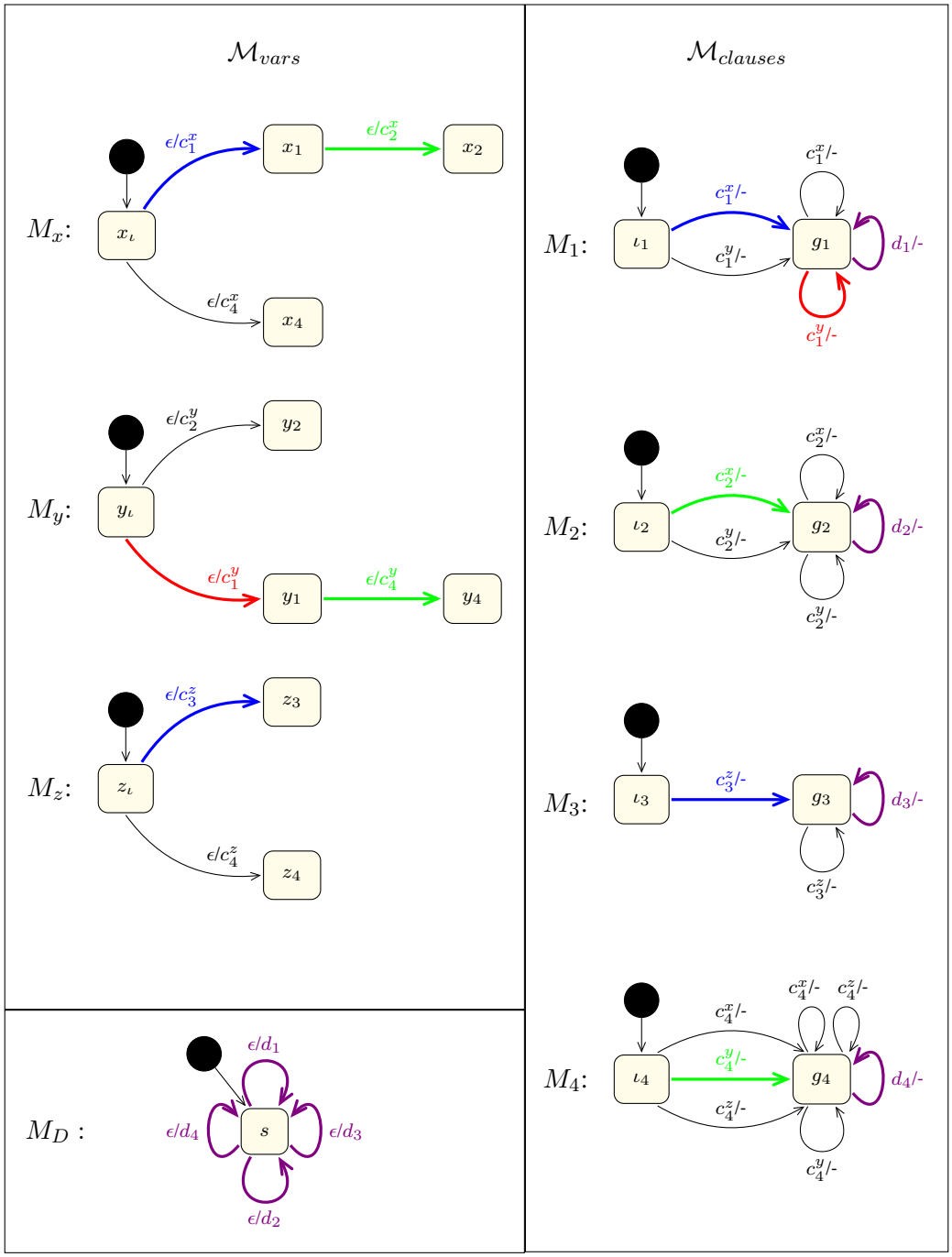

Figure 4.4.4: State machines reduced from the 3X3SAT instance in Table 4.4.2. Highlighted transitions represent a path leading from the global initial state to the execution of the sequence diagram. The transitions highlighted in blue belong to the first transaction, those highlighted in red to the second transaction, those in green to the third transaction, and those in purple to the execution of the sequence diagram.

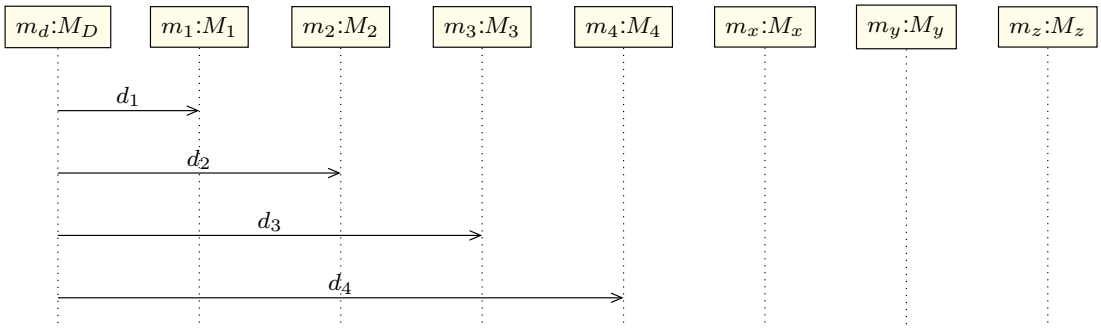

Figure 4.4.5: Sequence diagram reduced from the 3X3SAT instance in Table 4.4.2.

M_3 to its sink state in a similar way. Since all state machines in $\mathcal{M}_{clauses}$ are in their sink states, the sequence diagram can be executed. Therefore, $\mathcal{K}$ is a positive instance of k-SDCHECK.

This example shows the role of the looping transitions on the sink state of the state machines in $\mathcal{M}_{clauses}$. Without the looping transition receiving c_1^y in M_1, M_y could not send the symbol c_4^y to M_4 and therefore, M_4 would not reach its sink state. The communication of the symbols can be arranged in the path

$$
\begin{aligned}
&[\{(m_x, c_1^x, m_1), (m_z, c_3^z, m_3)\},\\
&\{(m_y, c_1^y, m_1)\},\\
&\{(m_x, c_2^x, m_2), (m_y, c_4^y, m_4)\},\\
&\{(m_d, d_1, m_1)\},\\
&\{(m_d, d_2, m_2)\},\\
&\{(m_d, d_3, m_3)\},\\
&\{(m_d, d_4, m_4)\}]
\end{aligned}
$$

of length 7, where the last four transactions contain messages of the sequence diagram (cf. Section 3.3 Definition 3.3.10, some brackets are omitted as the multimessages are single messages). ◊

Lemma 4.4.1. The k-SDCHECK problem is NP-hard.

Proof. Given an instance $\mathcal{S} = (\mathcal{V}, \mathcal{C})$ of the 3X3SAT problem, we construct an instance $\mathcal{K} = (\mathcal{M}, D, k)$ of the k-SDCHECK problem as follows.

$\mathcal{M}$ is the union of the sets $\mathcal{M}_{vars}$, $\mathcal{M}_{clauses}$, and $\{M_D\}$ of state machines, which are constructed as follows. For each variable v in $\mathcal{V}$, let $\mathcal{C}_v = \{C \mid v \in \mathsf{vars}(C)\}$ be the set of clauses containing a literal of v. Then the set $\mathcal{M}_{vars}$ contains for each variable v in $\mathcal{V}$ a state machine $M_v = (S_v, \iota_v, A_v^{tr}, A_v^{\mathit{eff}}, T_v)$ with

- $S_v = \{v_\iota\} \cup \{v_C \mid C \in \mathcal{C}_v\}$,
- $\iota_v = v_\iota$,

- $A_v^{tr} = \{\epsilon\}$,
- $A_v^{\mathit{eff}} = \{c_C^v \mid C \in \mathcal{C}_v\}$, and
- $T_v = \begin{cases} \{(v_\iota, \epsilon, \{c_A^v\}, v_A), (v_A, \epsilon, \{c_B^v\}, v_B), (v_\iota, \epsilon, \{c_C^v\}, v_C) \\ \quad \mid \ell \in A, \ell \in B, \overline{\ell} \in C, \mathsf{var}(\ell) = v, A, B, C \in \mathcal{C}_v\} & \text{if } |\mathcal{C}_v| = 3 \\ \{(v_\iota, \epsilon, \{c_A^v\}, v_A), (v_A, \epsilon, \{c_B^v\}, v_B) \\ \quad \mid \ell \in A, \ell \in B, \mathsf{var}(\ell) = v, A, B \in \mathcal{C}_v\} \ \cup \\ \quad \{(v_\iota, \epsilon, \{c_A^v\}, v_A), (v_\iota, \epsilon, \{c_B^v\}, v_B) \\ \quad\quad \mid \ell \in A, \overline{\ell} \in B, \mathsf{var}(\ell) = v, A, B \in \mathcal{C}_v\} & \text{if } |\mathcal{C}_v| = 2 \\ \{(v_\iota, \epsilon, \{c_A^v\}, v_A) \mid \ell \in A, \mathsf{var}(\ell) = v, A \in \mathcal{C}_v\} & \text{if } |\mathcal{C}_v| = 1. \end{cases}$

The set $\mathcal{M}_{clauses}$ contains a state machine $M_C = (S_C, \iota_C, A_C^{tr}, A_C^{\mathit{eff}}, T_C)$ for each clause C in $\mathcal{C}$ with

- $S_C = \{\iota_C, g_C\}$,
- $\iota_C = \iota_C$,
- $A_C^{tr} = \{c_C^v \mid v \in \mathsf{vars}(C)\} \cup \{d_C\}$,
- $A_C^{\mathit{eff}} = \emptyset$, and
- $T_C = \{(\iota_C, c_C^v, \emptyset, g_C), (g_C, c_C^v, \emptyset, g_C) \mid v \in \mathsf{vars}(C)\} \cup \{(g_C, d_C, \emptyset, g_C)\}$.

$M_D = (S, \iota, A^{tr}, A^{\mathit{eff}}, T)$ is a state machine with $S = \{s\}$, $\iota = s$, $A^{tr} = \emptyset$, $A^{\mathit{eff}} = \{d_C \mid C \in \mathcal{C}\}$, and $T = \{(s, \epsilon, d_C, s) \mid C \in \mathcal{C}\}$. Further, $D = (\mathcal{L}, \mathcal{N})$ is a sequence diagram over $\mathcal{M}$ where $\mathcal{L} = \{(m_v, M_v) \mid v \in \mathcal{V}\} \cup \{(m_C, M_C) \mid C \in \mathcal{C}\} \cup \{(m_d, M_D)\}$ and $\mathcal{N}$ is an arbitrary sequence containing each message of the set $\{(m_d, d_C, m_C) \mid C \in \mathcal{C}\}$ of messages exactly once. We set $k = 4|\mathcal{M}_{vars}|$.

The size of the reduced instance $\mathcal{K}$ of k-SDCHECK is linear with respect to the size of the instance $\mathcal{S}$ of 3X3SAT. In particular, it contains for each variable one state machine with at most four states and at most three transitions, for each clause one state machine with two states and at most seven transitions, and one additional state machine with one state and as many transitions as clauses. We proceed with showing that $\mathcal{S}$ is a positive instance of 3X3SAT if and only if $\mathcal{K}$ is a positive instance of k-SDCHECK.

($\Rightarrow$) If $\mathcal{S}$ is a positive instance of 3X3SAT, then there exists an interpretation σ for each variable in $\mathcal{V}$ under which the formula evaluates to *true*. This means that in each clause, at least one literal evaluates to *true* (cf. Section 2.2). By construction, a state machine M_v in $\mathcal{M}_{vars}$ can send one of two sequences of symbols representing the clauses satisfied due to an assignment of v to *true* or *false* respectively, or one sequence if the literal only occurs in one polarity in the formula. The state machines in $\mathcal{M}_{clauses}$ can receive the symbols representing each clause's literals, where they lead the state machine's initial state to the sink state or make the state machine stay in the sink state. It is always possible to execute all transitions along

a branch of a state machine in $\mathcal{M}_{vars}$ because the symbols can always be received by some state machine in $\mathcal{M}_{clauses}$, either by a transition between source and sink state or by a transition looping the sink state. A branch of a state machine in $\mathcal{M}_{vars}$ contains at most two transitions, which correspond to four transitions in the extended state machine. Assuming the case where each transaction in the path contains only one message, we need $k = 4|\mathcal{M}_{vars}|$ transactions to reach the sink states of each state machine in $\mathcal{M}_{clauses}$. Since under σ, at least one literal of each clause evaluates to *true* and exactly one state machine in $\mathcal{M}_{clauses}$ corresponds to one clause, all state machines in $\mathcal{M}_{clauses}$ reach their sink state. From a global state containing the sink states of all state machines in $\mathcal{M}_{clauses}$ and the state s of M_D, the sequence diagram D can be executed. This makes $\mathcal{K}$ a positive instance of k-SDCHECK.

($\Leftarrow$) If $\mathcal{K}$ is a positive instance of k-SDCHECK, then a path exists such that a partial global state containing all sink states of the state machines in $\mathcal{M}_{clauses}$ can be reached from the global initial state. Only from such a state the sequence diagram can be executed. Such a path contains for each state machine M_C in $\mathcal{M}_{clauses}$ a transaction containing one of the symbols occurring as trigger on one of the transitions between the initial state and the sink state of M_C. The senders of these symbols are state machines in $\mathcal{M}_{vars}$, each along one of its branches. Since each branch corresponds to an assignment to a variable in $\mathcal{S}$ and each state machine in $\mathcal{M}_{clauses}$ corresponds to a clause in $\mathcal{S}$, an assignment satisfying the clauses in $\mathcal{S}$ can be retrieved from the branches. Note that, since exactly one branch of each state machine in $\mathcal{M}_{vars}$ is contained in the path, it is ensured that a variable is assigned exactly one truth value. Therefore, $\mathcal{S}$ is a positive instance of 3X3SAT. □

Theorem 4.4.1. The k-SDCHECK problem is NP-complete.

Proof. The theorem follows from Corollary 4.4.1 and Lemma 4.4.1 □

Chapter 5

Evaluation

We implemented our solving methods for the k-SMREACH problem (cf. Section 4.3), the k-SDCHECK problem (cf. Section 4.4), and the SDMERGE problem (cf. Section 4.2) in the three tools Global State Checker, Sequence Diagram Checker, and Sequence Diagram Merger respectively. All tools integrate an off-the-shelf SAT solver. We tested the implementations with respect to their scalability and correctness.

All prototype implementations are available as Eclipse plugins on our project website.[1] They are built into one coherent framework and adhere to Ecore metamodels of the Eclipse Modeling Framework (EMF) that implement the metamodels described in Section 3.1.

To test our approaches, representative benchmark sets were required, but no existing benchmarks fulfilled our requirements. Benchmark sets as presented by Brosch et al. [25] contain only modeling scenarios of a single view and focus on class diagrams. We therefore established our own benchmark sets. First, we handcrafted three sets of intuitive and small instances, and second, we developed an approach to generate random instances of models defined by an Ecore metamodel in order to apply grammar-based white-box fuzzing. Although the models of our benchmark set are formulated in *tMVML*, they can be reused in other case studies because they are realized as Ecore models. Hence, a translation to other modeling languages like the UML can be achieved by the means of model transformations. All benchmarks and their solutions are also available on our project website.[2]

In this chapter, we first give an overview of the implementation of our framework and then describe the crafted instances and the generation of random instances. Each problem's evaluation is then discussed in a separate section. First, we describe the evaluations of the Global State Checker and the Sequence Diagram Checker on both crafted instances and random instances and then we describe the evaluation of the Sequence Diagram Merger on crafted instances.

[1] `http://modelevolution.org/prototypes`
[2] `http://modelevolution.org/media/eval-model-verification`

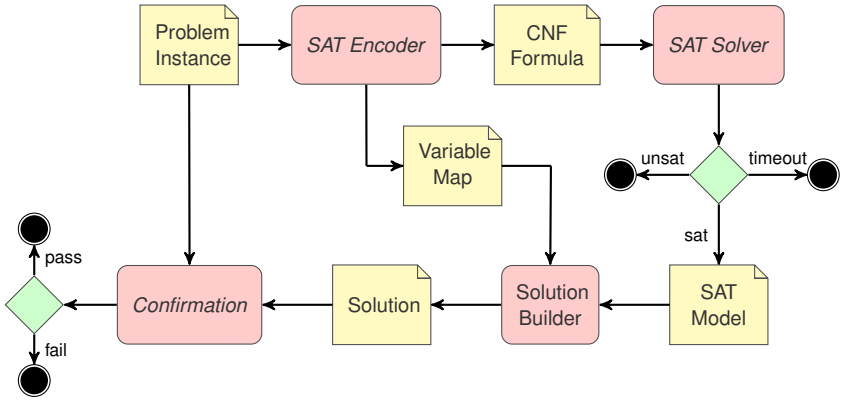

Figure 5.1.1: Workflow of a test run for an instance of one of our problems.

5.1 Implementation and Testing Environment

The implementations of our approaches to the three previously discussed verification problems are embedded into the Eclipse Modeling Framework (EMF), the core technology of the Eclipse Modeling Project.[3,4] Eclipse is a popular integrated development environment for many programming languages, in particular Java. EMF provides a metamodel (Ecore) to define the syntax of multi-view software models (cf. Section 2.3), an editing framework to create and modify the models, and a code generation facility that allows to retrieve Java code from the models. Ecore also allows to define metamodels other than the Ecore metamodel itself.

We provide instances of our problems as models that instantiate an Ecore metamodel. This metamodel implements the event-based metamodel (Figure 3.1.1) for instances of the SD-MERGE problem and the alternative metamodel (Figure 3.1.3) for instances of the k-SMREACH problem and the k-SDCHECK problem.

Figure 5.1.1 depicts the general workflow of a test run for an instance of one of the problems. First, the *SAT Encoder* translates the problem instance into a propositional formula according to its respective encoding. Measures, such as the introduction of Tseitin variables [114], are taken to convert the formula into conjunctive normal form (CNF) (cf. Section 2.2) without exponential blowup. The data structure representing the encoding is then handed to a *SAT Solver* and a *Variable Map* is created that assigns the encoding's Boolean variables to statements of the encoded problem (cf. descriptions of encodings in Sections 4.2, 4.3, and 4.4). All our tools are implemented in Java. As SAT solvers we employ Picosat [14] for the SDMERGE problem and SAT4J [85] for the k-SMREACH problem and the k-SDCHECK problem.

The SAT solver returns `UNSAT`, `SAT`, or it times out if the instance is too hard and a timeout is set. If it returns `UNSAT`, it means that the problem has no solution. If it returns `SAT`, it means that a solution exists. In this case, the SAT solver also returns a logical model for the formula that encodes the problem. This model contains the set of variables that are set to *true* in order to evaluate the formula to *true* (cf. Section 2.2). Using the Variable Map, these variables are

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

	Coffee	Mail	Philosophers
Number of state machines	3	3	6
Total number of states	8	15	15
Total number of transitions	18	23	21
Size alphabet	10	13	12

Table 5.2.1: Sizes of the crafted models.

then mapped back into statements regarding the respective problem by the *Solution Builder*. The mapping of all these variables represents a solution of the problem. We finally confirm that the retrieved solution indeed solves the initial problem instance by using a simulation tool that can check the (trigger) consistency of a software model by stepping through a sequence of transactions. This simulation should always pass. If it fails, then the tool has an error which must be corrected. In all test runs described in the following sections the simulation passed.

We executed all experiments on a computer with an Intel Core i5-540M CPU with 2.53GHz and 8GB of RAM.

5.2 Crafted Benchmark Set

We designed three models containing different numbers of state machines, states, and transitions in order to test our prototype implementations of the three problems on intuitive examples. Based on these models, we retrieved problem-specific instances for the k-SMREACH problem, the k-SDCHECK problem, and the SDMERGE problem. These three models are

- a model about a coffee machine ("Coffee") similar to our running example in Sections 4.3 and 4.4, as depicted in Figure 5.2.1,
- a simplified variant of the SMTP protocol ("Mail") similar to our running example in Section 4.2, as depicted in Figure 5.2.2, and
- a variant of the well-known dining philosopher problem with three philosophers ("Philosophers"), inspired by the running example in the work of Varró [120], as depicted in Figure 5.2.3.

Table 5.2.1 gives an overview of the sizes of the instances.

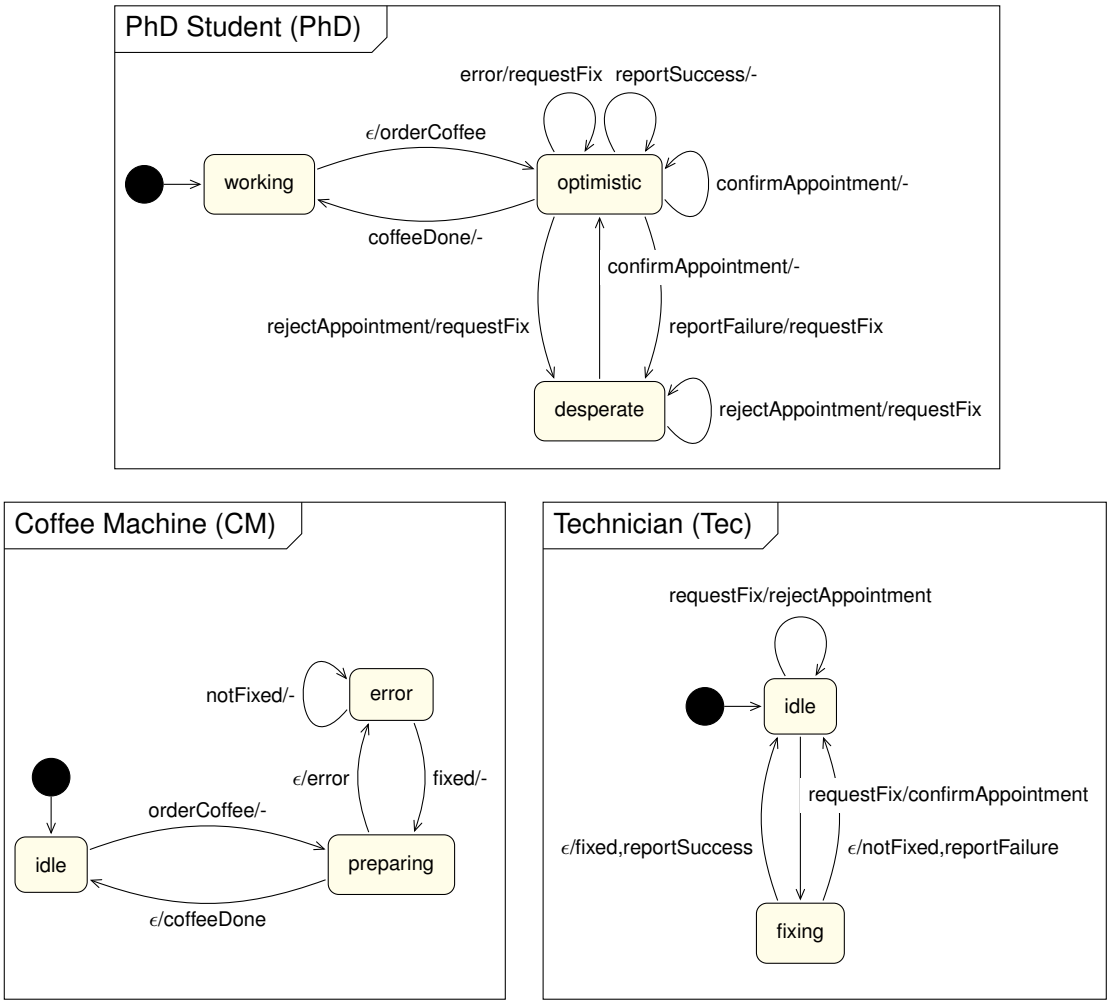

Figure 5.2.1: The crafted model “Coffee”.

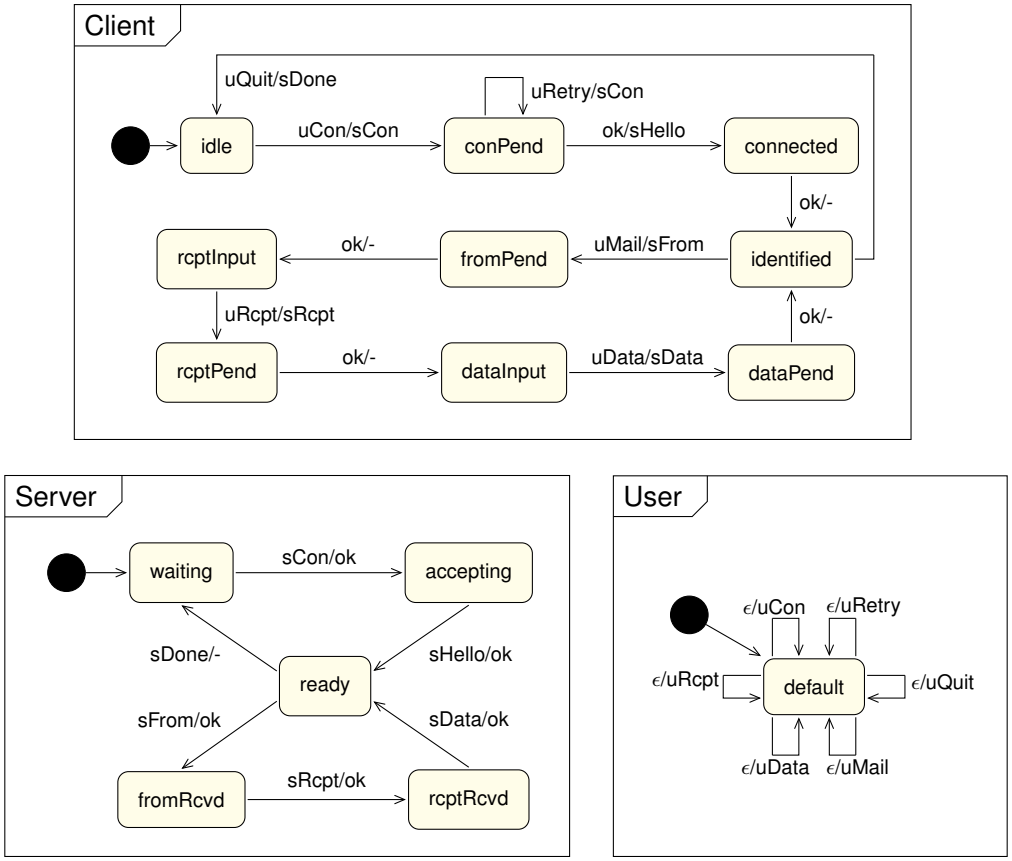

Figure 5.2.2: The crafted model “Mail”.

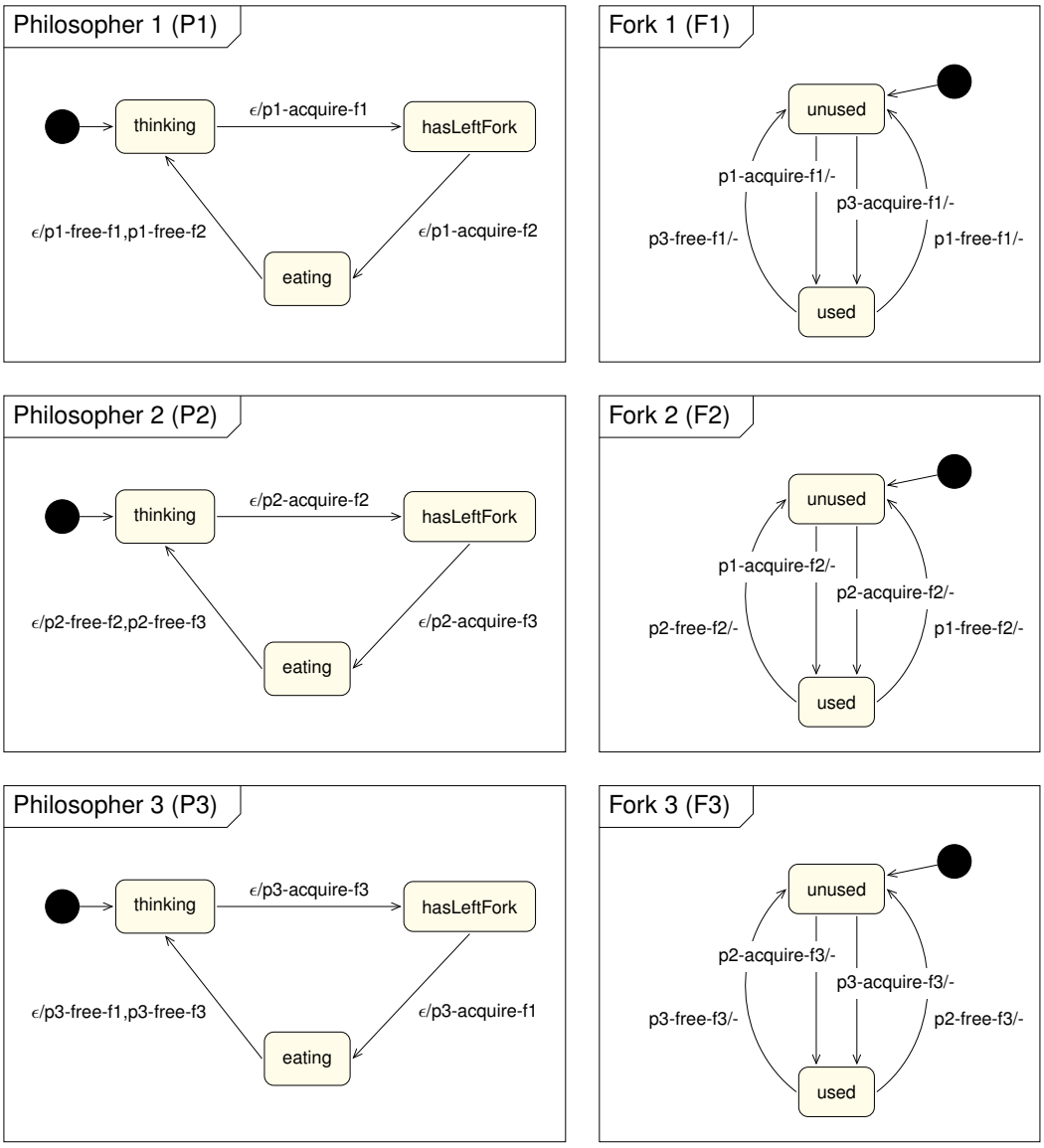

Figure 5.2.3: The crafted model “Philosophers”.

5.3 Random Model Generation

We generated instances of the k-SMREACH problem and the k-SDCHECK problem at random in order to apply white-box fuzzing for debugging and performance evaluation of our implementations. To this end, we developed a random model generator based on the tool we presented in previous work [123], which we adapted to the two problems. Our model generation tool consists of two components, a *generator* to build syntactically correct diagrams, and a *simulator* to ensure that a message sequence is indeed consistent with the state machines.

The generator takes as input a set of parameters regarding the size of the instances, such as the number of state machines, the number of states, the number of transitions, or the size of the alphabet. These parameters strongly influence whether the instance will be positive or negative. For example, for a set of state machines that have a high number of states with few transitions and many different triggers and effects, it is more difficult to find a path to some global state than it is for a set of state machines with few states, many transitions, and a small alphabet. We fine-tuned the parameter values manually in order to obtain a reasonable probability to generate a positive or a negative instance, respectively.

In the following, we first describe the random generation of state machines, which is used for evaluating both the k-SMREACH problem and the k-SDCHECK problem. Then we describe the random generation of a partial global state, which we use to create instances of the k-SMREACH problem, and the random generation of (in)consistent sequence diagrams, which we use to create instances of the k-SDCHECK problem.

5.3.1 Generation of State Machines

We build a set of state machines based on the following parameters.

- nrStatemachines: An integer greater than 0 defining the number of state machines.
- minNrStates and maxNrStates with minNrStates $\leq$ maxNrStates: Two integers greater than 0 defining bounds on the number of states per state machine. The actual number is chosen uniformly at random between and including these bounds for each state machine.
- minNrTrans and maxNrTrans with minNrTrans $\leq$ maxNrTrans: Two integers greater than 0 defining bounds on the number of transitions per state machine. The actual number is chosen uniformly at random between and including these bounds for each state machine. In order to avoid isolated states, these numbers should depend on minNrStates and maxNrStates.
- nrSymbols: An integer greater than 0 defining the size of the alphabet the state machines are defined over.
- probTrigger: A rational between 0 and 1 (inclusively) defining the probability of a transition to contain a trigger symbol other than ϵ.
- probEff: A rational between 0 and 1 (inclusively) defining the probability of a transition to contain an effect symbol.

For each state machine, our algorithm chooses uniformly at random a number of states and transitions in between the bounds minNrStates and maxNrStates, respectively 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 from the initial state the trigger ϵ is added in order to allow some communication sequence to start. To all other transitions, a trigger other than ϵ is added with probability probTrigger. To each transition containing ϵ as trigger an effect is added, and to all other transitions one effect is added with probability probEff and the empty set is added with probability $1 - \mathsf{probEff}$. 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.

5.3.2 Generation of Goal States

We build random instances of the k-SMREACH problem by randomly choosing a goal state based on a previously generated set of state machines. The additional parameter relGoalSize, a rational between 0 and 1 (inclusively), defines the size of the goal state relative to the number of state machines. We set the number of states in the goal state to relGoalSize times the number of state machines rounded up to the next integer.

5.3.3 Generation of Sequence Diagrams

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

- nrLifelines: An integer greater than 0 defining the number of lifelines to be contained in the sequence diagram.
- nrMessages: An integer greater than 0 defining the number of messages to be contained in the sequence diagram.
- probInconsistency: A rational between 0 and 1 (inclusively) defining the probability for the generator to insert a random message in order to make the model inconsistent.

For each lifeline, a state machine is chosen uniformly at random from the state machine view. If $\mathsf{nrLifelines} > \mathsf{nrStatemachines}$ then it is ensured that each state machine is instantiated at least once and otherwise it is ensured that no state machine is instantiated more than once. In order to ensure consistency, the model 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 with respect to 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. This is repeated $k + \mathsf{nrMessages}$ times in order to generate a sequence diagram that can be executed after at most

k steps. It can also happen that the sequence diagram can be executed after less than k steps after a different path than the one followed or even on the same path at an earlier position. 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, the generator inserts with probability probInconsistency one random message at a random position. To create a random message a sender, a receiver, and a symbol are chosen uniformly at random among the set of lifelines respectively the alphabet. After inserting this random message the current global state is reset to the global initial state. This procedure, however, still can result in a satisfiable instance. For this reason, the value for probInconsistency often has to be rather high in order to achieve a reasonable share of unsatisfiable instances.

The parameter values for the state machines influence to a great extent the difficulty of creating a consistent sequence diagram and it can easily happen that no or only a small message sequence can be generated. 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.

5.4 Evaluation of the Global State Checker

Our prototype implementation and testing environment for our encoding of the k-SMREACH problem correspond to the workflow described in Figure 5.1.1 of Section 5.1. A problem instance consists of a set of state machines, a partial global state over these state machines, and a value for the bound k. The encoder module receives such a problem instance and produces a propositional formula in CNF according to the encoding described in Section 4.3. The SAT solver either returns `UNSAT`, `SAT`, or times out. If it returns `UNSAT`, it means that the problem has no solution, i.e., that the specified state is not reachable in k steps. If it returns `SAT`, it means that there exists a solution, i.e., a path of length at most k from the initial configuration to a global state matching the specified goal. In this case, the SAT solver additionally returns a logical model of the formula representing the problem, which the Solution Builder translates to a solution of the k-SMREACH instance using the Variable Map. Our simulation tool then executes the path on the set of state machines in order to confirm the correctness of the Global State Checker and to give feedback to the user.

Figure 5.4.1 shows the graphical user interface of our prototype. The user can select the goal state directly in the modeling editor, enter the bound k, and start the Global State Checker. For convenience, it may be specified whether the selected goal state is expected to be reachable or not. Red or green highlighting indicates how the expected result compares to the actual result. The console in Figure 5.4.1 lists some test cases of the coffee machine model described in Section 5.2. The expanded subitems of the third test case in Figure 5.4.1 show the path to the goal state found by the Global State Checker.

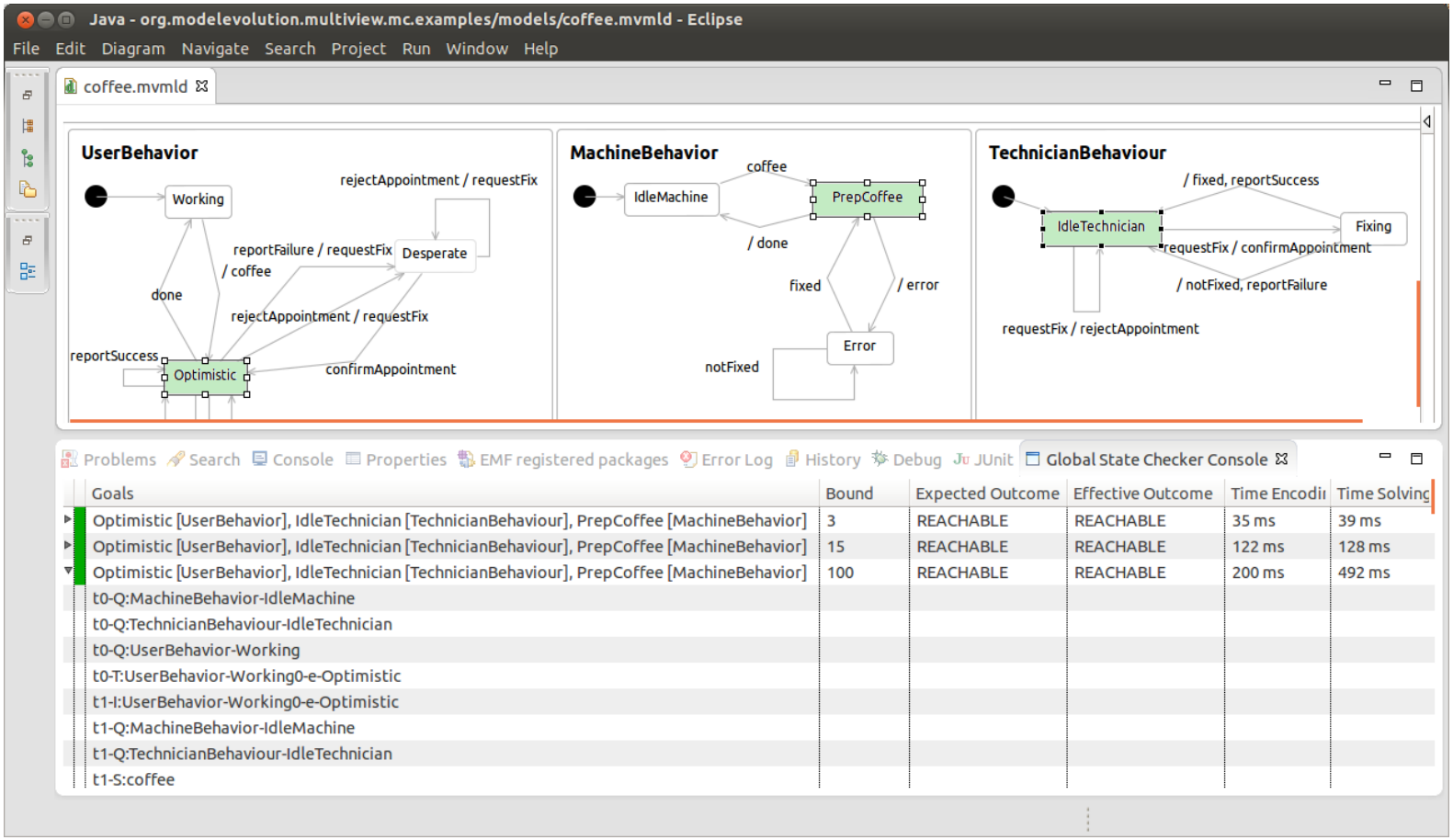

Figure 5.4.1: Graphical user interface of the Global State Checker.

5.4.1 Evaluation with Crafted Instances

We evaluated the Global State Checker using the set of crafted models described in Section 5.2. Based on these models, we built instances of the k-SMREACH problem using as goal states different combinations of states. For the models "Coffee" and "Mail" we built one instance for each possible goal state, and for the model "Philosophers", we selected 100 instances at random for goal states of size 1, 3, and 6. For each of the combinations for the goal states we tested instances for the values 3, 15, 100, and 500 for k. For instances of the model "Philosophers" we further set a timeout of 200 seconds as some of the instances with k set to 500 turned out to be very challenging for the SAT solver. For all instances, $\hat{s}$ (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.

Details on the outcomes of the test cases are presented in Tables 5.4.1, 5.4.2, and 5.4.3. As could be expected, the encoding times, solving times, numbers of clauses, and numbers of variables increase with an increasing value for k and the bottleneck for the overall runtimes is the task of solving rather than the task of encoding the instances with increasing size (number of clauses and number of variables) of the instances.

The solving times with respect to the size of the goal state show a different behavior for each model. With a higher value for the size of the goal state, the solving times decrease for satisfiable instances for the model "Coffee", they increase for satisfiable instances for the model "Mail", and they stay around the same values for the model "Philosophers".

The solving times for satisfiable instances are noticeably shorter than those for unsatisfiable instances. These runtimes depend on the SAT solver and can be different when employing a different solver or different heuristics. However, it is very interesting that for randomized instances (cf. next subsection) the opposite is the case despite of using the same solver.

The number of variables stays the same for different sizes of goal states because no new variables have to be introduced in order to describe a different goal. However, the numbers of clauses change slightly, which is due to the env function (cf. Section 4.3) that returns different numbers of states around the goal state depending on the transitions around the states contained in the goal state.

By "path length" we refer to the length of the found path from the initial global state to the goal state in satisfiable instances. The fact that it increases only slightly with increasing values for k indicates that lower values for k are sufficient to find a path for satisfiable instances.

The numbers of satisfiable instances increase with higher values for k, respectively the numbers of unsatisfiable instances decrease. This is the case because a goal state that can be reached by a path of length at most i can also be reached by a path with length at most j for $j > i$. The threshold for the choice of a value for k seemed to be at $3 < k \leq 15$ for all models. The value 15 as the upper bound for k coincides with the upper bound of states in the crafted models.

Size of goal state / number of state machines	1/3			
k	3	15	100	500
Number of instances SAT	5	8	8	8
Number of instances UNSAT	3	0	0	0
Average number of clauses	660	4,140	28,790	144,790
Average number of variables	200	848	5,438	27,038
Average path length SAT	0.4	4	17	23
Average encoding time	2	11	242	507
Average solving time SAT	<1	4	304	2,780
Average solving time UNSAT	<1	n/a	n/a	n/a

Size of goal state / number of state machines	2/3			
k	3	15	100	500
Number of instances SAT	6	11	11	11
Number of instances UNSAT	15	10	10	10
Average number of clauses	649	4,129	28,779	144,779
Average number of variables	200	848	5,438	27,038
Average path length SAT	0.5	4	12	19
Average encoding time	2	11	83	453
Average solving time SAT	<1	4	245	4,844
Average solving time UNSAT	<1	11	5,224	106,548

Size of goal state / number of state machines	3/3			
k	3	15	100	500
Number of instances SAT	2	3	3	3
Number of instances UNSAT	16	15	15	15
Average number of clauses	637	4,117	28,767	144,767
Average number of variables	200	848	5,438	27,038
Average path length SAT	0.5	3	18	26
Average encoding time	2	11	84	449
Average solving time SAT	<1	2	121	5,440
Average solving time UNSAT	<1	9	2,510	45,483

Table 5.4.1: Results of the evaluation of instances derived from the model "Coffee" for different sizes of the goal state and different values for k. All times are given in milliseconds.

Size of goal state / number of state machines	1/3			
k	3	15	100	500
Number of instances SAT	3	11	11	11
Number of instances UNSAT	12	4	4	3
Nr. instances timeout	0	0	0	1
Average number of clauses	926	7,382	53,112	268,312
Average number of variables	275	1,163	7,453	37,053
Average path length SAT	0.7	5	8	12
Average encoding time	4	17	176	654
Average solving time SAT	<1	6	183	2,493
Average solving time UNSAT	<1	15	1,583	31,707

Size of goal state / number of state machines	2/3			
k	3	15	100	500
Number of instances SAT	3	13	13	13
Number of instances UNSAT	56	46	46	44
Nr. instances timeout	0	0	0	2
Average number of clauses	881	7,337	53,067	268,267
Average number of variables	275	1,163	7,453	37,053
Average path length SAT	0.3	6	10	19
Average encoding time	3	17	125	639
Average solving time SAT	<1	9	164	5,390
Average solving time UNSAT	<1	15	1,356	38,844

Size of goal state / number of state machines	3/3			
k	3	15	100	500
Number of instances SAT	1	3	3	3
Number of instances UNSAT	44	42	42	41
Nr. instances timeout	0	0	0	1
Average number of clauses	854	7,310	53,040	268,240
Average number of variables	275	1,163	7,453	37,053
Average path length SAT	0	7	16	22
Average encoding time	3	16	122	640
Average solving time SAT	<1	11	660	8,214
Average solving time UNSAT	<1	13	1,265	35,445

Table 5.4.2: Results of the evaluation of instances derived from the model "Mail" for different sizes of the goal state and different values for k. All times are given in milliseconds.

Size of goal state / number of state machines	1/6			
k	3	15	100	500
Number of instances SAT	12	15	15	7
Number of instances UNSAT	3	0	0	0
Nr. instances timeout	0	0	0	8
Average number of clauses	621	3,861	26,811	134,811
Average number of variables	257	1,085	6,950	34,550
Average path length SAT	0.8	4	27	106
Average encoding time	1	12	86	452
Average solving time SAT	<1	6	503	116,543
Average solving time UNSAT	<1	n/a	n/a	n/a

Size of goal state / number of state machines	3/6			
k	3	15	100	500
Number of instances SAT	36	58	58	13
Number of instances UNSAT	64	42	42	2
Nr. instances timeout	0	0	0	85
Average number of clauses	611	3,851	26,801	134,801
Average number of variables	257	1,085	6,950	34,550
Average path length SAT	0.8	4	29	121
Average encoding time	2	12	86	447
Average solving time SAT	<1	6	1,396	170,540
Average solving time UNSAT	<1	42	26,646	30,472

Size of goal state / number of state machines	6/6			
k	3	15	100	500
Number of instances SAT	1	5	5	1
Number of instances UNSAT	98	95	95	79
Nr. instances timeout	0	0	0	20
Average number of clauses	605	3,845	26,795	134,795
Average number of variables	257	1,085	6,950	34,550
Average path length SAT	1	5	29	163
Average encoding time	2	11	81	436
Average solving time SAT	<1	6	1,235	177,892
Average solving time UNSAT	<1	8	4,541	2,434

Table 5.4.3: Results of the evaluation of instances derived from the model "Philosophers" for different sizes of the goal state and different values for k. All times are given in milliseconds.

	small	medium	large
nrStatemachines	4	12	30
minNrStates	2	4	8
maxNrStates	3	6	12
minNrTrans	6	16	40
maxNrTrans	9	24	60
probTrigger	0.5	0.3	0.3
probEff	0.5	0.3	0.3
nrSymbols	4	8	16
k	3	6	12

Table 5.4.4: Parameter values to generate random instances of the k-SMREACH problem.

5.4.2 Evaluation with Random Instances

We generated random instances of the k-SMREACH problem using our tool described in Section 5.3 and assigned them to three different groups according to their size.

An instance's size is determined by the values of the parameters discussed in Section 5.3 and by the bound k. Apart from its size, these values also determine the probability of a randomly generated instance to be positive or negative. We aimed at generating instance groups that differ clearly in their size and that contain enough positive and negative instances to draw some conclusions.

The parameters strongly influence each other. For example, a small number of states with a high number of transitions results in a rather dense state machine. A model containing dense state machines along with a small alphabet is likely to contain many goal states that are reachable as it facilitates a high number of communication scenarios. On the other side, a model with sparse state machines and a large alphabet are likely to contain many goal states that are unreachable. Also, a small value for k is likely to produce negative instances for models that contain state machines with many states, however, a high value for k may unnecessarily increase the size of the encoding for models that contain state machines with few states.

We executed some preliminary runs in order to fine-tune the parameter values. Table 5.4.4 shows the parameter values per group that we set for our test runs. As discussed above, the parameters strongly influence each other. These dependencies between the parameters are considered in our choice of their values. Parameters minNrTrans and maxNrTrans depend on the parameters minNrStates respectively maxNrStates in that their value is multiplied by 3 for the set "small", by 4 for the set "medium" and by 5 for the set "large". The values of nrSymbols are the values of minNrStates multiplied by 2. We set the value of k is to maxNrStates following the results of the evaluation of the crafted instances.

The size of the goal state is determined by relGoalSize. For this parameter we test the three values 0.25, 0.5, 0.75 and 1 for each category. It describes the size of the goal state relative to nrStatemachines (cf. Section 5.2). We further set a timeout of 300 seconds for the SAT solver.

Table 5.4.5 describes the results of our experiments over 1,000 randomly generated instances

in each category and for each value for relGoalSize.

We distinguish solving times for UNSAT and SAT instances. It can be seen that the average solving time for UNSAT instances is considerably lower than for SAT instances. This behavior is opposite to what we observed for crafted instances in the previous subsection. In many cases, the SAT solver detects trivial contradictions already when reading the input clauses, which explains the extremely low solving times for UNSAT instances. Further, the solving times scale 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. Together with the much lower solving times of UNSAT instances compared to SAT instances, this indicates that UNSAT instances are likely to contain a contradiction that is very easy to find by the SAT solver Sat4j.

The relative number of SAT instances decreases with increasing values for relGoalSize, which is an expected behavior since high values for relGoalSize result in more constraints. This can also be seen in the evaluation of the crafted instances (cf. Tables 5.4.1, 5.4.2, and 5.4.3). for this problem.

Also, the solving times increase with increasing values for relGoalSize in the SAT instances of sets "small" and "medium", as they did for the SAT instances of the crafted instances. For the SAT instances in the set "large" this is not the case. However, the average solving time for the instances with a fully specified goal state contains only one instance due to a high number of timeouts, so this number cannot be considered representative. The additional constraints resulting from higher values for relGoalSize seem to harden the search for the SAT solver in SAT instances. For the UNSAT instances we cannot observe such a behavior.

We further computed the length of the path to the goal state in the solution of positive instances. This path length remains unchanged between different sizes of the goal state. It increases with the size of the instance as could be expected. Note that this path length is not necessarily the minimal path length.

The number of clauses and the number of variables increase considerably with the sizes of the instances, however, up to the instances in the "large" set, this can still be handled reasonably by the SAT solver.

	small			
relGoalSize	0.25	0.5	0.75	1
Number of instances SAT	688	380	207	151
Number of instances UNSAT	312	620	793	849
Number of instances timeout	0	0	0	0
Average number of clauses	1,057	1,070	1,055	1,043
Average number of variables	270	271	270	269
Average path length SAT	0.6	0.7	0.8	0.9
Average encoding time	1	2	1	2
Average solving time SAT	<1	<1	<1	<1
Average solving time UNSAT	<1	<1	<1	<1

	medium			
relGoalSize	0.25	0.5	0.75	1
Number of instances SAT	571	251	89	15
Number of instances UNSAT	429	749	911	985
Number of instances timeout	0	0	0	0
Average number of clauses	49,408	48,997	49,137	49,212
Average number of variables	3,601	3,594	3,602	3,605
Average path length SAT	3	3	3	3
Average encoding time	80	82	82	82
Average solving time SAT	29	49	81	106
Average solving time UNSAT	9	15	15	13

	large			
relGoalSize	0.25	0.5	0.75	1
Number of instances SAT	826	562	18	1
Number of instances UNSAT	174	284	410	481
Number of instances timeout	0	154	572	518
Average number of clauses	1,840,776	1,838,285	1,842,420	1,844,834
Average number of variables	41,601	41,573	41,615	41,637
Average path length SAT	10	10	10	10
Average encoding time	3,043	3,062	3,070	3,093
Average solving time SAT	32,252	167,243	247,311	113,609
Average solving time UNSAT	556	1,066	2,456	2,717

Table 5.4.5: Results over 1,000 randomly generated instances of the SMREACH problem for each category. All times are given in milliseconds.

5.5 Evaluation of the Sequence Diagram Checker

Our prototype implementation and our testing environment for our approach to solving the k-SDCHECK problem correspond to the workflow described in Figure 5.1.1.

An instance of the k-SDCHECK problem consists of a set of state machines, a sequence diagram, and a value for the bound k. We translate such an instance to propositional logic by the SAT Encoder using the encoding described in Section 4.4. After the encoding phase, the obtained formula is passed to the SAT solver, which returns `UNSAT`, `SAT`, or it times out.

If it returns `UNSAT`, it means that the problem has no solution, i.e., the sequence of messages specified by the sequence diagram cannot be applied from any state reachable in k steps from the global initial state, and the model is inconsistent. If it returns `SAT`, it means that there exists a solution, i.e., a path in the state machines which conforms to the message sequence in the sequence diagram and is reachable by a path of length at most k from the global initial state. In this case, the SAT solver additionally returns a logical model of the formula representing the problem, which the Solution Builder translates to a solution of the k-SDCHECK instance using the Variable Map. We then apply our simulation tool to confirm the solution returned by the Sequence Diagram Checker and to give feedback to the user.

Other than in the evaluation of our approach to the k-SMREACH problem, we also further process negative instances of the k-SDCHECK problem, i.e., such instances, for which the SAT solver returned `UNSAT`. Therefore, we remove the last message from the encoding and call the SAT solver again. Eventually, the SAT solver returns `SAT` or no message is left. In the former case, the remaining sequence diagram is consistent with the state machines. The information about the removed messages can be used to debug the model.

Figure 5.5.1 shows the graphical user interface of the Sequence Diagram Checker for an instance of the "Coffee" model. The sequence diagram in this model can be executed by the state machines up to and including the symbol wantCoffee sent from bob to cm. The interface allows the user to step through a whole trace by coloring the current messages, transitions, and states, which can be very useful in order to understand the interplay between the different state machines and it provides valuable debugging assistance.

5.5.1 Evaluation with Crafted Instances

Similarly as for the k-SMREACH problem, we tested the implementation of our approach to solving the k-SDCHECK problem using instances based on the set of crafted models described in Section 5.2. For each model, we created a set of consistent instances and a set of inconsistent instances by adding a consistent, respectively an inconsistent, sequence diagram, and by setting the bound k to the values 3, 15, 100, and 500. The sequence diagrams are depicted in Figures 5.5.2, 5.5.3, and 5.5.4.

Recall that the encoding of instances of the k-SDCHECK problem is based on k', which denotes the bound k plus the number of positions required by the sequence diagram. More specifically, k' equals k plus four times the number of messages in the sequence diagram (cf. the description of the encoding in Section 4.4). The encoding of the k-SDCHECK problem hence iterates each constraint up to k' other than only up to k. For this reason, the numbers of clauses and variables are higher for the encodings of instances of the k-SDCHECK problem than they

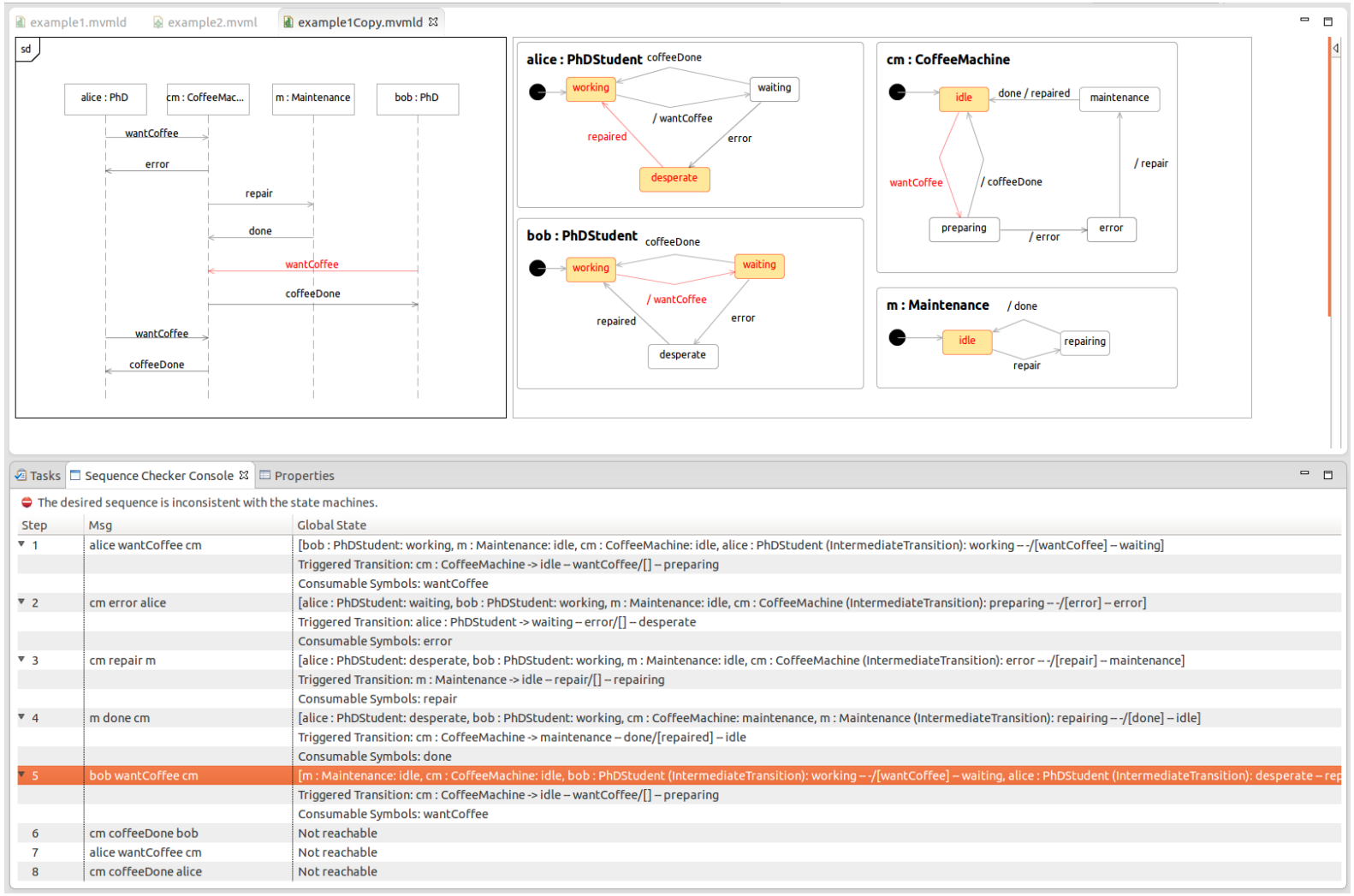

Figure 5.5.1: Graphical user interface of the Sequence Diagram Checker.

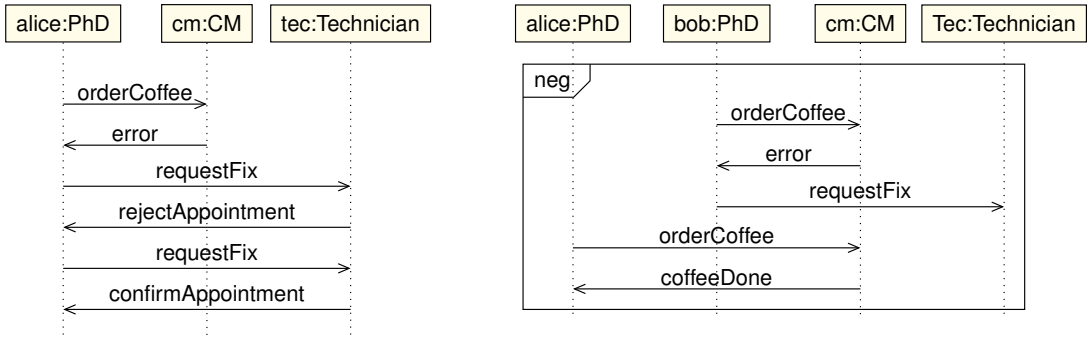

Figure 5.5.2: Two sequence diagrams for the crafted k-SDCHECK instances based on the model in Figure 5.2.1. The left diagram models a desired scenario, the right diagram models an undesired scenario.

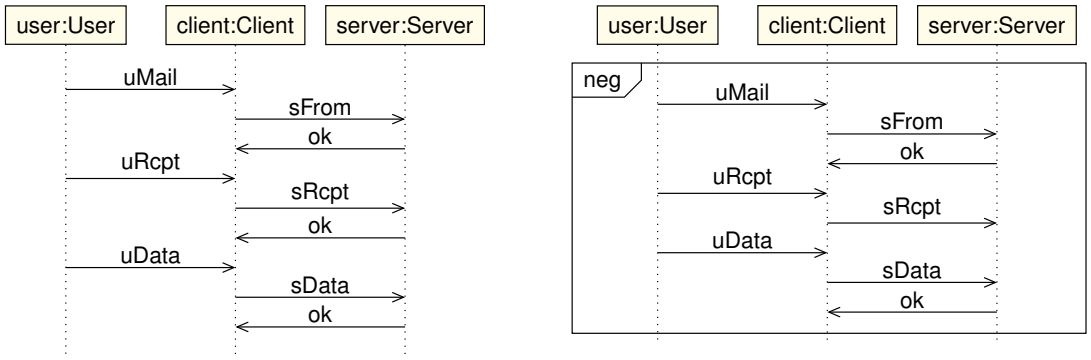

Figure 5.5.3: Two sequence diagrams for the crafted k-SDCHECK instances based on the model in Figure 5.2.2. The left diagram models a desired scenario, the right diagram models an undesired scenario.

are for instances of the k-SMREACH problem although the parameter values for creating the instances are the same.

Details on the outcomes of the test cases are presented in Tables 5.5.1, 5.5.2, and 5.5.3. The first two lines indicate the values for k and k'.

Except for the instances created from the model "Mail" all instances that were created consistently were found to be consistent already for k set to 3. It can be seen from the state machines in Figure 5.2.1 with the left hand sequence diagram in Figure 5.5.2 modeling the case "Coffee" and from the state machines in Figure 5.2.3 with the upper sequence diagram in Figure 5.5.4 modeling the case "Philosophers" that the first message of the sequence diagram occurs as effect in some outgoing transition of the initial state of some state machine. However, this is not the case for the sequence diagram for the model "Mail" in the left hand part of Figure 5.5.3

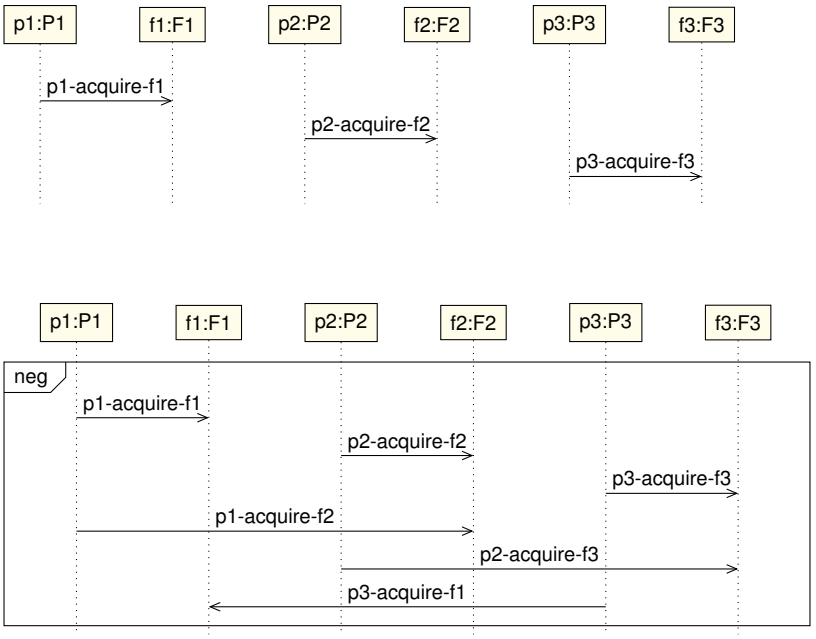

Figure 5.5.4: Two sequence diagrams for the crafted k-SDCHECK instances based on the model in Figure 5.2.3. The top diagram models a desired scenario, the bottom diagram models an undesired scenario.

and the set of state machines depicted in Figure 5.2.2. Here, the state identified of state machine Client has to be reached in order start the path represented by the sequence diagram. Therefore, with k set to 3, the instance is inconsistent, but with k set to 15 or higher, it is consistent.

As can be expected, the numbers of clauses, the numbers of variables, the encoding times, and the solving times increase with increasing values for k. For inconsistent instances, we report two values as solving time. One indicates the time taken to determine inconsistency for the full message sequence and the other indicates the time taken to find the first message where the sequence starts to be inconsistent. The former only consists of solving one unsatisfiable SAT instances, but the latter consists of solving one or more unsatisfiable SAT instances and at most one satisfiable instance. However, the more messages are removed, the smaller the instance becomes with respect to the number of clauses and the number of variables. The encoding times are considerably higher for inconsistent instances due to the implementation. It re-encodes each instance multiple times in order to find the first failing message. The numbers stated represent the sum over all these encodings. This re-encoding is not necessary – a faster implementation would instead track all added clauses and then remove those clauses that relate to the removed message(s).

For consistent instances we further report the length of the path leading to the execution of the sequence diagram. This length is expected to be 0 or more for the instances “Coffee”

and “Philosophers” and 5 or more for the instance “Mail”, as can be verified by the models depicted in the respective figures. Our tool reports longer paths for higher values of k because the number of possible solutions increases with increasing values of k and the SAT solver does not necessarily return the solution with the shortest path.

For inconsistent instances, we report the number of messages that have to be removed in order to make the sequence diagram k-consistent. Here also the values found by our tool are as expected. For example, for the inconsistent instance of the model “Mail” as depicted by the right hand sequence of Figure 5.5.3 and the state machines in Figure 5.2.2 we made a copy of the consistent instance as depicted by the left hand sequence of Figure 5.5.3 and removed the sixth message. As expected, for k set to 15, 100, and 500, the last three messages have to be removed in order for the sequence to be k-consistent. For k set to 3, all messages are removed, as for this bound, neither of the sequence diagrams in Figure 5.5.3 is k-consistent.

All instances except the instances from the model “Philosophers” with the highest bound are not only solved in reasonable time but also they are very easy for the SAT solver to handle. This can be concluded by the solving times being lower than the encoding times. This behavior cannot be observed in the evaluation of the crafted instances of the k-SMREACH problem (cf. Section 5.4, Tables 5.4.1, 5.4.2, and 5.4.3). It seems that the presence of the clauses encoding the sequence diagram allows the SAT solver to work considerably more efficiently on the generated instances of the k-SDCHECK problem than on the generated instances of the k-SMREACH problem.

	Coffee consistent			
k	3	15	100	500
k'	27	39	124	524
Number of clauses	7,895	11,375	36,025	152,025
Number of variables	1,508	2,156	6,746	28,346
Path length	0	0	14	12
Encoding time	212	257	510	1,218
Solving time	39	135	507	673

	Coffee inconsistent			
k	3	15	100	500
k'	23	35	120	520
Number of clauses	11,309	17,273	59,518	258,318
Number of variables	1,783	2,683	9,058	39,058
Messages removed	2	2	2	2
Encoding time	773	808	1,932	3,564
Solving time	37	62	78	156
Solving time to find message	115	285	3,297	56,981

Table 5.5.1: Results of the evaluation of four instances of the k-SDCHECK problem derived from the model "Coffee". All times are given in milliseconds.

	Mail consistent			
k	3	15	100	500
k'	39	51	136	536
Number of clauses	20,459	26,783	71,578	282,378
Number of variables	2,960	3,848	10,138	39,738
Path length	n/a	5	14	14
Messages removed	9	n/a	n/a	n/a
Encoding time	664	366	604	1,548
Solving time	24	125	355	1,132
Solving time to find message	103	n/a	n/a	n/a

	Mail inconsistent			
k	3	15	100	500
k'	35	47	132	532
Number of clauses	18,289	24,613	69,408	280,208
Number of variables	2,659	3,547	9,837	39,437
Messages removed	8	3	3	3
Encoding time	574	564	1,058	4,255
Solving time	19	18	21	76
Solving time to find message	82	314	547	2,857

Table 5.5.2: Results of the evaluation on four instances of the k-SDCHECK problem derived from the model "Mail". All times are given in milliseconds.

	Philosophers consistent			
k	3	15	100	500
k'	15	27	112	512
Number of clauses	4,017	7,257	30,207	138,207
Number of variables	1,088	1,916	7,781	35,381
Path length	0	3	25	170
Encoding time	164	210	530	2,128
Solving time	32	76	894	694,491

	Philosophers inconsistent			
k	3	15	100	500
k'	27	39	124	524
Number of clauses	7,407	10,647	33,597	141,597
Number of variables	1,919	2,747	8,612	36,212
Messages removed	3	3	3	3
Encoding time	470	536	1,373	3,695
Solving time	13	13	25	114
Solving time to find message	121	131	1,389	781,779

Table 5.5.3: Results of the evaluation on four instances of the k-SDCHECK problem derived from the model "Philosophers". All times are given in milliseconds.

	small	medium	large
nrStatemachines	3	6	12
minNrStates	2	4	8
maxNrStates	3	6	12
minNrTrans	6	16	40
maxNrTrans	9	24	60
probTrigger	0.5	0.3	0.3
probEff	0.5	0.3	0.3
nrSymbols	4	8	16
nrLifelines	4	12	30
nrMessages	2	4	10
probInconsistency	0.6	0.6	0.6
k	3	6	12
k'	11	22	52

Table 5.5.4: Parameter values to generate random instances of the k-SDCHECK problem.

5.5.2 Evaluation with Random Instances

We randomly generated instances of the k-SDCHECK problem using our tool described in Section 5.3 and assigned them to different groups according to their size similarly as we did to evaluate our approach to the k-SMREACH problem in Section 5.4.

Except for nrStatemachines, all parameter values are set as for the k-SMREACH problem. The value of nrStatemachines is lower because the size of the instance is regulated by the nrLifelines, i.e., the number of instantiations of the state machines. The values of nrLifelines therefore are the same as for nrStatemachines in the evaluation of the k-SMREACH problem. Further, the parameter probInconsistency denotes the probability for an instance to be tried to be created as negative instance (cf. Section 5.3). Table 5.5.4 shows the values of all parameters. A timeout of 300 seconds was set for the SAT solver.

Recall that the encoding of instances of the k-SDCHECK problem is based on k', which denotes the bound k plus the number of positions required by the sequence diagram. All constraints regarding the consistency therefore refer to positions up to k' rather than k, which can result in a considerably higher number of clauses and variables compared to the encoding of the k-SMREACH problem.

Table 5.5.5 describes the results of our experiments over 1,000 randomly generated instances in each category. We distinguish both encoding times and solving times by `SAT` and `UNSAT` instances. We further show the average times used to find the failing message in the cases of `UNSAT` instances, which includes a call to the SAT solver each time a message is removed from the sequence diagram, and the average number of removed messages. The numbers of clauses and the 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 less clauses than the initial encoding due to the removed message(s). We

distinguish the encoding times by SAT and UNSAT instances because those for UNSAT instances are considerably higher. As discussed above, this is due to the implementation, which re-encodes each instance when a message is removed from an inconsistent instance.

Similarly to the results of the evaluation with random instances of the k-SMREACH problem, the solving times for positive instances are considerably higher than those for negative instances. This is not surprising as the instances are created in a similar way and with the same values for the parameters they have in common (except for parameter nrStatemachines, which corresponds to the parameter nrLifelines of the k-SDCHECK problem). Negative instances are determined so fast that even multiple calls to the SAT solver in order to find the first failing message, are faster than determining a positive instance, as can be seen in the fifth line of Table 5.5.5. Note that finding the first failing message includes solving one positive instances, but this instance is smaller than the initial (negative) instance due to the removal of messages.

The numbers of clauses and the numbers of variables are much higher than those in the respective groups of randomly created instances of the k-SMREACH problem (cf. Table 5.4.5) because all constraints have to be encoded up to k' for the k-SDCHECK problem other than only up to k for the k-SMREACH problem. Like for the crafted instances, we also report the path lengths for the consistent instances and the numbers of removed message for the inconsistent instances. The former unsurprisingly increases with increasing values for k. The latter is around half of the number of messages, which is not surprising given that we create inconsistent instances by adding an arbitrary message at a position chosen uniformly at random.

Similarly as for the k-SMREACH problem, even though the number of variables and the number of clauses increase considerably, the SAT solver still manages to solve the instances with acceptable runtimes except for the instances of the group "large", where more than one third timed out. 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 using 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.

	small	medium	large
Number of instances SAT	399	437	101
Number of instances UNSAT	601	563	540
Number of instances timeout	0	0	359
Average number of clauses	3,940	78,800	2,661,457
Average number of variables	695	6,161	80,734
Average number of removed messages UNSAT	1.5	2.4	5
Average path length SAT	0.7	2.4	7.7
Average encoding time SAT	11	142	3,858
Average encoding time UNSAT	18	335	19,130
Average solving time SAT	3	210	149,072
Average solving time UNSAT	<1	45	3,302
Average solving time UNSAT find message	2	126	103,985

Table 5.5.5: Results over 1,000 randomly generated instances of the k-SMREACH problem for each category. All times are given in milliseconds.

5.6 Evaluation of the Sequence Diagram Merger

The prototype implementation of our approach for solving the SDMERGE problem is an extension of the workflow described in Figure 5.1.1. It consists of four modules: *difference provider*, *SAT encoder*, *SAT solver*, and *model merger*. All modules except for the SAT solver are implemented in Java. The difference provider and the model merger are based on an implementation of the *tMVML* metamodel (cf. Section 3.1) represented as Ecore model within the Eclipse Modeling Framework (EMF).[5] To solve SAT instances, we use the off-the-shelve SAT solver PICOSAT [14].

Figure 5.6.1 depicts the interaction between these modules. First, the *diff provider* prepares the instance to be more easily processable for the *SAT encoder*. Therefore, it takes a *tMVML* conformant trigger-consistent sequence diagram and two revisions thereof as input and calculates the set of atomic differences based on EMF Compare's three-way comparison.[6] Even if EMF Compare reports different kinds of atomic changes, such as add, update, or delete, our current implementation processes only those changes where messages or lifelines are added to the sequence diagrams. The differences are then analyzed and a table that implements the allow function is created.

The SAT Encoder receives as input three sequence diagrams and the table implementing the allow function, and generates the encoding as described in Section 4.2. The non-CNF part of the encoding is converted to CNF on-the-fly. As explained in Section 4.2, each message and each of its allowed positions according to the allow function is represented by a Boolean variable. This information is maintained in a table that maps message/position pairs to Boolean variables and vice versa.

Next, the *SAT solver* processes the generated formula and either returns `UNSAT` or a logical model consisting of a Boolean variable assignment. If a logical model is found, it is handed over to the *merger*. The merger identifies the variables that encode message/position pairs and maps them back to messages and positions. Based on this information, a consolidated version conforming to the *tMVML* metamodel is retrieved by copying the original sequence diagram and inserting the added messages of the two revisions at their respective positions. This consolidated version is verified for its trigger-consistency and added to the set of found solutions.

In practical applications, model developers are likely to be interested in more than one solution to an instances of this problem. They may want to retrieve different consistent merges in order to choose one that fits their intentions. Therefore, we try to find another solution by adding the negation of the logical model to the previous encoding. This way we make sure that in the following iteration a different logical model, if one exists, is found. This procedure is repeated until the SAT solver returns `UNSAT`, i.e., no more solutions exist.

We evaluated our approach based on the crafted benchmark set described in Section 5.2 which consists of three different families modeling a coffee machine, a simplified version of the SMTP protocol, and a variant of the dining philosophers problem. We defined five versioning scenarios for each of these families. For each versioning scenario, we distinguished three different cases with respect to the state machines related to the lifelines. In real-life development

[5] `http://www.eclipse.org/modeling/emf/`
[6] `http://wiki.eclipse.org/EMF_Compare`

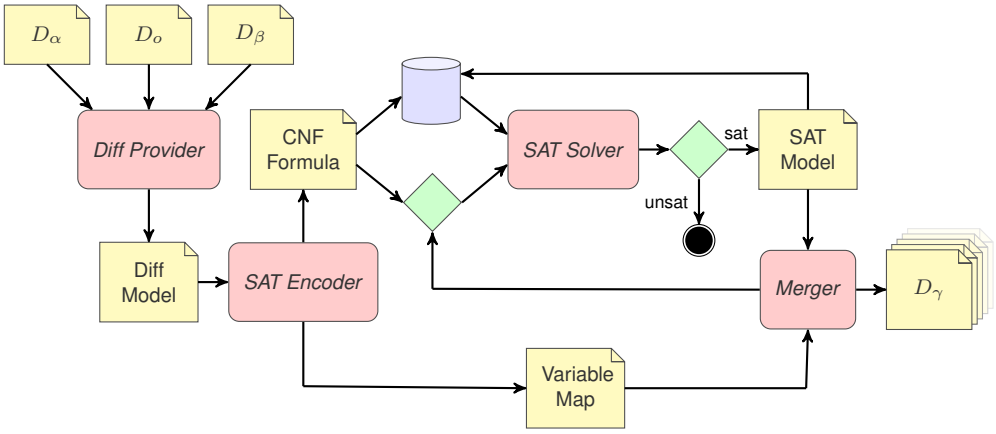

Figure 5.6.1: Implementation workflow to retrieve solutions for instances of the SDMERGE problem.

scenarios, it can happen that a lifeline is sketched without having a state machine specified. If no state machine is specified for a lifeline, we assume a state machine that contains only one state from which any action symbol can be received, similar to the state machine "User" shown in Figure 4.2.1 of Section 4.2. The cases we distinguish therefore are (1) all lifelines are fully specified by state machines, (2) some lifelines are specified by state machines, and (3) no lifeline is specified by a state machine.

The number of solutions when no state machine is specified equals the number of time-consistent solutions described in Observation 4.2.3 for the same sequence diagrams because time consistency disregards the state machine view. This number constitutes an upper bound to the number of solutions. In practice, it is drastically reduced by the constraints imposed by trigger consistency with the state machines.

The state machines featured in three different sets of problem instances are described in Section 5.2. The sizes of these state machines are described in Table 5.2.1. For each set, we constructed five different versioning scenarios as sequence diagrams. The sizes of these sequence diagrams are shown in the first six rows of Tables 5.6.1, 5.6.2, and 5.6.3. In particular, they show the names, the number of lifelines, and the number of messages of each instance's original sequence diagram and its two revisions.

These tables further show statistics on the numbers of solutions and on the runtimes of the different instances. For those instances whose number of solutions exceeded 1,000 we stopped the algorithm when 1,000 solutions were found. For each solution we confirmed its correctness as described in Section 5.1.

As could be expected, the evaluation shows that in general a specification of all state machines results in few solutions quickly found and a specification of few state machines results in many solutions. Some instances were created in a way that no merge is possible, which was also returned by the solver. As expected, when no state machines are specified, the number of merges adhered to the number of merges defined by formula discussed in Observation 4.2.3.

The runtimes are feasible for practical applications whenever there are few solutions. To deal

Coffee						
Instance		1	2	3	4	5
D_o	Number of lifelines	2	2	2	2	2
	Number of messages	5	5	0	5	5
D_α	Number of lifelines	2	2	2	2	2
	Number of messages	6	6	2	9	9
D_β	Number of lifelines	2	2	2	2	2
	Number of messages	9	6	2	9	9
All state machines	Number of solutions	2	0	2	2	34
	Runtime	<1	<1	<1	<1	2.9
Some state machines	Number of solutions	2	0	6	70	34
	Runtime	<1	<1	<1	6.0	2.9
No state machine	Number of solutions	5	2	6	70	70
	Runtime	<1	<1	<1	6.0	6.0

Table 5.6.1: Overview of the SDMERGE instances derived from the model "Coffee", their numbers of solutions, and their runtimes in seconds.

Mail						
Instance		1	2	3	4	5
D_o	Number of lifelines	3	3	3	3	3
	Number of messages	5	5	5	5	5
D_α	Number of lifelines	3	3	3	4	4
	Number of messages	7	8	14	14	14
D_β	Number of lifelines	3	3	3	3	3
	Number of messages	12	15	14	16	18
All state machines	Number of solutions	1	0	2	2	2
	Runtime	<1	<1	<1	<1	1.5
Some state machines	Number of solutions	1	2	2	2	2
	Runtime	<1	<1	<1	<1	<1
No state machine	Number of solutions	55	110	>1,000	>1,000	>1,000
	Runtime	3.8	8.0	205	215	232

Table 5.6.2: Overview of the SDMERGE instances derived from the model "Mail", their numbers of solutions, and their runtimes in seconds.

with the existence of too many solutions, an extension of our approach could accept additional constraints from the developers. For example, the developers may not want to consider all possible interleavings of messages that have been added between two messages of the original sequence diagram as the intended addition to a sequence diagram by one developer may be a scenario that they do not want to be interrupted.

Philosophers						
Instance		1	2	3	4	5
D_o	Number of lifelines	4	4	4	4	4
	Number of messages	0	0	0	0	0
D_α	Number of lifelines	4	4	4	4	4
	Number of messages	2	1	9	9	9
D_β	Number of lifelines	4	4	4	4	4
	Number of messages	5	5	9	5	5
All state machines	Number of solutions	6	0	506	253	0
	Runtime	<1	<1	90	33	<1
Some state machines	Number of solutions	15	5	>1,000	>1,000	>1,000
	Runtime	1.4	<1	201	167	168
No state machine	Number of solutions	15	5	>1,000	>1,000	>1,000
	Runtime	1.8	<1	164	120	121

Table 5.6.3: Overview of the SDMERGE instances derived from the model "Philosophers", their numbers of solutions, and their runtimes in seconds.

Chapter 6

Conclusions

We conclude this work with a summary of the main results and a discussion of future research directions. The work presented in this thesis was motivated by the increasing valorization of software models driven by the new paradigm of model-based software engineering. In their new role as first-class development artifacts, it is imperative for the models to have a formal semantics. Based on this semantics, consistency problems that occur during the evolution of a system can be identified and solved.

We *formalized a modeling language* based on the popular but mostly informal modeling language UML. To this end, we first described its syntax and behavioral aspects in a mathematical notation and then used the formal language of propositional logic to express some semantic properties regarding model consistency. Using this formalization, we *identified three problems* that can occur in models using this language; the SDMERGE problem, the k-SMREACH problem, and the k-SDCHECK problem. The SDMERGE problem asks whether two modifications of a sequence diagram can be merged into a new sequence diagram in a way that preserves consistency with the state machines of the model. The k-SMREACH problem asks whether in a set of state machines some combination of states is reachable in k steps from some global state. The k-SDCHECK problem asks whether the communication between a given set of state machines represents the sequence of messages defined by a sequence diagram, i.e., whether the two views are *k-consistent*.

We proposed an *encoding to the satisfiability problem of propositional logic* for each of the problems. This is a popular approach to NP-complete problems in order to solve them by off-the-shelf solvers. We also used this approach to solve the SDMERGE problem, which is tractable, in order to obtain a basis that we can use for future, possibly intractable, extensions of this problem.

We further showed the *computational complexity* of each of the problems: P-membership for the SDMERGE problem and NP-completeness for the other two problems.

We *evaluated* our approaches based on a set of handcrafted models and on grammar-based

whitebox fuzzing, for which we developed a random model generator. The implementation was based on the Eclipse Modeling Framework (EMF) [49] and the modeling language we expressed as Ecore model. The results of our experiments showed that we can solve instances of these problems of reasonable size on standard hardware with freely available SAT solvers despite of the generated instances of the SAT problem being relatively large.

We draw the following conclusions from this work:

- A concise formalization and a formal semantics of a modeling language are essential for the identification and for solving verification problems. Only if the meaning of the concepts is fixed, the properties they should satisfy can be unambiguously formulated and then be checked.
- Propositional logic is a practical host language to solve problems in the area of model verification. The similarity between the encoding of the k-SMREACH problem and of the k-SDCHECK problem shows how, by small adaptations, the encoding of one problem can be used for other problems in this area.
- An off-the-shelf SAT solver solves SAT instances generated from instances of the discussed verification problems in less than one second for randomly generated instances containing up to 12 state machines with a total of up to 72 states and 288 transitions, regardless of the instance being positive or negative. Larger instances, with up to 30 state machines and with a total of up to 360 states and 1800 transitions require more time, in particular when they are positive. Some negative instances of this size could be solved in less than five seconds, but the positive instances solved required around 2.5 minutes.
- The definition of the modeling language used in this work contains basic modeling concepts of the UML. However, despite leaving out more complex modeling features, two of the problems we identified are already NP-hard. It can be expected that extensions of the language result in even harder problems that have to be tackled with different approaches.

We proceed with discussing some open issues that follow immediately from this work and some long-term research topics in this area.

6.1 Open Issues

In the following we describe some future work regarding the three problems discussed in this work and regarding the evaluation of our approach.

6.1.1 The SDMERGE Problem

Our definition of the SDMERGE problem (cf. Section 4.2) is restricted in at least two ways. First, it is based on the weaker notion of trigger consistency rather than on full consistency (cf. Definitions 3.3.3 and 3.3.13). Second, it allows only additions as changes, i.e., it does not consider deletions. Further, as can be seen from our evaluation problem (cf. Section 5.6), the set of solutions can become very large, which is impractical. Direct interaction with the human

modeler can help to restrict the set of solutions when too many are available. Also, such an interaction could provide means to weaken the constraints when no solution can be found.

An approach that considers full consistency (or k- consistency) and changes that can consist of both additions and deletions would be more useful in practice. However, such extensions also pose new challenges when dealing with this problem. Using the semantics of full consistency other than trigger consistency implies taking into account the communication between state machines, which considerably increases the state space that has to be considered. This possibly lifts the computational complexity of the problem.

In order to solve such a variation of the SDMERGE problem based on k-consistency, ideas from the encodings of the k-SMREACH problem and the k-SDCHECK problem (cf. Sections 4.3 and 4.4) could be used, since these encodings include the semantics of k-consistency.

Dealing with deletions of messages as part of a revision of a sequence diagram imposes new challenges in interpreting the intentions of the modelers. For example, it could happen that modeler A adds a sequence of messages to the end of a sequence of messages of the original sequence diagram, modeler B deletes the last message of the original sequence diagram, but this last message is required in order to execute the sequence added by modeler A. Such contradictory changes can also happen when considering only additions as it could happen that no interleaving exists that contains all added messages of both modelers. However, when also considering deletions this may be even more likely to happen. Hence, it will be necessary to develop a method to capture the intention of the modelers whenever consistent merges are not possible. This could based on an interaction with the modeler, where they can add their intention via a graphical interface and instruct the tool to overrule one change or another.

An explicit expression of the modeler's intention could also help to decrease the number of possible merges, which can become very high (cf. Section 5.6). For example, the intention of a developer when adding a sequence of messages to a sequence diagram may be to keep this sequence atomic, i.e., without interleavings of another message sequence. Such constraints could be added graphically, e.g., by adding a combined fragment like "critical" defined by the UML [62].

6.1.2 The k-SMREACH and k-SDCHECK Problems

For these two problems, the communication between state machines (respectively, instances of state machines) takes place by multimessages, where a multimessage contains a set of messages sent by one state machine and received by different state machines. However, it is not clear what sending and receiving multimessages means for sequence diagrams. In our definitions (cf. Section 3.2) a sequence diagram contains only single messages. For the formulation of the k-SDCHECK problem this did not pose any problem, because a message is also a multimessage. It would be interesting to define a notion of sending and receiving multimessages in sequence diagrams and to clarify their impact on the notion of consistency with the state machines.

Another open issue is dealing with the bound k, which strongly influences not only the size of the encoding but also the performance of the SAT solver, as we could see from our experiments on crafted instances (cf. Sections 5.4 and 5.5). It would therefore be useful to have a method to determine a lower bound $\hat{k}$ for k such that a $\hat{k}$-inconsistent model remains $\hat{k}$-inconsistent for k greater than $\hat{k}$.

6.1.3 Evaluation

The results of our evaluations highly depend on the employed SAT solver. Depending on the solving techniques implemented by the SAT solver it may perfom better or worse on one type of encoding than on another. It could be interesting to compare the performance of different SAT solvers on our instances.

Given the lower computational complexity of the SDMERGE problem, it could be interesting to compare the symbolic approach to an implementation of the algorithm described in the proof of Theorem 4.2.1.

Further, the random generation of negative instances of the k-SDCHECK problem could be implemented in different ways. In this work, we inserted a random message at a random position assuming this to likely result in an inconsistent model. However, this method resulted in negative instances that were very easy to solve (cf. results in Section 5.5). In real life, it can be expected that an error introduced by a human modeler is not that random. For instance, they may rather insert a message that does indeed occur as trigger, respectively effect, in the state machines of the connected lifelines, but that only cannot be executed at the respective position. It could be interesting to see whether such inconsistencies are harder to find by the SAT solver.

6.2 Beyond SAT

In this work, we considered software models that implement a subset of the concepts suggested by popular modeling languages like the UML. This way, we built a foundation of formal semantics for a modeling language and of symbolic approaches to the consistency problems encountered. These problems were inside the complexity class NP and therefore reasonably solvable by a SAT solver. However, many practical extensions to both the modeling language and the problems described in this work can be found that probably result in problems that are of higher computational complexity. Also, problems other than verification of consistency can be identified, which also can be of higher computational complexity.

In this section, we present some ideas for extensions of the language and the problems discussed in this work. We propose to use quantified Boolean formulas [76] as a host language to solve these problems if they are PSPACE-complete. We further summarize our previous results in the area of solution extraction for QBF, which contributed to the applicability of QBF toolsets.

Possible extensions and new problems include the following.

6.2.1 Additional Language Concepts

Other language concepts can be added to our modeling language in order to make it more useful in practice. These include hierarchical states for state machines, combined fragments, invariants, and guards for sequence diagrams, and other types of diagrams like the activity diagram, all suggested by the UML Superstructure [62]. With these concepts added, we will have to extend the definition of consistency, and the computational complexity of the consistency problems discussed in this work may increase.

6.2.2 Extensions of the k-SMREACH and k-SDCHECK Problems

In this work, we considered k-bounded formulations of these two problems (cf. Sections 4.3 and 4.4). The chosen bound strongly influences not only the size of the encoding but also the performance of the SAT solver, as we could see from our experiments on crafted instances (cf. Sections 5.4 and 5.5). It will therefore be useful to investigate unbounded formulations of these problems and to determine their computational complexity. Depending on the result, solving the problems based on a QBF encoding can be attempted.

6.2.3 Identification of New Problems

The following are two examples for new problems that can occur during the evolution of software models. First, in *automated merging* environments like model versioning systems, it would be useful to have the merging tool compute a *repair strategy* whenever two versions of a model cannot be consistently merged. The need for assisted merging environments has already been identified by France and Rumpe [52] and Almeida da Silva et al. [37] and an approach to consistency detection and repair in a multi-view environment has been suggested by Macedo et al. [90], who employed a SAT solver for these tasks. For the models considered in our previous work, such a repair strategy could include the deletion or addition of messages or a modification of the state machines. Second, keeping state machines consistent with a set of sequence diagrams or creating a new set of state machines that is consistent with a given set of scenarios described in a set of sequence diagrams can be considered a *synthesis problem*. A tool solving such problems could help the developer create a software system from a set of scenarios. Works by Bloem et al. [16–18] could serve as basis for such an approach.

6.2.4 QBF as Host Language

In this work, we have shown that encodings to the SAT problem and the application of SAT solvers work well to solve verification problems of software models that are in the complexity class NP. To solve problems of higher complexity, a similar approach is to use symbolic methods like encodings to QBF for those problems that are PSPACE-complete. Such encodings allow to employ existing tools in a similar way as we employed a SAT solver for problems discussed in this work.

Practical applications of QBF as host language exist for problems in the areas of hardware debugging [1, 91], verification [38, 111], planning [104], two-player games [57], electronic design automation [70, 81], among others. In particular, QBF encodings proposed for bounded model checking [38, 71] could be interesting for our future work with encoding problems occurring in software model engineering.

Quantified Boolean logic extends propositional logic (cf. Section 2.2 by allowing quantifiers over Boolean variables. However, since the determination of truth or falsity of a QBF is a PSPACE-complete problem, dealing with the solution of a QBF is more difficult than it is for a formula in propositional logic. A solution of a QBF can be represented as a control strategy [57] expressed by an algorithm that computes assignments to universal variables rendering the QBF *false*, respectively existential variables rendering the QBF *true*, or as a control cir-

cuit [4] expressed by Herbrand functions for a false QBF, respectively Skolem functions for a true QBF.

Similarly to a model of a propositional formula, a control strategy or a circuit can be mapped to the solution of a problem encoded as a QBF, for example, the system to be synthesized. Therefore, efficiently dealing with solutions of a QBF is crucial for the practical usefulness of a symbolic encoding of a problem into a QBF. State-of-the-art approaches to obtain strategies or circuits require traversing a Q-resolution proof [77] of a QBF after being returned by a QBF solver such as DepQBF [88]. However, for many real-life applications, these Q-resolution proofs are too large to handle.

To deal with this problem, much effort has been spent recently in the QBF community. In particular, the Q-resolution calculus has since been extended in different ways. Balabanov and Jiang [4] proposed long-distance Q-resolution (LQ-resolution), which allows certain tautological resolvents for the deduction rules of Q-resolution, and Van Gelder proposed to add resolution over universal variables to the rules of Q-resolution (QU-resolution) [119]. Based on these results, we showed the following in other work.

- LQ-resolution allows for exponentially shorter proofs compared to Q-resolution for a family of QBFs [41, 42].
- LQ-resolution implemented in the popular solver DepQBF results in better performance with respect to the number of backtracks, resolution steps, and learned clauses [41].
- Strategy extraction as proposed by Goultiaeva et al. [57] can be applied in the same manner to LQ-resolution [41, 42].
- The extraction of circuits from LQ-resolution proofs is possible in time polynomial in the size of the proof [5].
- LQ-resolution and QU-resolution are incomparable with respect to their proof complexity [6].
- A proof system that combines LQ-resolution with QU-resolution is even more powerful than any of the two [6].

Our future work will hence include the identification of harder problems arising in the area of software modeling. Building on the propositional encodings presented in this work, we will translate these problems to a more powerful host language. QBF will be a natural choice to serve as such a language, being a natural extension of propositional logic, given the recent advances in QBF solving and certification, and given the availability of off-the-shelf toolsets. Also, the previously mentioned existing approaches to solving similar problems with QBF encodings can be used as references. It will be interesting to see how recent efforts in QBF solving, in particular our previous work on efficient solution extraction, affects practical results in solving such problems and whether state-of-the-art QBF solvers are capable to solve them in reasonable time.

Bibliography

[1] Moayad Fahim Ali, Sean Safarpour, Andreas G. Veneris, Magdy S. Abadir, and Rolf Drechsler. Post-verification debugging of hierarchical designs. In *2005 International Conference on Computer-Aided Design*, pages 871–876. IEEE Computer Society, 2005.

[2] Kerstin Altmanninger, Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. Why Model Versioning Research is Needed!? In *Proceedings of the MoDSE-MCCM Workshop @ MoDELS*, 2009.

[3] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems*, 23(3):273–303, May 2001.

[4] Valeriy Balabanov and Jie-Hong R. Jiang. Unified QBF Certification and Its Applications. *Formal Methods in System Design*, 41:45–65, 2012.

[5] Valeriy Balabanov, Jie-Hong Roland Jiang, Mikolas Janota, and Magdalena Widl. Efficient extraction of QBF (counter)models from long-distance resolution proofs. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 3694–3701. AAAI Press, 2015.

[6] Valeriy Balabanov, Magdalena Widl, and Jie-Hong R. Jiang. QBF resolution systems and their proof complexities. In *Theory and Applications of Satisfiability Testing*, volume 8561 of *LNCS*, pages 154–169. Springer, 2014.

[7] Stephen Barrett, Patrice Chalin, and Greg Butler. Model Merging Falls Short of Software Engineering Needs. In *Proceedings of the MoDSE Workshop @ MoDELS 2008*, 2008.

[8] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716, 1952.

[9] Lars Bendix and Pär Emanuelsson. Requirements for Practical Model Merge – An Industrial Perspective. In *Model Driven Engineering Languages and Systems*, volume 5795 of *LNCS*, pages 167–180. Springer, 2009.

[10] Marco Benedetti. sKizzo: A suite to evaluate and certify QBFs. In *Automated Deduction – CADE-20*, pages 369–376. Springer, 2005.

[11] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1-2):70–118, 2005.

[12] Simona Bernardi, Susanna Donatelli, and José Merseguer. From UML sequence diagrams and statecharts to analysable Petri net models. In *Proceedings of the 3rd International Workshop on Software and Performance*, pages 35–45. ACM, 2002.

[13] Jean Bézivin. On the Unification Power of Models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[14] Armin Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, 2008.

[15] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[16] Roderick Bloem, Uwe Egly, Patrick Klampfl, Robert Könighofer, and Florian Lonsing. SAT-based methods for circuit synthesis. In *FMCAD 2014*, pages 31–34. IEEE, 2014.

[17] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of Reactive(1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012.

[18] Roderick Bloem, Robert Könighofer, and Martina Seidl. SAT-based synthesis methods for safety specs. In *Verification, Model Checking, and Abstract Interpretation*, volume 8318 of *LNCS*, pages 1–20. Springer, 2014.

[19] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[20] Frederick P. Brooks Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987.

[21] Petra Brosch, Uwe Egly, Sebastian Gabmeyer, Gerti Kappel, Martina Seidl, Hans Tompits, Magdalena Widl, and Manuel Wimmer. Towards Scenario-Based Testing of UML Diagrams. In *Tests and Proofs 2012*, volume 7305 of *LNCS*, pages 149–155. Springer, 2012.

[22] Petra Brosch, Uwe Egly, Sebastian Gabmeyer, Gerti Kappel, Martina Seidl, Hans Tompits, Magdalena Widl, and Manuel Wimmer. Towards semantics-aware merge support in optimistic model versioning. In *Models in Software Engineering – Workshops and Symposia at MODELS 2011*, volume 7167 of *LNCS*, pages 246–256. Springer, 2012.

[23] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. The Past, Present, and Future of Model Versioning. In *Emerging Technologies for the Evolution and Maintenance of Software Models*, chapter 15, pages 410–443. IGI Global, 2011.

[24] Petra Brosch, Gerti Kappel, Martina Seidl, Konrad Wieland, Manuel Wimmer, Horst Kargl, and Philip Langer. Adaptable Model Versioning in Action. In *Modellierung*, volume 161 of *LNI*, pages 221–236. GI, 2010.

[25] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. Colex: A web-based collaborative conflict lexicon. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, pages 42–49. ACM, 2010.

[26] Petra Brosch, Martina Seidl, and Magdalena Widl. Semantics-aware versioning challenge: Merging sequence diagrams along with state machine diagrams. *Softwaretechnik-Trends*, 33(2), 2013.

[27] Manfred Broy and María Victoria Cengarle. UML formal semantics: lessons learned. *Software and Systems Modeling*, 10(4):441–446, 2011.

[28] Hans Kleine Büning and Theodor Lettmann. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, 1999.

[29] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verifying UML/OCL Operation Contracts. In *Integrated Formal Methods*, volume 5423 of *LNCS*, pages 40–55. Springer, 2009.

[30] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. Managing model conflicts in distributed development. In *Model Driven Engineering Languages and Systems*, volume 5301 of *LNCS*, pages 311–325. Springer, 2008.

[31] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[32] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead*, volume 2000 of *LNCS*, pages 176–194. Springer, 2001.

[33] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[34] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[35] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[36] Thomas H. Cormen, Clifford Stein, Charles E. Leiserson, and Robert L. Rivest. *Introduction to Algorithms*. MIT Press, 2001.

[37] Marcos Aurélio Almeida da Silva, Alix Mougenot, Xavier Blanc, and Reda Bendraou. Towards automated inconsistency handling in design models. In *Advanced Information Systems Engineering*, volume 6051 of *LNCS*, pages 348–362. Springer, 2010.

[38] Nachum Dershowitz, Ziyad Hanna, and Jacob Katz. Bounded model checking with QBF. In *Theory and Applications of Satisfiability Testing*, volume 3569 of *LNCS*, pages 408–414. Springer, 2005.

[39] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In *Models in Software Engineering – Workshops and Symposia at MODELS 2011*, volume 6627 of *LNCS*, pages 165–179. Springer, 2011.

[40] Jori Dubrovin and Tommi A. Junttila. Symbolic Model Checking of Hierarchical UML State Machines. In *Application of Concurrency to System Design*, pages 108–117. IEEE, 2008.

[41] Uwe Egly, Florian Lonsing, and Magdalena Widl. Long-distance resolution: Proof generation and strategy extraction in search-based QBF solving. In *Logic Programming and Automated Reasoning*, volume 8312 of *LNCS*, pages 291–308. Springer, 2014.

[42] Uwe Egly and Magdalena Widl. Solution extraction from long-distance resolution proofs. In *International Workshop on Quantified Boolean Formulas 2013, Workshop Report*, pages 6–15, 2013.

[43] Alexander Egyed. Instant Consistency Checking for the UML. In *International Conference on Software Engineering*, pages 381–390. ACM, 2006.

[44] Alexander Egyed. UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models. In *International Conference on Software Engineering*, pages 793–796. IEEE, 2007.

[45] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Testing the consistency of dynamic UML diagrams. In *Integrated Design and Process Technology*, 2002.

[46] Rik Eshuis. Symbolic model checking of UML activity diagrams. *ACM Transactions on Software Engineering and Methodology*, 15(1):1–38, January 2006.

[47] Andy Evans and Stuart Kent. Core meta-modelling semantics of UML: the pUML approach. In *«UML»'99—The Unified Modeling Language*, volume 1723 of *LNCS*, pages 140–155. Springer, 1999.

[48] Thomas Huining Feng and Hans Vangheluwe. Case study: Consistency problems in a UML model of a chat room. In *Workshop on Consistency Problems in UML-based Software Development*, page 18, 2003.

[49] Eclipse Foundation. The Eclipse Modeling Framework (EMF). `http://www.eclipse.org/modeling/emf/`, September 2014.

[50] Robert B. France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. *Computer Standards & Interfaces*, 19(7):325–334, 1998.

[51] Robert B. France, Sudipto Ghosh, Trung Dinh-Trong, and Arnor Solberg. Model-driven development using UML 2.0: promises and pitfalls. *IEEE Computer*, 39(2):59–66, 2006.

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

[53] Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. A classification of model checking-based verification approaches for software models. In *STAF Workshop on Verification of Model Transformations*, pages 1–7, 2013.

[54] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[55] Christian Gerth, Jochen Malte Küster, Markus Luckey, and Gregor Engels. Detection and resolution of conflicting change operations in version management of process models. *Software and System Modeling*, 12(3):517–535, 2013.

[56] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QuBE++: An Efficient QBF Solver. In *Formal Methods in Computer-Aided Design*, volume 3312 of *LNCS*, pages 201–213, 2004.

[57] Alexandra Goultiaeva, Allen Van Gelder, and Fahiem Bacchus. A uniform approach for generating proofs and strategies for both true and false QBF formulas. In *22nd International Joint Conference on Artificial Intelligence*, pages 546–553. AAAI Press, 2011.

[58] Bas Graaf and Arie van Deursen. Model-driven consistency checking of behavioural specifications. In *4th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 115–126. IEEE Computer Society, 2007.

[59] Object Management Group. Semantics of a foundational subset for executable UML models (FUML), version 1.0. `http://www.omg.org/spec/FUML/1.0/PDF/`, August 2011. Retrieved Aug 26, 2014.

[60] Object Management Group. Specification of the unified modeling language (OMG UML). `http://www.omg.org/spec/UML/`, August 2011. retrieved Aug 26, 2014.

[61] Object Management Group. Unified modeling language (OMG UML), Infrastructure V2.4.1. http://www.omg.org/spec/UML/2.4.1/, August 2011. retrieved Aug 26, 2014.

[62] Object Management Group. Unified modeling language (OMG UML), Superstructure V2.4.1. http://www.omg.org/spec/UML/2.4.1/, August 2011. retrieved Aug 26, 2014.

[63] Object Management Group. Object constraint language, version 2.4. http://www.omg.org/spec/OCL/2.4/, August 2014. retrieved Aug 26, 2014.

[64] David Harel. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer*, 25(1):8–20, 1992.

[65] David Harel and Bernhard Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, Weizmann Institute of Science, Jerusalem, Israel, 2000.

[66] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"?. *IEEE Computer*, 37(10):64–72, 2004.

[67] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[68] Paola Inverardi, Henry Muccini, and Patrizio Pelliccione. Automated check of architectural models consistency using SPIN. In *Automated Software Engineering*, pages 346–349. IEEE Computer Society, 2001.

[69] Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund M. Clarke. Solving QBF with Counterexample Guided Refinement. In *Theory and Applications of Satisfiability Testing*, volume 7317 of *LNCS*, pages 114–128. Springer, 2012.

[70] Jie-Hong R. Jiang, Hsuan-Po Lin, and Wei-Lun Hung. Interpolating Functions from Large Boolean Relations. In *International Conference on Computer-Aided Design*, pages 779–784. IEEE/ACM, 2009.

[71] Toni Jussila and Armin Biere. Compressing BMC encodings with QBF. *Electronic Notes in Theoretical Computer Science*, 174(3):45–56, 2007.

[72] Petra Kaufmann, Martin Kronegger, Andreas Pfandler, Martina Seidl, and Magdalena Widl. Global state checker: Towards SAT-based reachability analysis of communicating state machines. In *10th Workshop on Model-Driven Engineering, Verification, and Validation*, number 1069 in CEUR Workshop Proceedings, pages 31–40, 2013.

[73] Petra Kaufmann, Martin Kronegger, Andreas Pfandler, Martina Seidl, and Magdalena Widl. Intra- and interdiagram consistency checking of behavioral multiview models. *Computer Languages, Systems & Structures*, 44:72–88, 2015.

[74] Petra Kaufmann, Martin Kronegger, Andreas Pfandler, Martina Seidl, and Magdalena Widl. A SAT-based debugging tool for state machines and sequence diagrams. In *Software Language Engineering*, volume 8706 of *LNCS*, pages 21–40. Springer, 2015.

[75] Soon-Kyeong Kim and David Carrington. Formalizing the UML class diagram using Object-Z. In *«UML»'99—The Unified Modeling Language*, volume 1723 of *LNCS*, pages 83–98. Springer, 1999.

[76] Hans Kleine Büning and Uwe Bubeck. *Handbook of Satisfiability*, chapter QBF Theory. Oxford University Press, 2009.

[77] Hans Kleine Büning, Marek Karpinski, and Andreas Flögel. Resolution for Quantified Boolean Formulas. *Information and Computation*, 117(1):12–18, February 1995.

[78] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.

[79] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking – timed UML state machines and collaborations. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *LNCS*, pages 395–416, 2002.

[80] Alexander Knapp and Jochen Wuttke. Model checking of uml 2.0 interactions. In *Models in Software Engineering*, volume 4364 of *LNCS*, pages 42–51. Springer, 2007.

[81] Chih-Fan Lai, Jie-Hong R. Jiang, and Kuo-Hua Wang. BooM: A Decision Procedure for Boolean Matching with Abstraction and Dynamic Learning. In *Design Automation Conference*, pages 499–504. ACM/IEEE, 2010.

[82] Vitus S.W. Lam and Julian Padget. Consistency Checking of Sequence Diagrams and Statechart Diagrams Using the π-Calculus. In *Integrated Formal Methods*, volume 3771 of *LNCS*, pages 347–365. Springer, 2005.

[83] Kevin Lano, Juan Bicarregui, and Andy Evans. Structured axiomatic semantics for UML models. In *Rigorous Object-Oriented Methods*, Workshops in Computing, pages 5–20. British Computer Society, 2000.

[84] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997.

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

[86] Johan Lilius and Ivan Paltor. vUML: A tool for verifying UML models. In *Automated Software Engineering*, pages 255–258. IEEE Computer Society, 1999.

[87] Vitor Lima, Chamseddine Talhi, Djedjiga Mouheb, Mourad Debbabi, Lingyu Wang, and Makan Pourzandi. Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages. *Electronic Notes in Theoretical Computer Science*, 254:143–160, 2009.

[88] Florian Lonsing and Uwe Egly. Incremental QBF solving. In *Principles and Practice of Constraint Programming*, volume 8656 of *LNCS*, pages 514–530. Springer, 2014.

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

[90] Nuno Macedo, Tiago Guimarães, and Alcino Cunha. Model repair and transformation with Echo. In *Automated Software Engineering*, pages 694–697. IEEE, 2013.

[91] Hratch Mangassarian, Andreas G. Veneris, Sean Safarpour, Marco Benedetti, and Duncan Exon Smith. A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test. In *International Conference on Computer-Aided Design*, pages 240–245. IEEE Computer Society, 2007.

[92] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A manifesto for semantic model differencing. In *Models in Software Engineering*, volume 6627 of *LNCS*, pages 194–203. Springer, 2010.

[93] Kenneth L McMillan. *Symbolic model checking*. Springer, 1993.

[94] Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In *Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 200–214. Springer, 2006.

[95] Huaikou Miao, Ling Liu, and Li Li. Formalizing UML models with Object-Z. In *Formal Methods and Software Engineering*, volume 2495 of *LNCS*, pages 523–534. Springer, 2002.

[96] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and Merging of Statecharts Specifications. In *International Conference on Software Engineering*, pages 54–64. IEEE, 2007.

[97] Artur Niewiadomski, Wojciech Penczek, and Maciej Szreter. Towards checking parametric reachability for UML state machines. In *Ershov Memorial Conference*, volume 5947 of *LNCS*. Springer, 2010.

[98] Iulian Ober, Susanne Graf, and Ileana Ober. Validation of UML models via a mapping to communicating extended timed automata. In *Model Checking Software*, volume 2989 of *LNCS*, pages 127–145. Springer, 2004.

[99] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[100] David Parnas. Software engineering or methods for the multi-person construction of multi-version programs. In *Programming Methodology*, volume 23 of *LNCS*, pages 225–235. Springer, 1974.

[101] Patrizio Pelliccione, Paola Inverardi, and Henry Muccini. CHARMY: A Framework for Designing and Verifying Architectural Specifications. *IEEE Transactions on Software Engineering*, 35(3):325–346, 2008.

[102] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

[103] Thomas Reiter, Kerstin Altmanninger, Alexander Bergmayr, Wieland Schwinger, and Gabriele Kotsis. Models in conflict – detection of semantic conflicts in model-based development. In *International Workshop on Model-Driven Enterprise Information Systems*, pages 29–40, 2007.

[104] Jussi Rintanen. Asymptotically Optimal Encodings of Conformant Planning in QBF. In *National Conference on Artificial Intelligence*, pages 1045–1050. AAAI Press, 2007.

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

[106] James E. Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman Ltd., Essex, UK, 1999.

[107] Mehrdad Sabetzadeh, Shiva Nejati, Sotirios Liaskos, Steve M. Easterbrook, and Marsha Chechik. Consistency checking of conceptual models via model merging. In *International Requirements Engineering Conference*. IEEE Computer Society, 2007.

[108] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):357–369, 2001.

[109] Bran Selic. What will it take? a view on adoption of model-based methods in practice. *Software and System Modeling*, 11(4):513–526, 2012.

[110] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *International Conference on Principles and Practice of Constraint Programming*, volume 3709 of *LNCS*, pages 827–831. Springer, 2005.

[111] Stefan Staber and Roderick Bloem. Fault localization and correction with QBF. In *Theory and Applications of Satisfiability Testing*, volume 4501 of *LNCS*, pages 355–368. Springer, 2007.

[112] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming*, 76(2):119–135, February 2011.

[113] Patrick D. Terry. *Compilers and Compiler Generators: An Introduction with C++*. International Thomson Computer Press, 1997.

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

[115] Aliki Tsiolakis. Integrating model information in UML sequence diagrams. *Electronic Notes in Theorerical Computer Science*, 50(3):268–276, 2001.

[116] Muhammad Usman, A. Nadeem, Tai-hoon Kim, and Eun-suk Cho. A survey of consistency checking techniques for UML models. In *Advanced Software Engineering and Its Applications*, pages 57–62. IEEE Computer Society, 2008.

[117] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. Using description logic to maintain consistency between UML models. In *«UML» 2003 - The Unified Modeling Language, Modeling Languages and Applications*, volume 2863 of *LNCS*, pages 326–340. Springer, 2003.

[118] Ragnhild Van Der Straeten, Jorge Pinna Puissant, and Tom Mens. Assessing the Kodkod model finder for resolving model inconsistencies. In *Modelling Foundations and Applications*, volume 6698 of *LNCS*, pages 69–84. Springer, 2011.

[119] Allen Van Gelder. Contributions to the theory of practical quantified Boolean formula solving. In *International Conference on Principles and Practice of Constraint Programming*, volume 7514 of *LNCS*, pages 647–663. Springer, 2012.

[120] Dániel Varró. Automated Formal Verification of Visual Modeling Languages by Model Checking. *Software and System Modeling*, 3(2):85–113, 2004.

[121] Bernhard Westfechtel. A formal approach to three-way merging of emf models. In *International Workshop on Model Comparison in Practice*, pages 31–41. ACM, 2010.

[122] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *International Conference on Software Engineering*, pages 314–323. ACM, 2000.

[123] Magdalena Widl. Test case generation by grammar-based fuzzing for model-driven engineering. In *Haifa Verification Conference*, volume 7857 of *LNCS*, pages 278–279, 2012.

[124] Magdalena Widl, Armin Biere, Petra Brosch, Uwe Egly, Marijn Heule, Gerti Kappel, Martina Seidl, and Hans Tompits. Guided merging of sequence diagrams. In *Software Language Engineering*, volume 7745 of *LNCS*, pages 164–183. Springer, 2013.

[125] Shao Jie Zhang and Yang Liu. An automatic approach to model checking UML state machines. In *Secure Software Integration and Reliability Improvement Companion*, pages 1–6. IEEE Computer Society, 2010.

Magdalena Widl

Curriculum Vitae

✉ *lena@crespo-widl.at*
www.kr.tuwien.ac.at/staff/widl

Education

Jan 11 – now **PhD Computer Science**, *Vienna University of Technology (TUW)*, Vienna, Austria.

Sep 13 – Dec 13 **Research Visit**, *National Taiwan University*, Taipei, Taiwan.

Oct 07 – June 10 **MSc Computational Intelligence**, *TUW*, Vienna, Austria, with distinction.

Sep 06 – June 07 **Student Exchange**, *University of Alicante*, Alicante, Spain, Erasmus scholarship.

Oct 03 – Dec 06 **BSc Business Informatics**, *TUW*, Vienna, Austria.

Employments

Jan 11 – ongoing **Research Assistant**, *TUW, Institute of Information Systems*, Vienna, Austria.
- WWTF project "Formalizing and Managing Evolution in Model-driven Engineering"
- the FWF funded national research network "Rigorous Systems Engineering"

Oct 07 – Dec 08 **Software Developer**, *United Nations Office on Drugs and Crime (UNODC)*, Vienna, Austria.

Okt 05 – Aug 06 **Software Developer**, *UNODC and United Nations Industrial Development Organization*, Vienna, Austria, part-time.

Nov 02 – Sep 05 **Project Administrator**, *Environmental Software and Services GmbH*, Gumpoldskirchen, Austria, part-time.

Grants and Awards

Sep 14 *Best Paper Award* at the 7th International Conference on Software Language Engineering in Västerås, Sweden.

Sep 13 *Stipendium für kurzfristige wissenschaftliche Arbeiten im Ausland* (research visit grant) from the International Office, TUW.

June 11 *Förderungsstipendium* (research grant) from the Faculty of Informatics, TUW.

Dec 10 *Distinguished Young Alumnus/Alumna* for the best master thesis in winter semester 2010 awarded by the Faculty of Informatics, TUW.

Apr 10 *Special Recognition Award* for the poster exhibited at the Junior Scientist Conference 2010 in Vienna, Austria.

Teaching

Mar 12 – June 12 **Introduction to Knowledge-Based Systems**, *TUW*.

Mar 12 – June 12 **Introduction to Artificial Intelligence**, *TUW*.

Apr 05 – May 06 **Introduction to Linux**, *TUW*.

Scientific Activities

Conference talks Software Language Engineering 2014, 2012, Logic for Programming Artificial Intelligence and Reasoning 2013, International Workshop on Quantified Boolean Formulas 2013, Haifa Verification Conference 2012 (poster), Tests & Proofs 2012, International Conference on Model Driven Engineering Languages and System (MoDELS) Doctoral Symposium 2011, Models & Evolution 2011, Junior Scientist Conference 2010, Hybrid Metaheuristics 2010, Metaheuristics International Conference 2009.

Reviewing International Conference on Current Trends in Theory and Practice of Computer Science 2013, International Conference on Theory and Applications of Satisfiability Testing 2015, 2013, MoDELS 2014, 2011, International Conference on Business Process Management 2013, 2012, 2011, International Conference on Model Transformation 2012, International Joint Conference on Artificial Intelligence 2011, MoDELS Doctoral Symposium 2012.

Peer-reviewed Publications

SAT Application P. Kaufmann, M. Kronegger, A. Pfandler, M. Seidl, M. Widl: A SAT-based Debugging Tool for State Machines and Sequence Diagrams. In *Computer Languages, Systems & Structures*, vol. 44, p. 72–88, 2015.

P. Kaufmann, M. Kronegger, A. Pfandler, M. Seidl, M. Widl: A SAT-based Debugging Tool for State Machines and Sequence Diagrams. In *Software Language Engineering*, LNCS 8706, p. 21–40, Springer, 2014. Awarded "Best Paper".

P. Kaufmann, M. Kronegger, A. Pfandler, M. Seidl, M. Widl: Global State Checker: Towards SAT-Based Reachability Analysis of Communicating State Machines. In *Workshop on Model-Driven Engineering, Verification, and Validation*, CEUR 1069, p. 31–40, CEUR-ws.org, 2013.

P. Brosch, M. Seidl, and M. Widl: Semantics-Aware Versioning Challenge: Merging Sequence Diagrams along with State Machine Diagrams. In *Softwaretechnik-Trends*, vol. 33, 2013.

M. Widl, A. Biere, P. Brosch, U. Egly, M. Heule, G. Kappel, M. Seidl, and H. Tompits: Guided Merging of Sequence Diagrams. In *Software Language Engineering*, LNCS 7745, p. 164–183. Springer, 2012.

QBF Certification

V. Balabanov, J.-H. R. Jiang, M. Janota, M. Widl: Efficient Extraction of QBF (Counter)models from Long-Distance Resolution Proofs. In *Proceedings of the Twenty-Ninth Conference on Artificial Intelligence*, p. 3694–3701, AAAI Press, 2015.

V. Balabanov, J.-H. R. Jiang, M. Janota, M. Widl: Efficient Extraction of QBF (Counter)models from Long-Distance Resolution Proofs. In *International Workshop on Quantified Boolean Formulas, Informal Workshop Proceedings*, 2014.

V. Balabanov, M. Widl, J.-H. R. Jiang: QBF Resolution Systems and their Proof Complexities. In *Theory and Applications of Satisfiability Testing*, LNCS 8561, p. 154–169, Springer, 2014.

U. Egly, F. Lonsing, M. Widl: Long-Distance Resolution: Proof Generation and Strategy Extraction in Search-Based QBF Solving. In *Logic for Programming Artificial Intelligence and Reasoning*, LNCS 8312, p. 291–308, Springer, 2013.

U. Egly, M. Widl: Solution Extraction from Long-distance Resolution Proofs. In *International Workshop on Quantified Boolean Formulas, Informal Workshop Proceedings*, 2013.

Evolution of Software Models

M. Widl: Test Case Generation by Grammar-based Fuzzing for Model-driven Engineering. *Haifa Verification Conference*, LNCS 7857, p. 278–279, Springer, 2012.

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 *Tests and Proofs*, LNCS 7305, p. 149–155. Springer, 2012.

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 *MoDELS Workshops*, LNCS 7167, p. 246–256. Springer, 2011.

Metaheuristics

M. Widl and N. Musliu: The break scheduling problem: complexity results and practical algorithms. In *Memetic Computing*, vol. 6, nr. 2, p. 97–112, 2014.

M. Widl and N. Musliu: An Improved Memetic Algorithm for Break Scheduling. In *Hybrid Metaheuristics*, LNCS 6373, p. 133–147. Springer, 2010.

M. Widl, N. Musliu, and W. Schafhauser: A Memetic Algorithm for a Break Scheduling Problem. *Metaheuristics International Conference*, 2009.

November 17, 2015