FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

New Model Checking Techniques for Software Systems Modeled with Graphs and Graph Transformations

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Sebastian Gabmeyer
Matrikelnummer 0301025

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: O. Univ. Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel
Betreuung: Assist.-Prof. Dipl.-Ing. Dr.techn. Martina Seidl

Diese Dissertation haben begutachtet:

(Prof. Dr. Gerti Kappel) (Prof. Dr. Martin Gogolla)

Wien, 15.06.2015

(Sebastian Gabmeyer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

New Model Checking Techniques for Software Systems Modeled with Graphs and Graph Transformations

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Sebastian Gabmeyer
Registration Number 0301025

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: O. Univ. Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel
Advisor: Assist.-Prof. Dipl.-Ing. Dr.techn. Martina Seidl

The dissertation has been reviewed by:

(Prof. Dr. Gerti Kappel) (Prof. Dr. Martin Gogolla)

Wien, 15.06.2015

(Sebastian Gabmeyer)

Erklärung zur Verfassung der Arbeit

Sebastian Gabmeyer
Eduard Meder Gasse 29/3, 2514 Wienersdorf

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 Verfasser)

Acknowledgments

This thesis had not come into existence without the help of many whom I would like to thank in the following. I would like to offer my deep gratitude to my supervisor Prof. Dr. Gerti Kappel for her continuing support, professional guidance, and her thought-provoking questions that challenged me towards a deeper understanding of the problems I intended to solve. In the same spirit, I would like to express my very great appreciation to Dr. Martina Seidl for her invaluable and constructive suggestions, and the insightful discussions that formed the foundations of this thesis. My grateful thanks are also extended to Prof. Dr. Martin Gogolla for sharing his deep knowledge on OCL and his helpful suggestions during the finalization of this thesis. I would like to thank my colleagues at the Business Informatics Group for their supportive feedback and stimulating discussions on all matters of model-driven software development. I am particularly grateful for the assistance given by Dr. Petra Brosch who provided me with valuable feedback and suggestions at all times. I would also like to extend my thanks to Robert Bill whose tireless work on the implementation of the MocOCL model checker enabled us to evaluate our research on an entirely new level. I also wish to acknowledge the help of Dr. Manuel Wimmer for the advice he provided, his support in all teaching related matters, and his interest in my work in all the years we have worked together. Moreover, I would like to offer my special thanks to Katja Hildebrandt and Michael Schadler, who continuously assisted me with all administrative and technical issues.

Finally, I wish to thank my wife, Denise, and my children, Tabea and Theo, for being a constant source of motivation and for their patience. My special thanks extend to my parents for their unrelenting and generous support.

Abstract

In today's software, no matter how security and safety critical it may be, defects and failures are common. With the rising complexity of software and our growing dependency on its correct functioning as it permeates our every day life the software development process requires new approaches to integrate formal verification techniques. This thesis presents approaches on efficiently verifying software systems described by model-driven software development artifacts. These artifacts comprise the implementation of the system and consist of both structural and behavioral models. We present two model checking approaches, MocOCL and Gryphon, to verify the temporal specification of a system against its model-based implementation. Central to our approach is the twofold use of *graphs* to describe the system under verification. First, we describe the admissible static structure of an instance of the system by means of attributed *type* graphs with inheritance and containment relations, often referred to as *metamodel*. Second, we represent a state of the system as an *object* graph that enumerates a system's active objects, the references among them, and their attribute values. A change in the system, e.g., the deletion of an object or the modification of an attribute value, triggers a state change. The behavior of the system is thus described by actions that modify the state of the system. In this thesis we employ *graph transformations* to model such state-changing actions as they provide suitable means to formally describe modifications on graphs. The specification of the system, on the other hand, is written in our temporal extension of the Object Constraint Language (OCL) that is based on Computation Tree Logic (CTL). A specification written in our CTL extension for OCL, called cOCL, can be verified against a model-based implementation of the system with our explicit-state model checker MocOCL. Gryphon aims to increase the efficiency and scalability of the verification process and implements a symbolic model checking approach, that focuses the verification on safety specifications.

The work presented in this thesis also encompasses a survey and a feature-based classification of verification approaches that can be used to verify artifacts of model-driven software development. Methodologically, it provides the motivation for our work on MocOCL and Gryphon. Both our approaches are novel in their own respect; MocOCL for its capability to verify CTL-extended OCL expressions and Gryphon for its use of relational logic to build a symbolic representation of the system that can be verified with any model checker participating in the Hardware Model Checking Competition. Finally, MocOCL and Gryphon are evaluated performance-wise on a set of three representative

benchmarks that demonstrate the model checkers' preferred fields of application and its limitations.

Kurzfassung

Die steigende Komplexität von Software und der gleichzeitig wachsenden Abhängigkeit unserer Gesellschaft von ihrer korrekten Funktionsweise erfordert die Integration von formalen Methoden in den Softwareentwicklungsprozess. In dieser Disseration werden daher Ansätze und Techniken vorgestellt um Softwareartefakte zu verifizieren, die im Zuge eines modellgetriebenen Entwicklungsprozess erstellt werden. Im Konkreten untersucht die Arbeit Ansätze zur Verifikation von Struktur- und Verhaltensmodellen. Damit einhergehend werden zwei Model Checker, nämlich MocOCL und Gryphon—vorgestellt, die die modellbasierte Implementierung eines System gegen deren temporale Spezifikation verifizieren. Graphen nehmen in diesen beiden Ansätzen eine zentrale Rolle ein, da sie sowohl zur Beschreibung der Struktur des Systems als auch zur Beschreibung des Verhaltens des Systems verwendet werden. Während mit Hilfe von attributierten *Typgraphen*, oft auch als *Metamodelle* bezeichnet, welche Vererbungs- und Kompositionsbeziehungen enthalten, die zulässige Struktur des Systems erfasst wird, dienen Objektgraphen zur Beschreibung des Zustandes des System und listen zu diesem Zweck alle aktiven Objekte, die Beziehungen zwischen diesen und deren aktuelle Attributwerte auf. Eine Veränderung des Systems, welche beispielsweise durch das Löschen eines aktiven Objekts oder die Modifikation eines Attributs hervorgerufen wird, hat einen Zustandswechsel zur Folge, der sich in der Struktur des Objektgraphens widerspiegelt. In dieser Dissertation wird das Verhalten eines Systems mit *Graphtransformationen* modelliert, die es ermöglichen Modifikationen an Graphen formal zu beschreiben. Für die Spezifikation des Systems wurde cOCL im Rahmen dieser Dissertation entwickelt. Dabei handelt es sich um eine temporale Erweiterung der Object Constraint Language (OCL), die Operatoren der Computation Tree Logic (CTL) in OCL integriert. In weitere Folge kann nun die Richtigkeit eines Systems, dessen Struktur mit Hilfe von attributierten, getypten Graphen mit Vererbungs- und Kompisitionseziehungen beschrieben wird und dessen Verhalten mit Graphtransformationen modelliert wird, in Hinblick auf seine in cOCL formulierte Spezifikation mit dem Model Checker MocOCL verifiziert werden. Mit Gryphon wurde ein auf Performance und Skalierbarkeit ausgerichteter Model Checker entworfen, mit dem *Safety*-Spezifikationen effizient verifiziert werden können.

Zusätzlich werden in dieser Dissertation eine Studie und eine merkmalbasierte Klassifikation von Verifikationsansätzen vorgestellt, die die Verifikation von modellbasierte Softwareimplementierungen ermöglichen. Diese Studie und die daraus resultierende Klassifikation bildeten aus methodischer Sicht die Motivation für die Arbeiten an MocOCL und Gryphon, die in dieser Hinsicht beide einen neuartigen Ansatz beisteuern. Außerdem beinhaltet die Arbeit eine umfassende Performance-Evaluierung von MocOCL und Gryphon, die anhand von drei repräsentativen Benchmarks Stärken und Einschränkungen der Model Checker aufzeigt.

Contents

1 Introduction **1**

2 Background **7**
2.1 Model-Driven Development 7
2.2 Formal Methods in Software Engineering 15
2.3 Summary 20

3 Verification Approaches for Behavioral and Temporal Aspects in MDD **21**
3.1 Description of Verification Approaches 23
Theorem Proving 23
Model Checking of Rewriting-Based Systems 27
Model Checking of OCL Specifications 35
Model Checking of UML Diagrams 39
3.2 A Feature-Based Classification 43
3.3 Summary 49

4 Model Checking CTL-extended OCL Specifications **51**
4.1 Preliminaries 54
4.2 The CTL-extended Object Constraint Language 64
4.3 MocOCL — A model checker for cOCL specifications 67
4.4 MocOCL in Action 71
4.5 Summary 74

5 Symbolic Model Checking of Safety Properties with Gryphon **77**
5.1 Preliminaries 78
5.2 Translating EMF Models and Graph Transformations to Sequential Circuits 84
5.3 Summary 99

6 Evaluation **101**
6.1 The Pacman Game 102
6.2 The Dining Philosophers Problem 108
6.3 Verification of Interlocking Railway Systems 114
6.4 Summary 123

7 Conclusion **125**

Bibliography **127**

CHAPTER 1

Introduction

In model-driven software development (MDSD) developers employ models and model transformations to implement a software system. This dissertation is concerned with techniques to assert the quality and correctness of model based software implementations. The main contributions of this dissertation are as follows:

(i) an in-depth survey of verification approaches for behavioral aspects of models and model transformations,

(ii) a novel temporal extension for the Object Constraint Language (OCL) to specify behavioral properties of model based software and a model checker, called MocOCL, to verify these temporal OCL expressions, and

(iii) a new symbolic model checker, named Gryphon, that encodes models and descriptions of their behavior into bounded, first-order relational logic [95].

Motivation. During the last decades the importance of software in everyday life has grown apace [125]. With the increasing demand on more sophisticated functionality, however, the complexity of software rises considerably [8,53,141]. As a direct consequence of this increasing complexity, developers make mistakes and "introduce" defects into the software/system[1] [201]. A defect causes malfunctions and errors in the running program, ultimately leading to its failure in the worst case [201]. Failures in software may have far reaching consequences and can result in high fixing costs, devastating physical damage, and even loss of life. The list of severe failures caused by software defects ranges from a failed spacecraft launch [123] over an erroneous floating-point unit in an Intel processor [142], a state-wide electrical blackout [163], and a "bleeding" security protocol [49] to the death of at least five people by an improperly calibrated radiation

[1]We will most often use the terms *software* and *system* synonymous throughout this dissertation.

therapy machine [120], to name a few examples. Thus, a software's correct functioning is vital and a matter of utmost importance.

To counter the ever growing complexity of nowadays software, graphical and textual modeling languages, like the Unified Modeling Language (UML) [152], began to permeate the modern development process. The reason for this development is twofold; first, models abstract away irrelevant details and, second, they express ideas and solutions in the language of the problem domain. In the context of model-driven software development model transformations take a pivotal role. Their use and shape are manifold [50] and their field of application includes, among others, *model-to-text* transformation, that may be used, e.g., to generate executable code from models, and *model-to-model* transformations. The latter group can be divided into *endogenous* and *exogenous* model-to-model transformations. Exogenous model transformations describe translations between two types of models, where the so-called *source* model is converted into a *target* model of a different type. An example of an exogenous model transformation is the well-known and often portrayed object/relational transformation that maps a class diagram to an entity relationship diagram.

While exogenous model transformations perform a conversion between two types of models, endogenous model-to-model transformations modify or refine models, that is, source and target model of an endogenous transformation are of identical type. In this respect, endogenous model transformations are used to define computations on models, where the source model represents the *current state* and the target model the *next state* of the system. Hence, endogenous model transformations can capture the behavior of a system. The case of endogenous model transformations will be the focus of this dissertation.

Among the multitude of available model transformation languages, the theory of graph transformations [62,173] offers a formal, concise, and mathematically well-studied language to describe modifications on graphs. In the following we will thus assume that models are expressed in terms of attributed, typed graphs with inheritance and containment relations [24] and model transformations in terms of double-pushout (DPO) graph transformations [47]. In the context of model-driven software development a system may thus be formally described by graphs that define its static structure and graph transformations that capture the system's behavior.

Problem Statement. Graph transformations offer a Turing complete model of computation [128] and are as such as expressive as any other conventional programming language. Consequently, verification techniques that assert the functional quality of model based software are required to trace and eliminate defects in modeling artifacts, i.e., models and model transformations or, likewise, graphs and graph transformations. To increase the acceptance of such a verification technique it should

(i) allow users to formulate their verification tasks in the language of the problem domain they are familiar with,

(ii) present the result of the verification in the language of the problem domain, and

(iii) operate as automatic and efficient as possible.

Solution. The verification techniques presented in this dissertation are kept transparent to the user. This is achieved by allowing the user to formulate the verification problem, which consists of a formal representation of the system and its temporal specification, directly in the language of the models. Based on an extensive survey of existing verification techniques for model based software and a classification thereof we focus on model checking based verification techniques that provide a viable trade-off between automatic executability, expressivity of the supported specification languages, and verification efficiency. In brief, a model checker explores all or relevant parts of a system's finite state space, which encompasses all the states a system might be in during its execution, and checks whether a system's specification holds in each state. The scalability of a model checking based verification approach thus crucially depends on the representation of the states in memory. Originally, states were represented explicitly and stored in tables, but more recently symbolic approaches have been proposed that represent states and transitions between states by means of first-order or propositional logic formulas. While the latter can verify larger state spaces more efficiently, the former provides a faithful representation of the system that might not be preserved in every detail by a symbolic encoding. The specification against which the model checker verifies the system is usually formulated in temporal logic, most commonly, in either linear temporal logic (LTL) [158] or computation tree logic (CTL) [42], the latter of which is computationally easier to verify. To allow the users to express the specification in the language of the models and domain they are working with we

(i) extend the Object Constraints Language (OCL), which enjoys broad acceptance among MDSD users, with temporal operators to verify properties that are more complex than those already expressible in OCL, i.e., invariants and pre- and post-conditions of operations, and

(ii) use graph constraints to model desired and undesired states of the system.

Based on these considerations the presented solution encompasses an explicit and a symbolic model checker that verify specifications formulated either as CTL extended OCL expression or as graph constraints, respectively.

Contributions. The contributions of this dissertation are threefold and I present henceforward

(i) a survey and a feature based classification of existing, state-of-the-art verification techniques that assert the correctness of behavioral MDSD artifacts,

(ii) a branching time extension of the Object Constraint Language based on CTL and an explicit state model checker, called MocOCL, that verifies a system against a specification of CTL extended OCL expressions, and

(iii) a novel symbolic model checker, called Gryphon, that asserts the unreachability of bad states modeled as graph constraints.

Before we embarked work on the presented model checkers, MocOCL and Gryphon, we conducted an in-depth survey of existing verification approaches that analyzed and verified the behavior of systems represented by models and model transformations. This survey was condensed into a feature based classification of the different approaches that, on the one hand, allows to select suitable verification approaches given a set of requirements like supported input artifacts or the specification language and, on the other hand, allows to pin point verification approaches that have not been studied extensively yet. In this respect, the classification revealed that, although several approaches promote the use of rich temporal OCL specifications, none of these can be integrated into the formal semantics of OCL without modification and, moreover, none of them provides an accompanying model checker. With our CTL based extension of OCL, called cOCL and our model checker MocOCL that verifies cOCL expressions we intend to close this gap. Further, the classification shows that currently only a few symbolic approaches have been proposed and implemented. None of these few approaches, however, is capable to exploit the power of existing, industrial grade hardware model checkers. We thus present our model checker Gryphon that uses bounded, first-order relational logic to derive a symbolic, propositional logic representation of the system that can be verified by any model checker participating in the Hardware Model Checking Competition [21]. Finally, we perform a thorough performance evaluation, where we compare MocOCL and Gryphon with the state-of-the-art model checker integrated into the Groove tool [106].

Structure of the Dissertation. The next chapter presents the prerequisites and background material for the remainder of this dissertation. It introduces, in the first part, the model-driven software development paradigm, the Unified Modeling Language (UML), the Meta Object Facility (MOF), and the Object Constraint Language (OCL). The second part of the Chapter discusses formal system representations and formal verification techniques, and introduces, among others, the theory of graph transformations and model checking. Then, Chapter 3 presents our survey and classification of existing approaches that solve verification problems concerned with behavioral aspects encountered in the context of MDSD. The Chapter starts with a detailed discussion of the existing verification approaches followed by the presentation of our classification and the resulting feature model. The Chapter closes with a short discussion on the insights that we gain from our survey and classification. The next two chapters are devoted to our model checkers, MocOCL and Gryphon. Chapter 4 presents syntax and semantics of our CTL extension for OCL, called cOCL, and introduce the model checker MocOCL. Then, Chapter 5 develops the necessary translations to encode a type graph and a set of behavior describing graph transformations into a symbolic representation and discusses the internals of Gryphon. Finally, an evaluation of MocOCL, Gryphon, and Groove against three benchmarks is presented in Chapter 6. Chapter 7 concludes this dissertation and presents an outlook on possible future directions.

Remarks. The enumerative, explicit-state model checker MocOCL was implemented by Robert Bill in the course of his master's thesis [25] that I co-supervised.

CHAPTER 2

Background

In the following, we introduce the common terminology used in subsequent sections. First, we discuss the notion of models and modeling, position them in the context of software development, and highlight their importance to model-driven development. Then we shortly review the formal verification techniques relevant for this work.

At this point we want to emphasis that the term *model* is heavily overloaded in computer science. For example, in software engineering, models are design artifacts for describing the structure and behavior of software, whereas a model in logic usually provides a set of variable assignments for some formula f such that f evaluates to *true*, i.e., it is *satisfied* under this assignment. As we use both kinds of models in this thesis, we distinguish between *software models* and *logical models* and use these terms explicitly whenever ambiguities might arise.

2.1 Model-Driven Development

The development of software is an intricate task that requires mechanisms of abstraction to decrease its complexity [53, 102, 108]. The introduction of models to the development process, whose primary purpose is undoubtedly their ability to simplify complex concepts by abstraction, allows the developers to focus on a distinct subset of the problem by representing the system[1] under development with multiple, overlapping views. This in turn distributes the complexity involved in the development of the system among multiple, overlapping models, each of which represents a distinct view onto the system.

In model-driven development (MDD), models become the center of all development efforts. This contrasts code centric development methods where models are usually used only in the initial requirement gathering and design phases and later as a documentation of the implemented system. Most often, however, the models lag behind the actual

[1]A *system* may be anything that was built, designed, or developed for a specific purpose ranging from hardware components and software as atomic building blocks to composite structures and installations as the ventilation system in a factory or the factory itself.

status quo of their implementation, effectively rendering their documentation purposes obsolete. In MDD, on the other hand, an initial model of a system under development is refined in multiple iterations and eventually translated into the final executable program and other deliverables by so-called *model transformations.* The lifting of models to first class development artifacts thus goes hand in hand with the pivotal role of model transformations [180]. Model transformations have various fields of application in MDSD and may not only be used in the process of generating deliverables but also to describe computations on models or, similarly, to implement the behavior of a model by making explicit the effects on the model of an operation call of the system [50].

An important concept in MDSD is that of a *metamodel* that defines the language, or visually speaking, the building blocks that are allowed to be used in building a so-called *instance model* of the metamodel. A model *conforms to* the metamodel if it adheres to the structure prescribed by the metamodel. Similar to the relation between metamodel and instance model, a *meta-metamodel* defines the necessary conformity constraints for a metamodel. Each additional *meta*-level simplifies the set of available language features, or building blocks, until a core of concepts remains that is able to recursively define itself. Thus, the model at some meta-level conforms to itself, effectively bootstrapping the conformance relation. Note that we will follow the common convention that refers to an instance model simply as the *model* of some metamodel. In this way, a model of a meta-metamodel is a metamodel or, strictly speaking, an instance model of the meta-metamodel.

Generally, we distinguish between *descriptive* and *prescriptive* models, where the former *describe* an existing system and the latter *specify* what a system should look like; hence, it is also referred to as a *specification model.* This distinction leads to different notions of correctness. A descriptive model is correct if all the reified concepts correspond to actual observation of the existing system. In contrast, the system is deemed correct if it implements all the concepts found in the prescriptive model. Stated differently, the system is correct if it satisfies the specification defined by the prescriptive model. In the remainder of this dissertation, we will assume that all models are prescriptive models.

Model-Driven Architecture

In 2001, the *Object Management Group*[2] (OMG) proposed the *Model-Driven Architecture* (MDA) as a standardized architecture for MDD. Central to the MDA perspective is the separation of the model based implementation into a platform independent and a platform specific model. The ratio behind this separation is that platform independent models (PIM) is platform agnostic and does not change if the underlying, anticipated platform that provides the run-time environment for the system under development is exchanged in favor of another. Instead, the PIM is parameterized and translated into platform specific model (PSM) with a set of model transformations. The executable system is then generated from the PSM [154].

[2]`http://www.omg.org`

The realization of the MDA depends on various other standards issued by the OMG, most notably the Meta Object Facility [150] (MOF), the Unified Modeling Language [153] (UML), the Object Constraint Language [148] (OCL), and the XML Metadata Interchange [151] (XMI) standards.

The Meta Object Facility

The *Meta Object Facility* (MOF) [150] forms the basis for OMG's model driven architecture (MDA) initiative. The MOF 2 standard defines a common interchange format and interoperability layer for the modeling and exchange of metadata about models. In an MDA context, the MOF Model, that is defined by the standard, is intended to form the basis for all platform independent models, for which it provides a common metamodeling framework. For this purpose, MOF provides a reduced set of class modeling constructs, in essence a subset of UML's modeling constructs as MOF re-uses essential parts of the UML 2 infrastructure library, and a reflection mechanism that allows dynamic access to the components and features of a MOF based model. A MOF *based model* denotes any instance model that conforms to MOF Model, which is defined recursively by itself [150].

The MOF standard is split into two sub-standards, Essential MOF (EMOF) and Complete MOF (CMOF), of which we only discuss the former in the following.

EMOF provides a simplistic set of class modeling constructs offering *classes* that are composed of *properties* and *operations*. A property is either an *attribute* or an *association*, which are by default uni-directional. To model bidirectional association two *opposite* properties are required. As a property is a *structural feature*, it is further possible to specify whether a property is single or multi-valued by specifying lower and upper bounds. A class can be part of multiple inheritance hierarchy through *generalizations*. An operation is *parameterized* and may raise *exceptions*. The structure of EMOF is aligned to XMI to allow a *straightforward* serialization of MOF based metamodels [150].

The Unified Modeling Language

The Unified Modeling Language (UML) [153] standard defines a general purpose, object-oriented modeling language together with a set of diagrams that provide different views on the described system. These views are overlapping in the sense that a view contains information that may, in some parts, be derived from another view. UML was designed to be platform, programming language, and tool agnostic which emphasizes its universal character. The UML standard consists of two parts, the UML *infrastructure* [152] that specifies the metamodel of UML and the UML *superstructure* [153] that defines the *user level* modeling constructs atop of the infrastructure. In the following we present a subset of the available diagrams available in UML that are relevant for the subsequent chapters.

Class and Object Diagrams. A *class diagram* describes the static structure of the system under development, and defines the kind of *objects* and the relations, called *links*, among them. The common characteristics of an object are captures by a *class*. A class encapsulates *structural features*, which can be, among others, *attributes* and *associations ends*, and *behavioral features*, i.e., operations. A feature may only be accessed by objects that are within the scope defined by the feature's *visibility.* A feature's visibility can be set to either *public*, *package*, *protected*, or *private.* A structural feature is typed over either a *classifier*, e.g., a class or an interface, a *primitive data type*, or an *enumeration.* Further, a structural feature may carry multiplicities, which allow, for example, the specification of multi-valued attributes that describe arrays or lists of values [178]. An *operation* defines the behavior of the class and its specification consists of a set of typed parameters, a typed return value, and a set of exceptions it may raise. A class can be connected to other classes by *associations* that describe a relation or interaction between the participating classes [178]. An association may also express *part-of* relations, so-called *composite aggregations*, which demand that each part belongs to exactly one composite and all parts, which may be composites themselves and host other parts, are deleted in a cascading manner if their composite is deleted. If a part can be part of more than one composite, a *shared aggregation* may be used to model this relation [178]. Moreover, an *is-a* relation, referred to as a *generalization*, expresses that a class refines its more general superclass. Hereby, the subclass *inherits* all non-private features and associations of its superclass and may add additional attributes, operations, and associations. Classes are marked *abstract* if they capture common features but are not intended to be instantiated. Similarly, a non-instantiable class that describes common behavioral features, called a contract, may be marked as an *interface.* The UML standard does not prescribe whether a class with multiple generalizations is implemented using multiple inheritance relations as in C++ or multiple interfaces as in Java.

An object *instantiates* a class by assigning values to attributes and by establishing *links*, which are instantiations of associations, to other objects. The object diagrams captures such an instantiation of the system, which is also referred to as a specific *state* of the system.

Sequence Diagrams. The Sequence Diagram captures inter-object *interactions* based on the exchange of *messages* between the *interaction partners* of a system [153]. An interaction partner may be any participant or component of a system, for example, a user, an agent, a sensor, or an object in a software system, to name but a few possibilities. Each interaction partner is represented by a *lifeline.* A lifeline represents a sequence of events like send and receive events of messages, and time events. The primary purpose of a sequence diagrams is the study of the sequence of the exchanged messages, called a *trace*, that represents a specific behavior of the system under development. Even though data flow may be modeled explicitly it is not the primary goal of the sequence diagram to describe how data is used and altered along a trace [178].

The UML standard distinguishes between *synchronous messages* that wait for a *receive message* before the interaction continues, *asynchronous messages*, and *create* and *termination messages* that start or end of an interaction partner [153]. Further, *combined fragments* allow a concise specification of the control flow of an interaction. The standard offers, among others, *loop* and *alt* fragments, which loop or branch an interaction on a certain condition, *strict* and *par* fragments, which enforce a strict ordering on the messages or allow an interleaved ordering, and fragments that *forbid* or *assert* the occurrence of certain interactions.

State Machine Diagrams. The different *states* that an object may visit during its *lifetime* are captured by a behavioral state machine diagram, which are based on David Harel's state charts [85]. A state machine consists of a finite number of named *states* that an object of a class may be in and *transitions* among these states that describe order and occurrence of a state change. A state may specify a number of activities that must be executed on *entry*, while the state is *active*, on *exit*, or any combination thereof. The UML standard defines a number pseudo states, among these, the *initial state* that defines the entry point of the state machine diagram, *decision nodes* that branch to either one of set of successor states depending on the evaluation of a guard condition, *terminate nodes* that indicate abortion. In contrast to the initial state, the *final state* is not a pseudo state as the object may reside indefinitely in the final state [178]. A transition can be labeled with an *event*, a *guard*, and a list of *activities* that are executed if (i) the specified event occurs and (ii) the guard is satisfied at the time the event occurs. If no event is specified, the transition is triggered once all activities specified for a state are completed, which fires a *completion event* [178].

Activity Diagrams. The activity diagram models processes, functions, and operations, or, more generally, behavior, at different levels of granularity; it offers modeling constructs to describe accurately the sequence of computations that an operation of a class preforms and, at the same time, to capture the high-level description of a business process [178]. The main building blocks of an activity diagram are the *activities* and the directed edges that model the *control* and *data flow* between these activities. Similar to the operations that they can describe, activities may have several input and several output parameters. Moreover, their execution may be restricted to start in states that satisfy a certain *precondition* and end in states that fulfill the specified *postcondition.* An activity may in turn consist of a sequence of atomic *actions.* Similar to state machine diagrams, the UML standard provides a set of predefined elements to describe the control flow. Among these are the *initial node*, that marks the beginning of the control flow, the *decision node*, whose outgoing edges are guarded to model different branches of execution, the *merge nodes*, that join two or more branches, *parallelization* and *synchronization nodes*, that model the parallel and sequential execution of activities, and the *final node*, that marks the end of all, possibly parallel, control flows.

The control and data flow between actions and activities of an Activity diagram follow the token semantics introduced for Petri nets [139,157]. Hereby, an activity may only be executed if (i) the activities of all incoming edges possess a token, and (ii) if the guards on these incoming edges are satisfied. If an activity has multiple outgoing edges, it splits the token and places one token on each outgoing edge. In addition to controlling the flow of activities, a token may also carry data. The data flow between activities is modeled implicitly through input and output parameters and modeled explicitly with *object nodes* inside an activity.

Object Constraint Language

Often, modelers wish to incorporate fine-grained details of a domain into a model that cannot be expressed with standard modeling constructs provided by the OMG's UML and MOF specifications. The *Object Constraint Language* (OCL) [148] aims to fill this gap. It is a formal, declarative, typed, and side-effect free specification language to define invariants and queries on MOF-compliant models as well as pre- and postconditions of operations. An OCL constraint is defined within a *context*, i.e., the element of the model to which it applies, and consists of a constraint stereotype, either `inv`, `pre`, or `post` to declare an invariant, a pre- or a postcondition, that is followed by the OCL expression, which defines the property that should be satisfied/refuted in the context of the constraint. For this purpose, the OCL specification defines a rich library of predefined types and functions. In contrast to many other formal specification languages it has been designed to be user-friendly with regard to readability and intuitive comprehensibility. OCL is heavily used to dissolve ambiguities that arise easily in natural language descriptions of technical details. Thus, it plays an important role in many MDSD projects and OMG standards.

In section 4.1 we will discuss formal syntax and semantics of OCL, which we extend in section 4.3 by a set of temporal operators.

Eclipse Modeling Framework

The Eclipse Modeling Framework is a reference implementation of EMOF and enjoys broad industry acceptance. In EMF the language used to define (meta-)models is called Ecore and the framework provides sophisticated Java code generation facilities for Ecore based models and serialization support based on the XML Metadata Interchange (XMI) standard [151]. In this respect, EMF provides a bridge between Java, XML, and UML [190]. Similar to the MOF Model, Ecore recursively defines itself. Figure 2.1 depicts the class diagram of the Ecore language. An Ecore model consists of an `EPackage` that contains a set of `EClassifiers`. A classifier is either an `EClass` or an `EDataType`. A class may be abstract or an interface and setting either of the two flags to true has the same effect as annotating a Java class with the equivalently named keyword. A class may have an arbitrary number of superclasses and hosts a set of `EStrucuturalFeatures` and a set of `EOperations`, both of which are typed. A structural feature is instantiated either as an `EReference` or an `EAttribute`. A reference denotes an unidirectional association between two classes, its type is given by the class that it points to. Bidirectional associations are modeled with two (unidirectional) references pointing in the *opposite* direction where the containing class of the first reference corresponds to the referenced class of the second reference and vice versa. To indicated that two unidirectional references constitute a bidirectional reference the `eOpposite` association is set to point to the reference that runs into the opposite direction. If a reference acts

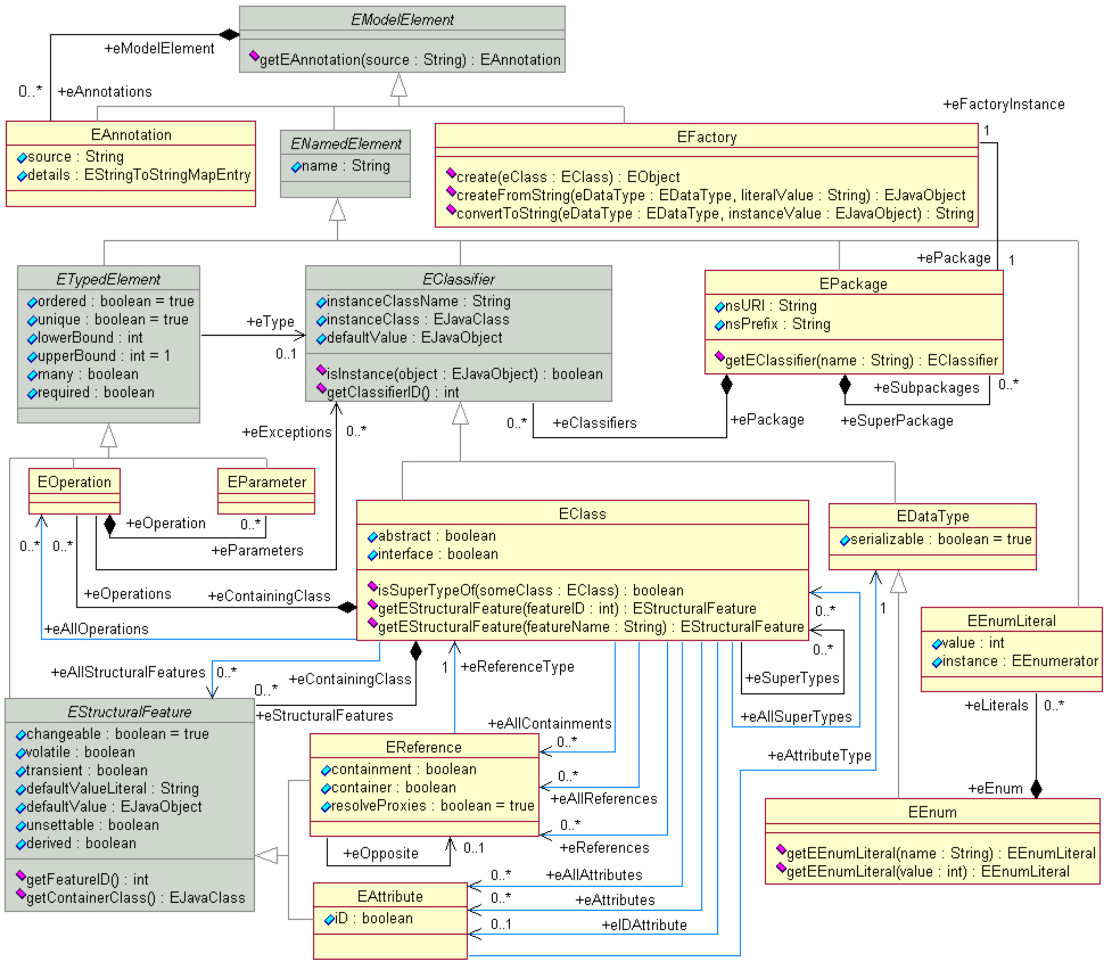
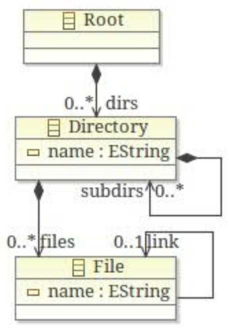


Figure 2.1: Ecore class diagram [60]

as a composition, its `containment` property is set to true. A model that instantiates the *containment reference* is required to ensure that every contained object (i) has exactly one container and (ii) is not transitively contained by itself, i.e., there does not exist a containment cycle. In contrast to references, attributes are typed over `EDataTypes`, which correspond to primitive data types, e.g., `EBoolean`, `EInt`, and `EString`, or to enumerations (`EEnums`). Each of the primitive data types is mapped to its corresponding Java type, i.e., `EBoolean` is mapped to `Boolean`, `EInt` is mapped to `Integer` and so on. Further, if a structural feature is many valued, i.e., its `lower` bound is less than its `upper` bound, it is implemented as a list. A class's operations may be parameterized and have a return type, which is either a class, a primitive data type, or an enumeration. Similar to Java, Ecore defines a single super class for all other classes, namely the `EObject` class. Moreover, each of the above mentioned modeling constructs may be annotated by an



(a) File-system

```
<xmi:XMI xmi:version="2.0"
    ...>
  <fs:Root>
    <dirs name="home">
      <subdirs
          name="user1">
        <files
            name="thesis.tex"/>
        </files>
```

(b) XMI serialization

Figure 2.2: A file-system structure modeled in Ecore and the XMI serialization of an instance model

`EAnnotation`.

Note that each Ecore model has a single root element that transitively contains all other elements; for example, for the Ecore model itself the `EPacakage` class acts as the root container. By introducing such a root node the translation to XMI becomes straightforward because the root tag of the XMI file maps directly to the root node of the Ecore model. Figure 2.2a displays a file-system structure modeled with Ecore where `Root` is the root container of `files` and `dirs` (directories). A `File` may `link` to other files and a `Directory` may have `subdirs` (subdirectories). In Fig. 2.2b, the XMI serialization of a file-system instantiation is depicted. The depicted file-system instance has one top-level directory beneath `Root`, namely `home`, which contains the `user1` directory. The file `thesis.tex` is contained in the `user1` directory.

Model Transformations

Model transformations can express arbitrary computations over models and, thus, take on a pivotal role in MDSD [180]. They are classified, among other characteristics, by their specification language, the relationship between the input model of the transformation and its output model, or whether the transformation is unidirectional or bidirectional [50]. In the subsequent sections we will use the term *source model* to refer to the input model of a transformation and the term *target model* to refer to the transformation's output or resulting model. A *trace model* establishes links between elements of the source and the target model to indicate the effect of the transformation on the respective element. A transformation is *endogenous* if source and target model conform to the same metamodel, it is *exogenous* otherwise. A transformation can be *in-place* meaning that source and target model coincide or *out-place* if source and target model are separate. A transformation that is in-place and exogenous is called *in-situ* transformation and mixes source and target model elements during intermediate steps.

Numerous languages have been proposed to develop model transformations. Two popular choice among these are the *Atlas Transformation Languages* (ATL) [101] and the *Query/View/Transformation* (QVT) standard issued by the OMG [149]. ATL is a rule based, primarily declarative transformation language for Ecore models that also allows to mix in imperative code sections. An ATL transformation rule thus consists of (*a*) a guarded `from` block that matches a pattern in the source model; (*b*) a `to` block that creates the desired elements in the target model once a match for the `from` block is found; and (*c*) an optional `do` block that executes the imperative code. It supports exogenous and endogenous model-to-model transformations. The Query/View/Transformations (QVT) specification is a standard issued by the OMG [149] that embraces the definition of model queries, the creation of views on models, and, in its most general form, the specification of model transformations. It defines three transformation languages for MOF 2.0 models, namely the QVT Relations, the QVT Core, and the Operational Mappings language. Both the QVT Relations and the QVT Core language are declarative transformation language that are equally expressive. QVT Relations supports complex pattern matching and template creation constructs, and it implicitly generates tracing links between the source and the target model elements. While the QVT Relations is designed to be a user-friendly transformation language, QVT Core provides a minimal set of model transformation operations that lead to simpler language semantics without loosing the expressiveness of the QVT Relations language; yet, QVT Core's reduced set operations increases the verbosity of its transformation definition. Further, QVT Core requires an explicit definition of traces between source and target model elements. The QVT standard defines a *RelationsToCore* transformation that can be used either to define the formal semantics of QVT Relations or to execute QVT Relations transformations on a QVT Core transformation engine. The third transformation language, Operational Mappings, is imperative and uni-directional. It extends OCL by side-effecting operations to allow for a more procedural-style programming experience. Operational Mappings are called from a Relations specification that establishes the tracing links.

In the presentation of our verification approach (see chapter 4 and 5) we will use neither ATL nor QVT, but so-called *graph transformations*, which we introduce briefly in the next section and in more detail in section 4.1. Algebraic graph transformations are a particular suitable model transformation framework for our purpose because they offer a formal language for expressing graph modifications. The extension of algebraic graph transformation that can be used to transform MOF-compliant models like Ecore has been presented by Biermann *et al.* [24].

2.2 Formal Methods in Software Engineering

In this section, we introduce formal representations for software models and discuss verification techniques based on theorem proving and on model checking. In the first part we discuss different types of graphs that are commonly employed as formal representations for software models, while the second part discuss formal verification techniques. The section is intended to give an overview and some of the concepts mentioned will be elaborated in greater detail in subsequent chapters. In particular, we describe the theory of algebraic graph transformations and model checking in more depth in chapters 4 and 5.

Formal Representations for Software Models

Due to a lack of formality in OMG standards, which often describe the semantics of modeling languages in prose rather than with formal, mathematical statements, many approaches choose graphs, hypergraphs, Petri nets, or combinations thereof that come with the desired mathematical foundation to represent software models and their semantics formally and unambiguously.

Graphs

Graphs and graph transformations are a popular choice to formally describe models and model transformations. For this purpose the theory of graph transformations has been extended to support rewriting of attributed, typed graphs with inheritance and part-of relations [24]. In the following, we summarize the concepts relevant for this work. A *graph* $G = (V, E)$ consists of a set V of vertices (also: nodes), a set E of edges. Further, we define a source and a target function, $src : E \to V$ and $tgt : E \to V$, that map edges to their source and target vertices. A *morphism* (also: function, map) $m : G \to H$ is a structure preserving mapping between graphs G and H. A graph transformation $p : L \to R$ describes how the left-hand side (LHS) graph L is transformed into the right-hand side (RHS) graph R. Technically, a graph rule $p = (v_G, e_G$ is a pair of morphisms that maps all those vertices and edges from the LHS graph to the RHS graph that are *preserved* by the graph transformation, while all nodes and edges that are not mapped to the RHS by p are *deleted*, and all nodes and edges in the RHS graph for which there exists no mapping from the LHS to the RHS are newly *created.* A graph rule $p : L \to R$ is *applied* to a *host graph* G if there exists a morphism $m : L \to G$, called a *match*, that maps the LHS graph L into the host graph G. The application of rule p at match m rewrites graph G to the *result graph* H. Numerous theoretical frameworks exist that determine how graph H is derived from G. In Section 4.1 we provide an introduction to an algebraic framework called the double pushout (DPO) approach. Other frameworks, for example, based on monadic second-order logic are explained in detail in the handbook [173]. For an in-depth treatment of the DPO approach and some of its extensions we refer the interested reader to the more recent monograph on this topic [62].

Hypergraphs

A *hypergraph* is a graph $G = (V_G, E_G, l_G, c_G)$ with an edge-labeling functions l_G and a connection function c_G that assigns to each edge one or more target vertices, i.e., an edge in a hypergraph may connect to multiple vertices. Similar to rewriting rules on graphs, graph transformations on hypergraphs consist of a LHS and RHS and an injective mapping α between the nodes of the LHS and those of the RHS.

Petri Nets

Petri nets are bipartite graphs that are often used to model concurrent systems and parallel processes [139]. Again, we summarize relevant concepts. A *Petri net* [157] is a place/transition graph $N = (S, T, m_0)$ where places $s \in S$ are connected via transitions $t \in T$. Each transition has zero or more incoming edges and zero or more outgoing edges. A place is an *in-place* (*out-place*) of a transition t if it connects to t over an incoming (outgoing) edge. Petri nets have a token based semantics. A *marking* m assigns tokens to places, and defines the current state of the represented system. The initial state is given by the initial marking m_0. We denote with $m(s)$ the number of tokens assigned to place s. A transition is *enabled* if all its in-places carry a token. Given a marking m_i, an enabled transitions *fires* by removing a token from each in-place and assigning a token to each out-place resulting in a new marking m_{i+1}. If more than one transition is enabled, one is non-nondeterministically chosen to fire. A transition with no in-places (out-places) is always enabled (never enabled). A marking m_j is *reachable* from m_i if there exists a sequence of firing transitions that, starting with m_i result in m_j. A marking m is *coverable* if there exists a reachable marking m' such that $m'(s) \geqslant m(s)$ for every place s in the Petri net. This second property is useful to determine whether the Petri net deadlocks.

Formal Verification Techniques

Even before the first program was run on a computer techniques were developed to ensure that a program executes as intended. Starting with the lambda calculus and Turing machines, research in this field continues along numerous branches bringing forth, among others, Hoare Logic [91] and Dijkstra's guarded commands [54], abstract interpretation [48] as a formal framework to reason over compiler optimizations, model checking [67, 164], and, finally, automated theorem proving for first-order logic and decidable subsets thereof [172]. In the following, we present briefly the two branches most relevant for subsequent chapters, namely model checking and (semi-)automatic theorem proving.

Model Checking

Model checking is an automatic verification technique that asserts the correctness of a system w.r.t. its specification by exhaustively exploring the set of all possible but finitely many system states. Model checking has been successfully applied to some hardware verification problems and, recently, with increasing success also to some software verification problems [99]. In hardware model checking a system is represented by a circuit and a *state* of the system is denoted by the valuations of a circuit's latches. In software model checking, on the other hand, the system is represented a program, i.e., data and operation on the data, and a distinct valuation of the system's variables identifies a *state*. Given an *initial state* of the system, the set of all possible latch or variable valuations that are reachable from the this initial state is referred to as the *state space* of the system. The state space is traditionally represented by a Kripke structure, but also by finite automata and linear transition systems (LTS). A Kripke structure is a finite, directed graph whose nodes represent the states of the system. The nodes are labeled with those atomic propositions that are true in that state. The edges of a Kripke structure represent transitions between their source and their target node and are (usually) unlabeled. In contrast, automata and linear transition systems label transitions with the system's operations that trigger the state change. All of them have in common that they describe execution *paths* of the system, which are defined as, possibly infinite, sequences $\pi = s_1 s_2 s_3 \ldots$ of states $\sigma_i, i \geqslant 0$. We say that a state σ_n is *reachable* from the initial state σ_0 if there exists a finite path $\pi = \sigma_0 \ldots \sigma_n$.

The *specification* is usually formulated in a temporal logic and expresses desired properties of the system. It is most commonly formulated either in Computation Tree Logic (CTL) [42] or in Linear Temporal Logic (LTL) [158]. Both share the same set of temporal operators, namely X (*next*), F (*finally*, also: *eventually*), G (*globally*), and U (*until*). In case of CTL, each occurrence of a temporal operator must be preceded by a *path quantifier*, either A (*on all paths*) or E (*on some paths*), whereas LTL formulas are implicitly all-quantified. Intuitively, the formulas $\mathsf{AX}\,\varphi$, $\mathsf{AF}\,\varphi$, $\mathsf{AG}\,\varphi$, and $\mathsf{A}\,\varphi\mathsf{U}\psi$ are satisfied if, along *all* paths that start in the current state, φ holds in the next state, in some future state (including the current state), in all future states (including the current state), and in all states until ψ holds eventually. It follows that a CTL formula may explore multiple branches due to the requirement that every temporal operator is path-quantified, while an LTL formula branches only once in the state where the evaluation starts.

Temporal formulas describe properties that the system should satisfy and can be categorized into *safety*, and *liveness* properties.[3] Safety properties are characteristically specified by $\mathsf{AG}\,\varphi$ and describe invariants of the system [126] that hold in every state on all paths. They assert that "nothing bad" ever happens. Liveness properties test if "something good" happens eventually or repeatedly and are either of the form $\mathsf{F}\,\varphi$ or $\mathsf{GF}\,\varphi$ [15]. Moreover, *reachability* properties are used to test if there exists a path to a state that eventually satisfies some condition φ. They are of the form $\mathsf{EF}\,\varphi$ [15].

To algorithmically verify a system with a model checker the user supplies both a representation of the system and its specification as input. In its simplest form, the model checker first builds the state space of the system and then evaluates the specification. If the specification is violated, the model checker returns a *counterexample trace* that describes paths to states that falsify the specification. Otherwise, it informs the user that the specification holds.

Model checking is applicable only to finite state representations of systems. The state space may, however, become exorbitantly large, because it grows exponentially

[3]Note that there exist two classification schemes, namely the *safety-liveness* [2] and the *safety-progress* classification [41].

with every additional system variable. This phenomenon is known as the *state explosion problem.* Reducing the number of states and improving the efficiency of the state space's traversal has been the subject of active research for that past 30 years and still is. This line of research has brought forth several techniques that pushed the number of feasibly analyzable states from 10^5 to 10^{20} and beyond. McMillan [130] proposed the first *symbolic model checking* technique in an effort to reduce the space required to store an explicit enumeration of all states, and represented states and transitions with Boolean formulas, which he encoded into (Reduced Ordered) Binary Decision Diagrams (BDD) [37]. Later, Biere *et al.* [20] presented *bounded model checking* (BMC), a symbolic approach that does not require BDDs. It analyzes execution paths of bounded length, thus, offering an efficient technique that is sound, yet not necessarily complete. In a different line of research, the framework of abstract interpretation [48] is employed to represent a set of concrete states by a single abstract state. This overapproximation is conservative, i.e., if a property holds in the abstract system, it holds in the concrete system, too. If, however, the property fails in the abstract system, the returned counterexample trace need not describe a realizable trace in the concrete system. This is due to the overapproximation of the abstract system that may permit execution paths that do not exist in the concrete system. If this is the case, the counterexample that is not realizable in the concrete system is identified as being *spurious.* To eliminate a spurious counterexample it is necessary to refine the abstraction. This refinement procedure may be guided by the returned counterexample and thus performed automatically. This procedure is now known as *counterexample-guided abstraction refinement* (CEGAR) [43].

Theorem Proving

Theorem proving is the task of deriving a conclusion, i.e., the theorem, from a set of premises using a set of inference rules. Traditionally performed *manually*, nowadays, *interactive* proof assistants like ISABELLE/HOL [145], COQ [46], or PVS [156] are often used to aid the trained user in producing machine checked proofs. These proofs are developed interactively. Given a set of premises and a goal, i.e., the desired conclusion, the proof assistant attempts to prove intermediate steps automatically and, if unable to continue, it resorts to the user. The user then guides the proof search by adding new lemmas such that the assistant is finally able to complete the proof. Due to the undecidability of many logics beyond the propositional level, interactive proving strategies are necessary. Sometimes, user guided proof search is considered a limitation. Thus, there exist a number of approaches that (i) either accept non-terminating proof searches or (ii) reduce the expressivity of the logic to a decidable subset and thus achieve full automatism that requires no user guidance. Automatic theorem provers like VAMPIRE [113] prove the satisfiability/validity of many hard classical first-order formulas. The proof search, however, may not terminate in all cases. A number of first-order theories exists that are both decidable and expressive enough to formulate non-trivial system properties, examples of which include, among others, Presburger and bit vector arithmetic, or the theory of arrays. The satisfiability modulo theories, for example, decidable first-order theories is established by SMT solvers like Z3 [51], Yices [58], and CVC4 [13], to name but a few.

2.3 Summary

In this chapter we presented the background material required in subsequent chapters. We gave, however, only a brief and far from complete overview of many different topics, but refer the interested reader to explore the provided references to obtain a more thorough overview on selected topics. In later chapters, we will extend on the material of this chapter where necessary.

CHAPTER 3

Verification Approaches for Behavioral and Temporal Aspects in MDD

This chapter reviews the state-of-the-art in verification approaches for model-based software development. The focus lies on approaches that establish a *behavioral equivalence relation* which assert that either the modeled behavior of a system satisfies its specification, i.e., the system behaves as prescribed by, e.g., the functional requirements, or that two representations of a system describe the same behavior. We classify these approaches according to their distinguishing characteristics. These characteristics encompass (i) the verification scenarios which the verification approach is able to solve, (ii) the types of models the approach accepts as input, (iii) the internal, formal representation of these input models, (iv) the specification language usable with the approach, and (v) the underlying verification technique employed by the approach. The classification of the reviewed verification approaches is supported by a feature model that succinctly summarizes the characteristics of each approach.

The importance of verification approaches that assert the behavioral equivalence of modeling artifacts becomes apparent from the different steps involved in developing software following an MDSD approach which is exemplarily sketched in the following.

Example. The development team of an MDSD project typically starts with the definition of a set of platform independent models (PIM) that focus on architectural concepts as well as functional and non-functional requirements but exclude details on the specific implementation technologies [189]. The set of functional requirements gives rise to what we will refer to as the *specification* in the following.

In the initial stage of the project, a combination of different types of models are used to describe the behavior of a system. Each of these models focuses on a specific aspect of the system. Because these models overlap, the information presented in one of them will be used in another model with a different viewpoint onto the system. Thus, the developers need to ensure that the behavior presented in one of these models is consistent with or *equivalent to* all other models.

At some point in the development process, model transformations will be used to derive other models from a set of input models. For example, from a set of platform independent models and a configuration model, a platform specific model might be derived. In this case, the behavior described by the input models needs to be accurately reflected by the resulting output models.

The behavior of a system can also be expressed by means of model transformations. In this case, a set of model transformations describes an algorithm that operates on instances of a model that represents the system. Instead of changing values of, e.g., variables, the model transformations rewrite the structure of the instance. The sequence of instance models that result from continuously applying a set of model transformations to an instance model is an execution trace of the system. Dedicated verification techniques, for example, model checking [42,164], assert that the set of execution traces, which describe the behavior of the system, satisfy the specification. In other words, these verification techniques check whether the behavior of the system given by its execution traces is equivalent to the behavior described by the specification.

Related Work. Previous surveys focus solely on the verification of model transformations. To the best of our knowledge, Amrani *et al.* [4] were the first to propose a tri-dimensional categorization of verification approaches for model transformations. They categorize approaches according to (i) the type of the model transformations that can be verified, (ii) the properties of the transformations that can be analyzed including termination as well as syntactic and semantic relations, and (iii) the employed verification technique. Recently, they presented a generalization of their categorization and introduce a catalog of intents that allows them to classify model transformations according to their intended applications, which includes, but is not limited to, verification [3]. Calegari and Szasz [40] re-use Amrani *et al.*'s tri-dimensional categorization and suggest further subcategories for each dimension. Rahim and Whittle [1] classify formal and informal approaches according to the technique employed to assert the correctness of model transformations. In contrast, we consider model transformations as one of many possibilities to specify the behavior of a system and, more generally, concentrate on verification approaches that assert if a model behaves as prescribed by another model or its specification.

Bibliographic Note. The classification and the resulting feature model described in this chapter where first presented at the VOLT2013 workshop [72]. The original classification of five different verification approaches was later extended to more than 40 different approaches and published as a technical report [73].

3.1 Description of Verification Approaches

Theorem Proving

In the following, we review the diverse field of theorem proving-based approaches. It is characterized by the use of rich and highly expressive specification languages. Since all of the approaches propose either a manual or an interactive proving process, their main area of application is that of security critical systems, for which the significant increase in time, effort, and expertise required to perform the verification is justified.

Model Transformations from Proofs

Poernomo and Terrell [159] synthesize transformations from its specification and thus ensure the translation correctness of the transformations. The synthesis is performed in the interactive theorem prover Coq [46]. In essence, their approach derives a correct-by-construction transformation from a proof of the transformation's specification using the Curry-Howard isomorphism. They encode OCL-constrained, MOF-based (meta)models into co-inductive types in Coq, which allows them to model bi-directional associations. The specifications are formulated as OCL constraints and are encoded in Coq into $\forall\exists$ formulas, i.e., $\forall x \in A.\ \mathrm{Pre}(x) \rightarrow \exists y \in B.\ \mathrm{Post}(x, y)$. This specification schema demands that for all source models x, which conform to metamodel A and satisfy the pre-condition $\mathrm{Pre}(x)$, there exists a target model y conforming to metamodel B such that the postcondition $\mathrm{Post}(x, y)$ holds. According to the Curry-Howard isomorphism, a transformation can be extracted from a proof of this specification that converts a source model x satisfying $\mathrm{Pre}(x)$ into a target model y such that $\mathrm{Post}(x, y)$ holds. The Curry-Howard isomorphism establishes a mapping between logic and programming languages, where propositions correspond to types and their proofs correspond to programs. It essentially states that a function f can be extracted from a proof of a proposition $A \rightarrow B$ such that f applied to an element of type A returns an element of type B [188]. Then, the extracted function f corresponds to the transformation that satisfies the specification. Further, Poernomo and Terrell propose to partition the transformation specification into a series of sub-specifications, which allows the users to express more complex transformations and to reason modularly over the sub-specifications.

Correctness of Graph Programs

Poskitt and Plump [161, 162] present a Hoare calculus for graph transformations, which are specified with graph programs [127]. The calculus consists of a set of axioms and proof rules to assert the partial [161] and the total correctness [162] of graph programs. Graph programs operate on untyped, *labeled graphs*. Labels can be attached to nodes and edges, and may represent identifiers and attributes. Multiple attributes can be assigned to a node as an underscore-separated list of values. For example, the string "TheSimpsons_MattGroening" identifies the node of a movie database that represents Matt Groening's sitcom "The Simpsons."

A *graph program* consists of a set of conditional rule schemata and a sequence of commands that controls the execution order of the rule schemata. A conditional rule schemata, in the following just *rule*, is a parameterized function, whose instruction block consists of a labeled left-hand and a labeled right-hand side graph. A label is an integer or a string expressions over the function's parameters and can be attached to a node or an edge. An instruction block can contain an optional *where*-clause that restricts the applicability of the rule. The rewriting is performed according to the double pushout approach with relabeling [83]. The sequence of commands that controls the execution of a graph program is a semicolon-separated list of rules that are either executed once or as long as applicable.

Poskitt and Plump represent (software) systems by labeled graphs and transformations with graph programs. They can verify both the translation correctness and the behavioral correctness. In the latter case, the graph program describes the behavior of the system; in the former, it describes the transformation that needs to be verified. The specification of a graph program is formulated as a Hoare-triple $\{c\}\,P\,\{d\}$ that consists of a precondition c, a postcondition d, and the graph program P. Pre- and postconditions are defined by so-called *E-conditions*, which are either *true* or have the form $e = \exists(G \,|\, \gamma, e')$. An E-condition consists of a premise $G \,|\, \gamma$, where G is a graph and γ is an *assignment constraint* that restricts the values assignable to labels in G, and a conclusion e', which is again a (nested) E-condition. Intuitively, a graph H satisfies an E-condition $e = \exists(G \,|\, \gamma, e')$ if G, whose variables are assigned to values that satisfy the assignment constraint γ, is a subgraph of H and the nested E-condition e' holds.

A graph program P is partially correct if postcondition d holds in all graphs H that result from a terminating run of P on any source graph G that satisfies precondition c. Similarly, total correctness is achieved if P terminates on every graph G that satisfies precondition c and postcondition d holds in all resulting graphs H. The actual verification process is performed manually and results in a proof tree, which derives, i.e., proves, the specification $\{c\}\,P\,\{d\}$ (cf. Hoare logic [91]).

Verifying QVT Transformations

Stenzel *et al.* [191] verify properties of operational QVT (QVTO) transformations with the interactive theorem prover KIV [107]. They implement a sequent calculus based on dynamic logic [84] in KIV. A dynamic logic extends a base logic, for example, propositional or first-order logic, with a modality $\langle . \rangle$, called the *diamond* operator. A dynamic logic formula $\langle p \rangle\, \varphi$ is satisfied if φ holds in all successor states of the current state after the execution of program p, which is required to terminate. Note that φ is either again a dynamic logic formula or a formula of the base logic. In case of Stenzel *et al.*'s approach, programs p are of the form $(\varepsilon)\alpha$, where α is a QVTO expression and $\varepsilon = (in, out, trace)$ is the environment, which consists of an input model *in*, an output model *out*, and a trace model *trace*. Their calculus defines proof rules for a subset of the commands offered by QVTO. The proof rules are of the form $\Gamma \vdash \Delta$ and consist of a set Γ of premises and a set Δ of conclusions. Premises and conclusions are dynamic logic formulas of the form $\langle(\varepsilon)\alpha\rangle\, \varphi$. The specification can now be expressed, analogous to a Hoare-triple

$\{\varphi\}\,\alpha\,\{\psi\}$, with a sequent $\varphi \vdash \langle\alpha\rangle\psi$, where α is the QVTO expression that triggers the execution of the transformation provided that φ is satisfied. For example, we can express that a transformation `CDtoER`, which converts a UML class diagram (CD) into an entity relation (ER) diagram, produces for every class a table carrying the name of the corresponding class with the dynamic logic formula

$$\begin{aligned}&\text{conformsTo}(in, CD) \vdash \langle(in, out, trace)\ \texttt{CDtoER} :: \texttt{main}()\rangle\\ &(\forall c \in in.\ \exists t \in out.\ \text{isClass}(c) \wedge \text{isTable}(t) \rightarrow [\![c.name]\!]_{CD} = [\![t.name]\!]_{ER}),\end{aligned}$$

where conformsTo(m, M) tests conformance of model m to metamodel M and $[\![expr]\!]_{\{M\}}$ evaluates expression *expr* according to the semantics associated with M.

The authors use their calculus in a framework to prove semantic properties of a code generator that produces an intermediate model, called the Java Abstract Syntax Tree (JAST) model, from a set of Ecore models. The JAST model is mapped to a formal Java semantics defined in KIV. The JAST model acts as the source model for the model-to-text transformation that generates the actual Java code. They set up a transformation chain that translates several Ecore models into a JAST model and the JAST model to Java code. The authors verify the type correctness of the Ecore-to-JAST transformation and check that the transformation satisfies a set of user-defined, semantic properties.

Behavior Preserving Transformations

Hülsbusch *et al.* [93] present two strategies to manually prove that a model transformation between a source and a target model preserves the behavior. One strategy is based on triple graph grammars (TGG) [177] and the other on in-situ graph transformations and borrowed contexts [66]. The source and the target models of the transformation are represented as graphs, which may be typed over different type graphs, and the operational semantics of the source and the target graphs are defined with graph transformations. They declare a model transformation, either a TGG transformation or an in-situ graph transformation, *behavior preserving* if there exists a weak bisimulation[1] between the source and the resulting target graph with respect to their operational behavior. In case of TGG, the bisimulation can be derived from the correspondence graph that relates the source and the target graph and vice versa. The second proof technique uses in-situ transformations that perform the rewriting directly in the source model (*in-place*) thereby mixing source and target model elements. The bisimulation relation is established via *borrowed contexts* [88, 94]. A third technique to assert that a model transformation preserves the behavior is presented by Ehrig and Ermel [64]. Similar to Hülsbusch *et al.* they define the operational behavior with graph transformations, called the *simulation rules*, and another set of graph transformations that convert the source to the target model. They then apply the latter to the simulation rules, that is, they rewrite the simulation rules, and check if the transformed simulation rules of the source model match the simulation rules of the target model.

[1]Weak bisimulation allows *internal* steps for which no corresponding step in the opposite system may exist.

Giese and Lambers [77] sketch a technique to prove automatically that a TGG-based model transformation is behavior preserving. They show that the problem of asserting bisimilarity between the graph transition systems for the source and target model can be reduced to checking if a constraint over the graph transition systems, the bisimilarity constraint, is inductive.[2]

Verification of OCL Specifications

Starting with version 2.3 of the OCL standard Brucker and Wolff formalize a core of OCL in Isabelle/HOL, called Featherweight OCL [35, 36]; most notably they implement OCL's now four-valued logic consistently, which already led to corrections that were incorporated into version 2.4 of the OCL standard. They suggest to replace the current "Annex A" of the OCL standard [148] by an automatically derived formal proof document generated directly from their formalization with the document generation facilities provided by Isabelle/HOL. Although Featherweight OCL provides a consistent formalization of OCL, its primary is to act as a reference implementation for OCL tool developers. In this line of work, they propose an Isabelle/HOL-based compiler that generates for a given UML/OCL model, i.e., a UML class model with OCL constraints, a corresponding object-oriented data type theory in Isabelle/HOL [124]. This generated data type theory includes proofs for lemmas of type casts, type tests, constructors, and attribute accessors, which provide the basis for more involved proofs on, e.g., the behavior of the system represented by the UML/OCL model.

Kyas *et al.* [116] present a prototype that verifies OCL invariants over simplified UML class diagrams, whose behavior is described by state machines. They assert the behavioral correctness of a system and translate its class diagrams, state machines, and OCL specifications into the input format of the interactive theorem prover PVS [156]. Similar to other theorem proving-based approaches, they are able to prove OCL properties of infinite state systems; for example, they demonstrate how to verify a system that grows indefinitely, i.e., has an unbounded number of objects.

Verification with Isabelle/HOL

Strecker [192] formalizes the theory of graph transformations in higher-order logic with the intention to prove behavioral properties of systems interactively with the Isabelle/HOL theorem prover [145]. With this formalization it is possible to reason about the effect of a transformation and to derive assertions on the shape of the graph that results from the application of a transformation. Thus, the reasoning is not limited to the behavioral correctness properties, but also admits the verification of translation correctness. Software models are encoded into untyped or typed graphs, where nodes are indexed by and mapped to natural numbers and edges are represented as a binary relation over

[2]In general, a constraint or assertion c over a transition system is said to be *inductive* if $G_0 \Rightarrow c$ (base case) and $c \wedge T \Rightarrow c'$ (induction step) holds where G_0 is the initial state, T is the transition relation, e.g., a graph rewriting rule $r : G \rightarrow G'$ transforming a graph G into G', and c' denotes the constraint in the next state.

natural numbers. A typing function assigns types to nodes and the type correctness of a graph is enforced by a well-formedness constraint. Note that attributes and inheritance hierarchies are not supported natively. Hence, a graph consists of a set of natural numbers to represent the graph's nodes, a binary relation over the natural numbers to represent edges, and a typing function to assign types to nodes. The LHS and the RHS of a transformation are encoded separately into an *application condition* and an *action*, respectively. An application condition is specified as a *path formula* that describes the structure of the graph required to apply the transformation. The action then describes the effects of the transformation by adding or removing indices to or from the set of nodes and updating the edge relation accordingly.

The formalization provides a Hoare-style calculus that verifies the partial correctness of a higher-order logic specification. A specification $w \vdash \{c\}\, P\, \{d\}$, intuitively, demands that, given some graph G, which satisfies the well-formedness constraint w and the precondition c, the postcondition d holds in graph H that results from the application of the transformation P to graph G.

Strecker [193] proposes two reduction techniques[3] to simplify the interactive proving procedure for reachability properties. The first decomposes a graph into smaller subgraphs such that properties proven for a subgraph hold in the original graph. The second technique aims to restrict the reasoning to the shape of the graph transformation itself and is applicable only if the matching morphism is assumed to be injective and the application condition is a conjunction of relations over edges.

Model Checking of Rewriting-Based Systems

When software systems are modeled with graphs and their behavior is described by graph transformations, temporal properties can be verified with model checking-based techniques. Here, states are represented by graphs and state transitions correspond to the application of a graph transformations to a source state, which results in (or leads to) a target state [87]. More formally, given a graph grammar $\mathcal{G} = (\mathcal{R}, \iota)$ with a set of graph transformations $\mathcal{R}$ and an initial graph ι, a *graph transition system* (GTS) is constructed by recursively applying the graph transformations to the initial graph and all resulting graphs. The graphs generated by the graph grammar constitute the states of the GTS and the transitions between two states G and G' correspond to the application of a graph transformation $p : G \to G'$.

The same technique is also employed by term rewriting-based approaches and tools, e.g., Moment2 [30], where states are represented by terms and transitions correspond to (term) rewrite rules that are applicable to these terms.

Model Checking of Graph Transition Systems

One of the first model checkers for graph transition systems was CheckVML [176, 200]. It targets the behavioral verification of systems defined by

[3]The source files for Isabelle/HOL are available from `http://www.irit.fr/~Martin.Strecker/Publications/proofs_graph_transformations.tgz`

UML-like class diagrams. CheckVML receives a metamodel that describes the structure of the system, a set of graph transformations that define the system's behavior, and a model instance that describes the system's initial state to produce a graph transition system. Internally, the metamodel is represented as an attributed type graph with inheritance relations and the initial model is an instance graph conforming to the type graph derived from the metamodel. CheckVML uses the model checker spin as its verification back-end. It thus encodes the GTS into Promela code, the input language of spin. For each class the encoding uses a one-dimensional Boolean array, whose index corresponds to the objects' IDs, and the value stored for each object indicates whether the object is active or not. Since arrays are of fixed size CheckVML requires from the user an *a priori* upper bound on the number of objects for each class. Further, for each association CheckVML allocates a two-dimensional Boolean array that stores whether there exists an association between two objects. To construct a finite encoding of the system the domain of each attribute is required to be finite such that it can be represented by an enumeration of possible values in Promela. Further, since spin has no knowledge of graph transformations all possible applications for each transformation are pre-computed and transitions are added to the Promela model accordingly. To reduce the size of the state space CheckVML tries to identify static model elements that are not changed by any transformation and omits them from the encoding. The state space, however, still grows fast as symmetry reductions for the encoding are possible only to a very limited extent in spin. For example, a direct comparison [167] with Groove [165] (see below) showed that the encoding of the dining philosophers problem with ten philosophers produces 328 503 states but only 32 903 are actually necessary. Interestingly, even though the state space is an order of magnitude larger, the performance of the verification does not degrade as anticipated. CheckVML with its spin back-end verifies the dining philosophers instance 12x faster (16.6 seconds including pre-processing) than Groove (199.5 seconds) [167].[4] CheckVML supports the specifications of safety and reachability properties by means of *property graphs* that are automatically translated into LTL formulas for spin. Unfortunately, counter-example traces from spin are not translated back automatically.

A similar approach is proposed by Baresi *et al.* [11], whose encoding produces BIR (Bandera Intermediate Representation) code for the model checker Bogor [171]. They translate typed, attributed graphs into sets of records. They, too, bound the number of admissible objects per class. Again, associations are encoded into arrays of predefined, fixed size. This approach supports class inheritance, i.e., in a preprocessing step all inheritance hierarchies are flattened such that attributes of the supertypes are propagated to the most concrete type. Like CheckVML, containment relations are not supported natively. Further, they distinguish between static and dynamic references and keep track of

[4]With version 4.5.2 of Groove (build: 20120606174037) the verification requires 13413.8ms on an Intel Core i5 2.67Ghz with 8GB of RAM running Gentoo Linux with OpenJDK 1.6. Taking into consideration that Groove was in its infancy when the comparison was performed in 2004, this improved result reflects the development efforts of past years. In contrast, spin, the verification back-end of CheckVML, has been under active development since the 1980s [14]. However, we cannot provide up-to-date runtimes for CheckVML as it is currently not available to the public.

the set of currently active objects. For each transformation two distinct BIR fragments are generated. The transformation's LHS is encoded into a matching fragment, while the RHS is encoded into a thread that executes the effects of the transformation once a match has been detected. Since BOGOR is not aware of graph transformation theory either and does not provide constructs to match graph structures, the matching fragment queries attribute values and existence of associations from every possible combination of active objects that could possibly be matched. The user can specify safety properties that should hold in the system, just like in CHECKVML, with property graphs. These are converted into LTL formulas and encoded into BIR property functions.

Previously to the above described approach, Baresi and Spoletini [12] presented an encoding that allows to analyze graph transformations specified in AGG [174] with the ALLOY analyzer [96]. The authors model a (software) system by means of a type graph, which captures the static components of the system, and a set of graph transformations that specify the system's behavior. Instance graphs that conform to the type graph represent the possible states of the system. The execution of a system is modeled with finite paths, which are sequences of instance graphs. The encoding creates ALLOY signatures for the type graph and predicates for the graph transformations. Each predicate represents the effect of a transformation, i.e., the addition, removal, and preservation of vertices and edges, with relational logic formulas. The resulting transformation predicates are used to define the possible state transitions of the system and restrict the execution paths to valid system behaviors, i.e., a transitions between two instance graphs is only possible if the effect of the transition satisfies a transformation predicate. ALLOY supports reachability and safety analysis of system properties, which are specified as first-order formulas. The authors use this feature to show that either (*a*) a certain state is reachable or (*b*) a counter-example exists that violates a safety property.

Kastenberg and Rensink [106] propose GROOVE,[5] an enumerative state model checking approach to verify the behavioral correctness of object-oriented (OO) systems. The static structure of an OO-system is described by an attributed type graph with inheritance relations, while the system's behavior is, again, defined through graph transformations. Hence, states are represented by (instance) graphs conforming to the type graph. The GROOVE Simulator [165] generates the state space on the basis of a *graph grammar* $\mathcal{G} = (\mathcal{R}, \iota)$, which consists of an initial graph ι and a set $\mathcal{R}$ of graph productions. Between two states s and s' there exists a transition if a graph transformation can be applied to the graph of s such that the result is isomorphic to the graph of s'. The resulting state-transition structure is a graph transition system (GTS) and converted into a Kripke structure, where states and transitions of the GTS correspond directly to those of the Kripke structure. The Kripke structure's labeling function assigns to each state the names of the applicable graph productions. In GROOVE, similar to CHECKVML's property graphs, system properties are defined with graph constraints, i.e., named graph productions, whose LHS and RHS are equivalent. The names of these graph constraints define the alphabet of the propositions that can be used in the specification of the system. GROOVE is able to verify CTL and LTL formulas that express either reachability,

[5]Available from `http://groove.sourceforge.com`

safety, or liveness properties. To reduce the size of the state space GROOVE checks if a graph is isomorphic to any existing graph before adding it to the state space. The isomorphism check is, however, computationally costly and, thus, Rensink and Zambon investigated alternative state space reduction methods that use neighborhood [168] and pattern-based abstraction techniques [169], of which the former has been implemented in GROOVE. Neighborhood abstraction partitions a graph into equivalence classes and folds two nodes into the same equivalence class if (*a*) they have equivalent incoming and outgoing edges, and (*b*) the target nodes of these edges are comparable [168]. For each equivalence class, neighborhood abstraction records the precise number of folded nodes up to some bound k and beyond that simply ω for *many*. Pattern-based abstractions capture the properties of interest in layered *pattern graphs*, which are, similar to neighborhood abstraction, folded into *pattern shapes*. The abstraction of the system's transformations is then directed by these pattern shapes. The resulting *pattern shape transition system* (PSTS) is an over-approximation of the original GTS and the authors show that properties that hold in the PSTS also hold in the GTS. However, an implementation cannot be derived straightforwardly and its efficiency, compared to neighborhood abstraction, is subject of future evaluations.

HENSHIN[6] [6] is a model transformation tool for Ecore models. Internally, it represents Ecore models as typed, attributed graphs with inheritance and containment relations [24]. HENSHIN is thus the only graph-based tool that natively supports containment relations. Similar to GROOVE [165], HENSHIN provides an enumerative state space explorer for graph transition systems and provides an interface to communicate with external model checkers. Currently, it supports the model checker CADP [75] out-of-the-box, which is able to verify μ-calculus [114] specifications. Moreover, invariant properties specified with OCL constraints can be checked over the entire state space.

In contrast to the above described approaches that target the verification of a system's behavioral correctness, Narayanan and Karsai [140] use a bisimulation-based approach to assert the translation correctness of a set of exogenous graph transformations that transform a source model conforming to type graph A into a target model conforming to type graph B. More specifically, they check if the graph transformations preserve certain, user-imposed reachability properties of the source model. The approach does not require that the actual behavior of the source and target model be defined explicitly. Instead it verifies the transformations by establishing a *structural correspondence* between source and target type graph, which consists of a set of *cross-links* that trace source model elements to target model elements and a set of *correspondence rules* that define conditions on the target model to enforce the reachability properties across the transformation. Then, the structural correspondence defines the bisimilarity relation between the source and the target model. The approach assumes that the correspondence rules are developed (*a*) independently from the transformation and (*b*) with fewer or zero errors because they are less complex as compared to the actual graph productions. Further, the cross-links need to be established whenever a graph production generates traceable target model elements. The verification of the reachability properties is performed for

[6]Available from `https://www.eclipse.org/henshin/downloads.php`

each source instance that is translated into a target instance. The verification engine uses the source instance, the cross-links, and target instance and checks if the correspondence rules are satisfied. If the verification succeeds the target instance is *certified correct*.

Verification of Infinite State Graph Grammars

Besides the recent abstraction mechanisms introduced into Groove, the approach by Baldan *et al.* [10] and by König and Kozioura [110], who extend the former, are the only model checking approaches that use abstraction techniques to verify infinite state spaces. Given a graph grammar $\mathcal{G} = (\mathcal{R}, \iota)$ they construct a *Petri graph.* A Petri graph is a finite, over-approximate unfolding of $\mathcal{G}$ that overlays a hypergraph with an $\mathcal{R}$-labeled Petri net such that the places of the Petri net are the edges of the hypergraph. Each transition of the Petri net is labeled with a rule $r \in \mathcal{R}$, and the in- and out-places of a transition are the hypergraph's edges matched by the LHS and the RHS of rule r. From a $\mathcal{G} = (\mathcal{R}, \iota)$ a pair (P, m_0) is constructed that consists of the Petri graph P and an initial marking m_0 that assigns a token to every place with a corresponding edge in ι. That is, a marking of the Petri net assigns tokens to the edges of the hypergraph. Each marking defines, in this manner, a distinct state of the system, which is obtained by instantiating an edge for each token it contains and gluing together the edges' common nodes to build the resulting hypergraph. The firing of a transition then corresponds to the application of the rule r that labels the transition and triggers a state change, i.e., the marking resulting from the firing defines the next system state.

The approximated unfolding constructs a Petri graph that, in the beginning, consists of the initial hypergraph ι and a Petri net without transitions where the places are the edges of ι. An *unfolding* step selects an applicable rule r from $\mathcal{R}$, extends the current graph by the rule's RHS, creates a Petri net transition labeled with r, whose in- and out-places are the edges matched by the rule's LHS and RHS, respectively. A *folding* step is applied if, for a given rule, two matches in the hypergraph exist such that their edges (i.e., places) are coverable in the Petri net and if the unfolding of the sub-hypergraph identified by one of the matches depends on the existence of the sub-hypergraph identified by the second match. The folding step then merges the two matches. The procedure stops if neither folding nor unfolding steps can be applied. Baldan *et al.* [10] show that the unfolding and folding steps are confluent and are guaranteed to terminate returning a unique Petri graph for each graph grammar $\mathcal{G}$. Moreover, the Petri graph overapproximates the underlying graph grammar conservatively, that is, every hypergraph reachable from ι through applications of $\mathcal{R}$ is also reachable in the resulting Petri graph.

Since the Petri graph over-approximates the unfolding of $\mathcal{G}$, there exist, however, runs that reach a hypergraph unreachable in $\mathcal{G}$. Such a run is classified as *spurious.* If such a spurious run violates the specification, that is, there exists a *spurious counterexample* trace to an error that is due to the over-approximation and not realizable in the original system. Inspired by the work on counterexample-guided abstraction refinement (CEGAR) [43], König and Kozioura [110] present an abstraction refinement technique for Petri graphs. They show that spurious counterexamples result from the folding operation that merges nodes. Thus, their technique identifies nodes that must not be merged in order to prevent a spurious counterexample. Their CEGAR techniques for hypergraphs is implemented in AUGUR 2.[7]

Recently, König and Kozioura extended their CEGAR-based verification approach to attributed graph grammars [112]. The Petri graph then consists of an *attributed* or colored Petri net and an overlaid, *non-attributed* hypergraph structure. The over-approximated unfolding proceeds as above, but without taking the attributes into account, which, intuitively, leads to the coarsest possible abstraction. Only when the over-approximation has been constructed are the attribute values of the initial graph ι assigned to the corresponding places of the Petri graph. As the domains of the attribute values are usually infinite, abstract attribute values are computed [48]. A spurious counterexample may now be either due to the structural over-approximation of the hypergraph or due to the attribute abstraction. In the first case, the abstraction is refined as described above; in the second case, the abstract domain is refined semi-automatically with the help of the user or according to a predefined scheme that runs a certain number of iterations and aborts if the spurious counterexample is not eliminated.

The technique admits the verification of specifications formulated either over the (attributed) Petri net or the hypergraph structure of the over-approximated Petri graph. That is, the user needs to decide whether the specification is expressed over the marking of the (attributed) Petri net or if it is best captured by a graph morphism over the hypergraph [10]. In both cases, the specification is described with graphs, either by means of a discrete graph that represents a marking of the Petri net, or by means of a graph morphism with equivalent LHS and RHS graphs (cf. with property graphs in CHECKVML and GROOVE). If the specification can be verified over the Petri net, it is possible to verify reachability, boundedness, and liveness properties, while graph morphisms can express reachability properties.

Verification of QVT and ATL Transformations

Boronat *et al.* describe systems with OCL-constrained, MOF-based metamodels and their behavior with QVT-like model transformations. They present algebraic semantics for (*a*) MOF [32]; (*b*) model conformance with respect to OCL-constraints [31]; and (*c*) QVT-like model transformations [30] based on membership equational logic (MEL) [132] and rewriting logic (RWL) [131]. This formalization allows them to express OCL-constrained Ecore models and QVT-like model transformations as theories in MEL and RWL, respectively. A MEL theory (Σ, E) consists of a signature Σ and a set E of Σ-sentences. The signature defines a set of *function symbols* and a set of *kinds*, where each kind is associated with a set of (ordered) sorts. Given a set X of variables, every variable in X and every function symbol applied to a variable or another function symbol defines a Σ-*term*. If a term t is a member of just a kind but not of a sort it represents an undefined or an error value. For example, the constant term `NaN` (*Not a Number*) is member of kind `Number` but neither member of sort `Real` nor `Integer`. A

[7]Available from `http://www.ti.inf.uni-due.de/research/tools/augur2/`

division-by-zero error can thus be expressed, for example, by returning the term `NaN`. Sentences in E are conditional equations of the form $t = t'$ *if* $\bigwedge_{i \in I} v_i = w_i \wedge \bigwedge_{i \in I} t_i : s_i$, which consist of an atomic equation $t = t'$ and a condition, i.e., a conjunction of atomic equations $v_i = w_i$ and membership assertions $t_i : s_i$ that assign a term t_i to some sort s_i. A rewriting logic theory (Σ, E, R) consists of a MEL theory and a set R of rewrite rules that are of the form $t \rightarrow t'$ *if* C where condition C is a conjunction of atomic rewrite rules, atomic equations, and membership assertions. A RWL theory can be used to represent a concurrent system, where the system's states and transitions are defined by a deterministic MEL theory[8] and a set of rewrite rules, respectively. Each term, rewritten to its unique normal form[9] by the MEL's equations (interpreted from left to right), defines a state of the concurrent system. A rewrite rule in R applied to a term defines a transition in the concurrent system. An RWL theory can be executed as a *system module* in MAUDE [45].[10] The MOMENT2 tool[11] automatizes the process of translating Ecore models and corresponding model transformations into system modules such that MAUDE's reachability analysis and LTL model checker can be used to verify the system's specification [30]. MAUDE builds the state space as a *derivation tree* for both analyses and proceeds as follows. Given an initial term that represents the system's initial state, MAUDE applies all rewriting rules in R recursively to each resulting term, thus, building a derivation tree rooted in the initial term. In both cases, MAUDE explores the state space of the system enumeratively. For a reachability property, which is specified by a term that should be shown reachable in the derivation tree, MAUDE searches breadth-first trough the derivation tree starting from the given initial term. The search stops if (*a*) the term is found; (*b*) the entire state space has been explored and the term is not found; (*c*) the user-provided search-depth is reached without encountering the term, or (*d*) MAUDE runs out of memory while searching for the term. If the term that MAUDE searches for in the derivation tree expresses an error state, safety properties can be verified by asserting that such a term is not reachable. LTL specifications are formulated over a set of propositions that are defined as equations where the right-hand side defines the name of the proposition and the left-hand side defines the pattern or conditions required for the proposition to hold. Then, if the proposition-defining equation is interpreted as a rewrite rule and a state can be rewritten in this manner, i.e., a state's sub-term matches the left-hand side of the proposition-defining equation and is thus labeled with the name of the proposition, then the state is said to satisfy the proposition. MOMENT2 does not support the specification of LTL formulas; they have to be written and executed directly in MAUDE.

Gagnon *et al.* [74] have proposed a similar model checking based approach based on MAUDE. They, too, target the behavioral verification of systems but represent these systems and their behavior by means of UML class diagrams as well as state and communication diagrams, respectively. They describe how simplified class, state, and com-

[8]A MEL theory is *deterministic* if its equations, interpreted from left to right, are confluent and terminating such that every term can be rewritten into a unique normal form.

[9]For an introduction to term rewriting refer to [7] and [17].

[10]For an RWL theory to be executable as a system module has to be *coherent* [45].

[11]Available from `ftp://moment.dsic.upv.es/releases/20070727/`

munication diagrams can be (manually) encoded into RWL theories and show how LTL specifications can be verified within MAUDE.

Troya and Vallecillo [199] present for ATL transformations a formal semantics based on rewriting logic. They formalize ATL's default and refining execution mode such that both translation and behavioral correctness can be asserted. Further, their formalization makes it possible to automatically translate ATL into MAUDE system modules. In particular, they translate matched rules, (unique) lazy rules, called rules, helper functions, and imperative rule blocks into RWL theories. They, too, propose to use MAUDE's reachability analysis to verify safety properties of systems that are described by Ecore models and whose behavior is specified by ATL transformations. However, they do not integrate the verification into their ATL-to-MAUDE translation and only sketch the possibility that their approach admits the verification of behavioral properties and do not consider the possibility to assert the translation correctness of the ATL transformation at all.

Büttner *et al.* [38] verify with ALLOY [96] if an exogenous ATL transformation that is defined for an OCL constrained Ecore model preserves the target model's invariants. From metamodels MI, MO, where $MI \nparallel MO$, and an ATL transformation $t : MI \rightarrow MO$ that transforms a source model conforming to MI into a target model conforming to MO Büettner *et al.* build a *transformation model* that captures MI, MO, and the ATL transformation t in a single model. Basically, a transformation model traces which elements of the source model are translated to what elements of the target model. Further, they define a conversion from transformation models to ALLOY using the UML2ALLOY tool [5]. The verification is performed in three steps. First, an OCL constraint defined for the target model is selected and negated, while all other constraints are disabled. Observe that all instance models of the target metamodel that satisfy the negated constraint are invalid. In the second step, the transformation model is constructed from the source metamodel, the ATL transformation, and the target metamodel, where the selected OCL constraint has been negated. Finally, ALLOY is used to check if there exists a model that satisfies the modified, but invalid transformation model. Otherwise, if it finds no counterexample that satisfies the negated constraint of the target model, a counterexample is found to the validity of the original transformation model and we conclude that the ATL transformation does not preserve the invariants of the target model. If it finds no such model, we can, however, only conclude that the ATL transformation is correct up to a certain number of instance objects in the source model. This restriction is due to ALLOY that demands a bound on the number of investigated objects such that its search space of possible logical models remains finite. In contrast to the approach presented by Troya and Vallecillo [199], this approach can only handle ATL's *matched rules* and no recursive helper functions are allowed. The strength of the approach, however, lies in its lightweight methodology that builds on the *small scope hypothesis* [97] and its ability to translate counterexamples from ALLOY back into Ecore.

Model Checking of OCL Specifications

When MOF or UML models are used to describe systems, OCL is often the language of choice to phrase the specification of the system. However, the language is limited to express properties over a single snapshot of the system and cannot reason over sequences of system snapshots, i.e., execution traces. Thus, numerous temporal extensions to OCL have been proposed to overcome this limitation. In this section, we review model checking-based verification approaches that use either existing or custom-built temporal OCL extensions to formulate a system's specification.

Language Extensions & Mappings

Distefano *et al.* [55] propose a CTL-based logic, called BOTL, to specify static and temporal properties of object-oriented systems, but they do not support inheritance nor subtyping. Instead of extending OCL, they map OCL onto BOTL, thus providing a formal semantics for a large part of OCL. With BOTL they are able to express OCL invariants and pre- as well as postconditions. BOTL does not assume that methods are executed atomically and allows intermediate states during the execution of a method. Consequently, invariance properties specified with BOTL need to ensure explicitly that no method is executed that alters a value associated with an invariant while the invariant is being checked.

Ziemann and Gogolla [203] propose TOCL (Temporal OCL) that extends syntax and semantics of OCL with past and future temporal operators like next, until, and before. These operators are evaluated over linear, infinite traces of the system. Moreover, they introduce constructs that allow them to express that a function is executed in the next or the previous state. TOCL was the first temporal extension, whose semantics and evaluation were aligned with the formal semantic of OCL [170]. For this purpose, they introduce an additional index into the evaluation environment of an OCL expression that points to the *current state*.

Soden and Eichler [185] present an LTL-based extension for OCL and suggest four additional keywords, next, always, eventually, and until. They, too, align the semantics of their extension with those of OCL and, like Ziemann *et al.*, they introduce an additional index into the evaluation environment that captures the current time instant. They suggest to define the operational semantics of MOF-conforming models with the Model Execution Framework for Eclipse (MXF) (formerly called *M3Actions*) [186]. This allows them to define a finite execution trace by a sequence of changes, from which the actual states are derived by applying the changes in succession to the initial model up to the current state.

Flake and Müller [70] aim at a tight integration of UML class diagrams, state machines, and OCL, where the state machines describe the behavior of the class diagrams. They formalize a subset of the UML 2.0 standard covering attributed classes with inheritance, associations, methods, signals, events, sequential and orthogonal composite states, guarded transitions, and pseudo states. The system state is captured by the set of currently active objects, their attribute values, connections among them, the current state machine configuration that contains the tree of active states, and the set of active operations and their parameter values. This definition allows them to specify the formal semantics of the oclInState(s : *OclState*) function. They use time-annotated traces to capture the evolution of the system and propose a UML profile to specify state-oriented, real-time invariants, whose semantics are defined by a mapping to clocked CTL formulas.

In regard to expressiveness, Bradfield *et al.* [33] propose the richest extension by embedding OCL into the observational μ-calculus [34]. As noted by the authors this expressiveness comes at the price of complexity that is inherent to specifications using the μ-calculus. They thus suggest the use of predefined templates, from which the actual μ-calculus are automatically synthesized. The templates are designed with the purpose of conveying their semantics in an intuitive manner. As an initial example, they introduce the *after/eventually* template that is used to assert that, *after* an event has occurred, an action is *eventually* executed. They also propose *after/immediately* and *provided/infinitely often* templates with their obvious interpretation.

Similar to Bradfield *et al.*'s proposed templates, Kanso and Taha [105] introduce a temporal extension based on Dwyer *et al.*'s patterns [59] for the specification of properties for finite state systems. Although these patterns are not as expressive as their CTL, LTL, or μ-calculus counterparts, they greatly simplify the property specification process. Kanso and Taha define a scenario-based semantics for their extension, where each scenario is a finite sequence of events. They distinguish between operation call, start, and end events as well as state change events that are triggered upon a change of an attribute value. The occurrence of an event is queried (a) by the isCalled(op, pre, $post$) function, which test if an operation op is called in a state satisfying pre with the effect that $post$ holds immediately after the atomic execution of op; and (b) by the becomesTrue(P) function, which test if a predicate P that was *false* in the immediately preceding state turned *true* in the current state. For example, the LTL formula $\mathsf{G}(req \Rightarrow \mathsf{F}\, ack)$ ("every request is eventually acknowledged") is expressed as globally becomesTrue(ack) responds to isCalled(req, $true$, $true$). Kanso and Taha implement a test case generator that uses the temporal OCL specification to derive test cases that are useful to examine the correctness of the implementation w.r.t. to the specification. They provide an implementation on their website.[12]

Language Extensions & Implementations

Mullins and Oarga [138] present an extension to OCL, called EOCL, that augments OCL with CTL operators. It is strongly influenced by BOTL but, in addition, supports inheritance. They define EOCL's operational semantics over object-oriented transition systems that in each state keep track of the active objects, active methods, and the active objects' attribute valuations. Their SOCLe tool[13] is able to assert the behavioral correctness of a system that is defined by a class diagram, a set of state machines for each class in the class diagram, and an object diagram that defines the initial state. For

[12] `http://wwwdi.supelec.fr/~taha_saf/temporalocl/`

[13] Unfortunately, SOCLe does not seem to be available to the public anymore.

the verification, SOCLe translates the class, state machine, and object diagrams into an abstract state machine. Then, it checks enumeratively and on-the-fly if the system satisfies its EOCL specification, which expresses either reachability, safety, or liveness properties.

Gogolla *et al.* present USE, the UML-based Specification Environment, that is aims to assist developers in validating their model-driven software artifacts. In particular, the USE tool allows to execute UML models and analyze OCL invariants and pre- and postconditions; hence, USE admits checking of both structural and behavioral properties. Although originally not intended as a verification environment recent extensions enabled work on at least two verification approaches, namely the *filmstripping* based and the *snapshot* based verification approaches, both of which are discussed below.

Gogolla *et al.* [81,89] verify the behavioral correctness of a system whose description consists of a static structure, defined by a UML class diagram, its behavior, provided by a set of OCL operation contracts, and a set of invariants that the system must satisfy. Their approach transforms the system's description into a so-called *filmstrip model* that explicitly represents sequences of system states and operation calls. The filmstrip model consists of a set of system states, referred to as *snapshots*, connected by operation calls. A snapshot comprises the active objects, the links between these objects, and the attribute values of these objects. An operation call represents an operation of system, whose functionality is described by an operation contract. Two snapshots are then associated by an operation call if the pre- and postconditions of the operation contract are satisfied by the first and the second snapshot, respectively. Since the sequence of states are thus made explicit in the filmstrip model, OCL pre- and postconditions of an operation contract for the original UML class diagram may now be translated into an OCL invariant for the filmstrip model. This is achieved by replacing the reference to a previous system state (`@pre`) by backward navigating the association between two snapshots established by an operation call. Thus, the problem of checking pre- and postconditions is reduced to checking an invariant over two snapshots. The verification procedure [90] is implemented in the USE (see above) and proceeds as follows. After the filmstrip model is generated, additional *frame conditions* [129,137] are added manually. Next, the verification condition and the conditions characterizing the initial states are formulated on top of the filmstrip model. Finally, global bounds on the number of objects and associations in the snapshots as well as the number of operation calls need to be defined. Subsequently, the verification problem defined in the previous steps is translated into bounded, relational, first-order logic and solved by the model finder KODKOD [198]. If KODKOD finds a model for the verification problem, an instance of the filmstrip model that violates the verification condition has been found. This counterexample is visualized as a sequence diagram. Recently, the authors have applied their approach to the verification of transformation models, that describe mappings between a source and target model [80]. To show that a transformation preserves a property of the source model in the target model, an invariant for the transformation model is defined and verified.

Similarly, Al-Lail *et al.* [117] verify the behavioral correctness of systems, too. They describe systems with class diagrams, where the operations' contracts, which are specified by OCL pre- and postconditions, capture the behavior of the system. They use TOCL [203], an LTL-inspired extension of OCL supporting past and future temporal operators, to specify reachability and safety properties of the system. The user initiates the verification process by providing the class diagram that includes a contract for each operation and the TOCL specification. Then, the model checker builds the so-called *Snapshot Transition Model* (STM) that is similar in nature to the filmstrip model discussed above as it describes the state space of the system in terms of *snapshots*. A snapshot, i.e., a single state of the system, contains all active objects, their associations, and their current attribute values. The application of an operation defined by its contracts to a source snapshot yields a transition to a target snapshot. The USE Model Validator [79, 187] verifies the TOCL specification over the STM and searches for sequences of snapshots, i.e., a scenario, that violate the specification. If a counterexample is found, the violating execution trace is visualized as a UML sequence diagram and presented to the user. Note that the search space is bounded by a user-defined *scope* that defines an upper bound on the length of the scenario and an upper bound on the number of objects that the scenario may contain. Thus, this verification approach implements a (symbolic) bounded model checking algorithm that uses the USE Model Validator to translate the problem into a *bounded, relational problem description*, which is subsequently converted into a Boolean formula by Kodkod [198]. This Boolean formula can be solved with any off-the-shelf SAT-solver like MiniSAT [61].

With MocOCL,[14] Bill *et al.* [27] present an enumerative model checker for their CTL-based OCL extension, called cOCL. cOCL is thus far the only temporal extension for OCL that integrates the CTL operators and their semantics seamlessly into the existing formal semantics of OCL and requires no changes to the existing definitions. Their extension introduces six keywords, next, eventually, globally, until, and unless (equivalent to *weak until*), each of which is preceded by a mandatory *path quantifier*, either always or sometimes. These keywords implement the standard CTL semantics of their corresponding temporal operators [42]. To assert the behavioral correctness of a system, MocOCL expects as input an Ecore model that captures the static structure of the system, a set of graph transformations that describe the system's behavior, an initial model that represents the initial state, and a specification formulated in cOCL. MocOCL constructs the state space using Henshin [6]; thus, internally MocOCL uses graphs to represent states. The evaluation of a cOCL specification is performed incrementally. Starting with the initial state, the state space is expanded step-wise by applying the behavior-describing transformations to the most recently expanded states. Then, the cOCL specification is evaluated in the single-step expanded state space. If the specification is violated, MocOCL informs the user of the failure and returns a *cause* to the user that contains a counterexample to the specification. Otherwise, the state space is expanded once more and the specification is evaluated again. This loop continues until the state space cannot be expanded further, i.e., all states have been visited. If the specification still holds, MocOCL reports back the success of the evaluation and, again, returns a cause that

[14]Available from `http://www.modelevolution.org/prototypes/mococl`

explains why the evaluation was successful.

Model Checking of UML Diagrams

Finally, we survey approaches that employ *model checking* in the context of verifying the correctness of *UML models*. Because size and structure of UML leave much room for the application of model checking, much effort has been devoted to the adoption of model checking techniques to UML. Due to the large number of papers falling into this category, we list them separately in Table 3.2 that, in essence, captures all features of the feature model presented above, however, re-arranged to provide for a better overview. Due to the many similarities between the approaches in this category, we refrain from discussing each approach individually, but highlight only their distinguishing contributions. In this section we will often give precedence to the term *diagram* over *software model* in accordance with the UML standard's preference of the former, but in general use the two synonymous.

Verification Goals and Scenarios

In general, model checking of UML models either aims to (a) ensure correct behavior of one diagram, i.e., behavioral correctness, or (b) assert two different diagrams consistent. In Table 3.2, we group the different approaches according to their pursued verification goal and list them in alphabetical order. In general, consistency asserting approaches analyze whether a set of different diagrams describes the overall system in a consistent way, that is, they verify that the information presented in one diagram does not contradict the information of another diagram. Note that we also assign approaches to this category, that use one diagram to define the specification and another diagram to represent the implementation that is required to satisfy the specification, i.e., is consistent with the specification diagram. In contrast, behavioral correctness is usually asserted with respect to a single diagram and its specification that defines the desired or forbidden behavior of the system. These specifications are usually formulated in temporal logic and demand, for example, that the system is free of dead- and livelocks.

Domain Representation

UML as general purpose modeling language is too large as to be supported by any verification approach in its entirety. Therefore, all reviewed works focus on a subset of UML that is essential for the intended application areas. The UML metamodel [152] precisely defines the syntax of the modeling language, i.e., it describes the available language concepts. Further, some semantic aspects are documented, but especially the execution behavior is only informally described. As a precise definition of the meaning of a diagram is essential for the verification, works on model checking UML models often spend a lot of emphasis on describing the semantics of the models under consideration. For example, communication mechanisms, concurrency models, timing specification features etc. have to be introduced concisely. The differences arising from incompatible semantic

Table 3.1: The *Software Model Verification Approach* feature model.

Software Model Verification Approach	***Verification Goal*** Consistency	***Verification Goal*** *Translation Correctness* (xor) Source-Target Analysis	***Verification Goal*** *Translation Correctness* (xor) Transformation Analysis	***Verification Goal*** *Behavioral Correctness* (or) by transformation	***Verification Goal*** *Behavioral Correctness* (or) by operation	***Domain Representation*** (xor) Graphs	***Domain Representation*** (xor) *OMG Standards* (or) UML	***Domain Representation*** (xor) *OMG Standards* (or) MOF	***Domain Representation*** (xor) *OMG Standards* (or) OCL	***Domain Representation*** (xor) *OMG Standards* (or) QVT	***Verification Representation*** (xor) *Logic*	***Verification Representation*** (xor) *Transition System*	***Verification Representation*** (xor) Graphs	***Specification Language*** (or) *Logic*	***Specification Language*** (or) Bisimulation Relation	***Specification Language*** (or) Graphs	***Specification Language*** (or) OCL	***Verification Technique*** (xor) *Theorem Proving* (xor) Automatic	***Verification Technique*** (xor) *Theorem Proving* (xor) Manual/Interactive	***Verification Technique*** (xor) *Model Checking* (and) *State space representation* (xor) enumerative	***Verification Technique*** (xor) *Model Checking* (and) *State space representation* (xor) symbolic/abstract	***Verification Technique*** (xor) *Model Checking* (and) *Property type* (or) Reachability	***Verification Technique*** (xor) *Model Checking* (and) *Property type* (or) Safety	***Verification Technique*** (xor) *Model Checking* (and) *Property type* (or) Liveness
Poernomo, Terrell [159]			✓					✓	✓		H			H					✓					
Poskitt, Plump [161]			✓	G		✓							✓	F		✓			✓					
Stenzel *et al.* [191]			✓					✓			D			D					✓					
Hülsbusch *et al.* [93]		✓				✓							✓		✓				✓					
Ehrig, Ermel [64]		✓				✓							✓		✓				✓					
Giese, Lambers [77]		✓				✓							✓		✓				✓					
Kyas *et al.* [116]					S		✓				H						✓		✓					
Strecker [192][a]			✓	G		✓					H			H					✓					
Schmidt, Varró [176]				G		✓						$\mathcal{L}$				✓				✓		✓	✓	
Baresi *et al.* [11]				G		✓						$\mathcal{L}$				✓				✓		✓	✓	
Baresi, Spoletini [12]				G		✓					R			R						✓		✓	✓	
Kastenberg, Rensink [106][b]				G		✓						$\mathcal{G}$		C		✓				✓	✓	✓	✓	✓
Arendt *et al.* [6][c]				G		✓						$\mathcal{G}$		μ						✓		✓	✓	✓
Narayanan, Karsai [140]		✓				✓							✓		✓					✓		✓		
König, Kozioura [111][d]				G		✓							✓			✓					✓	✓	✓	
Boronat *et al.* [30][e]				Q				✓	✓	✓	W			LW						✓		✓	✓	✓
Gagnon *et al.* [74]					S		✓				W			L						✓			✓	✓
Troya, Vallecillo [199]				A				✓			W			W						✓		✓	✓	
Büttner *et al.* [38][f]			✓				✓		✓		R						✓			✓			✓	
Mullins, Oarga [138]					O		✓		✓			$\mathcal{A}$		C			✓			✓		✓	✓	✓
Gogolla *et al.* [80, 81][f]			✓		O		✓		✓		R						✓				✓	✓	✓	
Al-Lail *et al.* [117]					O		✓		✓		R			L			✓				✓	✓	✓	
Bill *et al.* [27][g]				G				✓				$\mathcal{G}$		C			✓			✓		✓	✓	✓

Legend: *Behavior:* A. . . ATL, G. . . Graph Transformation, O. . . OCL, Q. . . QVT, S. . . State Machine
Logic: C. . . CTL, D. . . Dynamic L., F. . . FOL, H. . . HOL, L. . . LTL, R. . . Relational L., W. . . Rewriting L., μ. . . μ-calculus
Trans. Systems: $\mathcal{A}$. . . ASM, $\mathcal{G}$. . . GTS, $\mathcal{L}$. . . LTS

[a] Isabelle/HOL source files available from `http://www.irit.fr/~Martin.Strecker/Publications/proofs_graph_transformations.tgz`
[b] Available from `http://groove.sourceforge.com`
[c] Available from `https://www.eclipse.org/henshin/downloads.php`
[d] Available from `http://www.ti.inf.uni-due.de/research/tools/augur2/`
[e] Available from `ftp://moment.dsic.upv.es/releases/20070727/`
[f] Available as plugin in USE: `http://sourceforge.net/projects/useocl`
[g] Web interface available at `http://www.modelevolution.org/prototypes/mococl`

interpretations are one reason why the approaches are hard to compare. In the following, we shortly review which diagrams and concepts of UML have been subject to model checking.

Because UML *state machines* are very close to finite automata, it has soon been realized that model checking is a suitable technique to verify their correctness. The basic language concepts supported by most approaches are (*a*) states, including initial and final states; (*b*) transitions, which can be labeled with an event, a guard and a set of effects that represent actions triggering other effects; and (*c*) choices. Hierarchical states (e.g., [78]) and orthogonal states (e.g., [103]) as well as fork and join states (e.g., [78]) are only supported by a few approaches. Zhang and Liu's [202] model checking approach for state machines, for example, integrates all of these language elements.

Apart from number and type of supported language concepts, the various systems differ in their behavioral semantics, which describes in what order events are triggered and dispatched. Usually, concurrent completion events, i.e., events which are automatically triggered when some activity is completed, are forbidden and each event triggers exactly one transition. Further, data processing is not considered, and timing issues are also neglected. Some approaches are based on asynchronous communication [103], while others assume synchronous communication [134]. In case of state machines, the above mentioned incompatible semantic interpretation can be circumvented with Polyglot [9], a tool that translates the different state machine semantics to a common intermediate representation based on the programming language Java prior to performing the verification. The intended semantics, however, have to be implemented in form of pluggable modules. The separation of a model's structure and its semantics allows the combination of state machines from different sources and different tools.

Class diagrams are used to describe the static structure of a system by offering many concepts that are also found in object-oriented programming languages. On their own class diagrams contribute only little to the verification process if not paired with a description of the system's behavior. For example, Ober *et al.* [147] use classes to describe processes whose behavior is specified by state machines. Likewise, the Rhapsody VE [175] specifies a system in terms of class diagrams and state machines. While the classes provide the structure of and the relationship among the elements contained in the systems, the state machines describe the system's behavior. The classes are annotated with the maximal number of its instances allowed during the model checking process. On this basis, the required memory usage is restricted. Ji *et al.* [100] check whether for the scenarios shown in a collaboration diagram all required associations are available in the class diagram. Jussila *et al.* [103] use class diagrams without inheritance relations and operation declarations to model the active objects occurring in a system. They specify the initial configuration, i.e., the initial state of the system, by means of a deployment diagram at the object level.

Interaction diagrams, similar to sequence diagrams, are used to illustrate communication scenarios, i.e., they represent snapshots of interactions. Hugo supports model checking of sequence diagrams against state machines [109]. For this purpose, the sequence diagrams are translated into finite automata. They support a wide range of concepts available in sequence diagrams, among others, partially ordered event occurrences, state invariants, weak and strict sequencing, parallel and alternative composition, loops, as well as the `neg` operator. The content of `neg` fragments is restricted in such a manner that the resulting automaton is deterministic and, hence, can be negated directly. The vUML tool [160] uses sequence diagrams to report counterexamples back to the user, i.e., displays an error trace that allows the reproduction of the error. Lima *et al.* [122] focus on the verification and validation of sequence diagrams containing combined fragments which allow for a compact representation of sets of traces.

Only a few works deal with model checking for *activity diagrams*. The reason for this is probably that prior to UML 2.0 activity diagrams and state machines shared more commonalities than there were distinguishing features. Now, activity diagrams are close in semantics to Petri nets, for which a wealth of literature exists [139]. Eshuis [68] present a symbolic model checking approach for activity diagrams, where activity diagrams are mapped to finite state machines.

Target Representation, Specification Language, and Properties

Almost all approaches use existing verification back-ends in order to achieve their verification goals. Very popular model checkers are SPIN and NuSMV (refer to Table 3.2 for the details). Grumberg *et al.* [82] translate the state machines, for the purpose of verification, to C code, which is than handed to software model checker CBMC [44]. Bounded model checking is applied, but they point out how unbounded model checking might be realized. The UMC framework [195] implements an on-the-fly model checker, i.e., the representation of a state machine as doubly labeled transition system is created on demand in order to deal with the state explosion problem. They use the specification language UCTL [195], a state and event based temporal logic that is tailored towards the verification of UML models. While Mozaffari and Haounabadi [136] propose to translate sequence diagrams to executable, colored Petri nets, on which they perform the verification of the given properties, Shen *et al.* [181] take advantage of verification tools available for abstract state machines [29]. Some approaches translate the UML models to formal languages for which dedicated model checkers are available. In contrast to most other approaches that use high-level intermediate languages of verification systems, Niewiadomski *et al.* [143, 144] directly encode their model checking problem in propositional logic. In a first case study, the authors show that the direct encoding outperforms the approaches that rely on high-level verification systems.

In principal, the specification language of the verification back-end could be used directly to formulate the properties that are checked on the given diagrams. This is, however, often problematic due to the disparity between the UML diagrams, i.e., the domain representation, and the verification representation, i.e., the encoding of the UML diagrams into the input language of the verification back-end. Thus, several concepts for expressing properties in a notation close to the domain representation have been explored. Siveroni *et al.* [183] propose an LTL-based language that introduces additional predicates to ease reasoning on UML class diagrams and state machines with temporal expressions. Ober *et al.* [146] suggest *observer objects* based on UML stereotypes and state machines for specifying properties that should hold. Therefore, they use UML components together with temporal extensions. Also Porres [160] introduces stereotypes into the UML models in order to annotate them with constraints. The specification language column (Spec. Lang.) in Table 3.2 shows whether an approach provides a custom textual or graphical language or if it uses the specification language by the verification back-end.

Summary

Over the last 15 years, many approaches have been presented that aim to increase the quality and specification adherence of UML models by applying model checking techniques. Because the UML standard contains numerous semantic ambiguities, many works show how to resolve these inconsistencies and propose different encodings based on their semantic interpretation. The large number of different semantic interpretations and the non-availability of tools impede a direct comparison of the different approaches. Because most works focus on resolving semantic issues and the efficiency of their encoding, little is said about the practical application scenarios of the proposed verification approaches. It thus comes without surprise that hardly any of the available solutions can be used out-of-the-box in arbitrary application scenarios.

3.2 A Feature-Based Classification

We propose a feature-based view to make different verification approaches comparable. We classify verification approaches by (*a*) the pursued *verification goal* that captures the intention of the verification; (*b*) the *domain representation* that defines the expected input format of the verification approach; (*c*) the *verification representation* that the underlying verification engine uses to perform the actual verification; (*d*) the *specification language* used to describe the relevant properties that the system should satisfy; (*e*) and the *verification technique* that is applied to achieve the verification goal. Some aspects of the classification we use have been captured by previously published surveys, all of which focus solely on the verification of model transformations. For example, Amrani *et al.* [4] propose a tri-dimensional categorization for model transformation verification approaches. They categorize approaches according to (*a*) the type of the model transformations that can be verified, (*b*) the properties of the transformations that can be analyzed including termination as well as syntactic and semantic relations, and (*c*) the employed verification technique. Recently, they presented a generalization of their categorization and introduce a catalog of intents that allows to classify model transformations according to their intended applications, which includes, but is not limited to, verification [3]. Calegari and Szasz [40] re-use Amrani *et al.*'s tri-dimensional categorization and suggest further subcategories for each dimension. Rahim and Whittle [1] classify formal and informal approaches according to the technique employed to assert the correctness of model transformations. In contrast, we consider model transformations as one of

Table 3.2: Model Checking Approaches for UML

		Domain Representation					Spec. Lang.			∗	Prop.	
	Authors	Class Diagram	State Machine	Sequence Diagram	Activity Diagram	Collab. Diagram	Graph. Language	Text. Language	Temporal Logic	Model Checker	Liveness	Safety
behavioral correctness	Balasubramanian *et al.* [9][a]		✓						L	F		✓
	ter Beek *et al.* [195][b]	✓	✓						C	O	✓	
	Del Bianco *et al.* [18]		✓						C	K		✓
	Dong *et al.* [56]		✓						L	S		✓
	Dubrovin, Junttila [57][c]		✓						B	N	✓	✓
	Eshuis [68]	✓			✓				L	N		✓
	Gnesi *et al.* [78]		✓						C	J	✓	✓
	Grumberg *et al.* [82]		✓						L	C	✓	✓
	Jussila *et al.* [103][d]	✓	✓						L	S		✓
	Lam, Padget [118]		✓						C	N		✓
	Lilius, Porres [121]; Porres [160]		✓	✓		✓	✓			S		✓
	Lima *et al.* [122]			✓				✓		S		✓
	Mikk *et al.* [134]		✓						L	S		✓
	Mozaffari, Haounabadi [136]			✓				✓		O		✓
	Niewiadomski *et al.* [143][e]		✓					✓		O		✓
	Oubelli *et al.* [155]			✓					L	S	✓	✓
	Shen *et al.* [181]	✓	✓					✓		A	✓	✓
	Siveroni *et al.* [183]	✓	✓					✓		S		
	Zhang, Liu [202][f]		✓						L	P	✓	✓
consist.	Ji *et al.* [100]	✓	✓			✓	✓			S		✓
	Knapp, Wutke [109][g]		✓	✓			✓			S		✓
	Ober *et al.* [146][h]	✓	✓				✓			I		✓
	Schinz *et al.* [175]		✓	✓			✓			V	✓	✓

Temporal Logics: C. . . CTL, L. . . LTL, B. . . CTL and LTL
Model Checker: A. . . ASM, C. . . CBMC, F. . . Java Path Finder, I. . . IF-tool-suite, J. . . Jack, K. . . Kronos, N. . . NuSMV, O. . . own, P. . . Pat, S. . . SPIN, V. . . VIS

Note: The column titled ∗ corresponds to the *Verification Representation* of our classification.
[a] Available from `https://wiki.isis.vanderbilt.edu/MICTES/index.php/Publications`
[b] Web interface available at `http://fmt.isti.cnr.it/umc/V4.1/umc.html`
[c] Available from `http://www.tcs.hut.fi/Research/Logic/SMUML.shtml`
[d] Available from `http://www.tcs.hut.fi/SMUML/`
[e] Available from `http://artur.ii.uph.edu.pl/zimplit/bmc4uml.html`
[f] Available from `http://www.comp.nus.edu.sg/~pat/`
[g] Available from `http://www.pst.informatik.uni-muenchen.de/projekte/hugo/`
[h] Available from `http://www.irit.fr/ifx/`

many possibilities to specify the behavior of a system and, more generally concentrate on verification approaches that assert if a software model adheres to its specification.

Verification Goal

The verification goal describes the purpose or the intent of the verification. We distinguish between three types of goals: *consistency*, *translation correctness*, and *behavioral correctness*. In the following, we explain and compare the different verification goals. For each goal we provide an example scenario that describes the verification goal in the context of a fictional development process.

Consistency. Approaches that verify the consistency of a set of models, each of which describing a different part of one and the same system, aim to ensure that their intersection, i.e., the parts where the models overlap, does not contain contradicting information. In a multi-view modeling language like UML, where diagrams provide distinct views onto the system under development, developers need tools to assert that the different diagrams are not inconsistent.
Example Scenario: The development team defines the behavior of the system with a set of sequence diagrams. Next, they define the structure of the system and devise corresponding state machines for each class. In such a setting, the system is deemed consistent w.r.t. the sequence diagrams if the message sequences described by each of the sequence diagrams correspond to execution paths in the state machines.

Translation Correctness. When performing model-to-model or model-to-code transformations, the *correctness of the translation* becomes the subject of the verification. The primary correctness criterion among the approaches in this category deems a translation correct w.r.t. the source model, if the target model preserves the semantics of the source model. This requires that both the semantics of the source model and the target model are formally defined.
Example Scenario: The development team generates from a UML activity diagram a Petri net that is used to perform additional verification tasks. Before the analysis with the Petri net can be performed they need to assert the correctness of the *activity-diagram-to-Petri-net* transformation to ensure that all states that are reachable in the activity diagram are reachable in the Petri net, too.

Although the term "translation correctness" suggests that source and target model conform to different metamodels, we also assign approaches to this category that assert the correctness of endogenous transformations.
Example Scenario: When performing model refactorings that alter the structure of the system but not its behavior, developers assert the refinement correctness of the performed changes to ensure, for example, that the target model behaves like the source model in every possible run of the system.

Behavioral Correctness. The behavior of a system is governed by a set of rules. In our classification and the approaches we analyze, these rules are either provided as a set of *model* or *graph transformations*, or as a set of *operation contracts*. Each operation contract is associated with an operation or function provided by the system. It describes the necessary conditions to execute the operation and its effects. Thus, an operation contract consists of a set of preconditions that define in which state of the system the operation can be executed and a set of postconditions that define the state of the system after the operation has terminated. Similarly, a transformation defines application conditions, which control when the transformation can be applied to the source model, and a set of instructions that define the structure of the target model after it has been executed. Hence, under the assumption that a contract's conditions are formulated in first-order logic and a transformation rewrites graph-based structures, transformations and operation contracts are equally expressive and interchangeable. The sequence of states, called a *trace*, resulting from the execution or application of an operation or transformation yield the behavior of the system. Hence, a specification describes the necessary and forbidden traces of a system, often by means of a temporal logic formula. The algorithmic procedure performing the verification compares the system's behavioral description, i.e., the all possible traces resulting from execution, application of the operation contracts, or transformations, with the specification.
Example Scenario: The development team designs a new security protocol and models the behavior of the two communicating agents and the behavior of the attacker with graph transformations. They want to ensures that no attacker can hijack a secured channel and formulate the specification accordingly as an LTL formula. The number of interactions between the agents and the attacker is finite, however, large. Thus, they use a model checker to assert that the transition system, which captures the interaction of the agents and the attacker, satisfies the protocol's specification.

Discussion. To clarify the classification we highlight the discriminating features between translation and behavioral correctness in the following. Approaches that target the behavioral correctness always analyze endogenous transformations that may not terminate. Translation correctness approaches analyze both exogenous and endogenous transformations, but demand that these transformations terminate. Obviously, a translation of a source to a target model requires a result and thus a definite end; otherwise, we would have identified an error in the translation, i.e., a source model, on which the translation diverges. A system, whose behavioral correctness we want to verify, may, in contrast, continue to expose its behavior indefinitely; and hence, some of the transformations that describe the system's behavior may be applicable over and over again. Further, we observe differences in the way specifications are phrased. Translation correctness aims to assert that the semantics of the source model are preserved by the target model, that is, the properties that hold in the source model should still hold in the target model after the execution of the transformation. On the contrary, the specifications for behavioral correctness express system properties over traces, i.e., sequences of states, and often use temporal logics to formally describe these properties.

Domain Representation

The input or *domain representation* defines type and format of the source model(s) that the verification approach is able to analyze. We distinguish between *graph-based* representations and representations that use notations and visualizations defined in an *OMG standard*. Simple graphs can be enhanced with different constructs to raise their expressivity. They can be labeled [83], typed, or attributed and may support inheritance relations or compositions (also: part-of relations) [24]. Approaches that use the notation of an OMG standard may use elements or combinations of UML [153], MOF [150], QVT [149], or OCL [148].

Verification Representation

The *verification representation* classifies the approaches according to the formal representation that is used to perform the verification. We distinguish between *logical*, *state-transition*, and *graph-based* representations. As most approaches do not implement their own verification back-end, this representation correlates with the input language of the underlying verification tool. For example, approaches that employ MAUDE [45] represent models as algebraic data types such that MAUDE's search and model checking capabilities may be used to verify the system. Approaches based on ALLOY analyzer [96] or Kodkod [198] convert models and transformations into relational declarations and predicates.

Logical verification representations can be partitioned into approaches using higher-order logic (HOL) [119], first-order logic [184], dynamic logic [84], rewriting logic [133], relational logic [95], or temporal logics, e.g., CTL [42], LTL [158], or μ-calculus [114]. Likewise, different kinds of state-transition systems are in use and the classification of approaches can be further refined according to their use of either labeled transition systems (LTS), graph transition systems (GTS), or abstract state machines (ASM). The approaches that use a graph-based representation usually use a combination of extensions, e.g., types, attributes, and inheritance relations, to increase the precision of the verification.

Specification Language

Different *specification languages* for expressing the properties to be checked are in use with varying degrees of expressivity. We distinguish between *logical*, *bisimulation-based*, and *graph-based* specifications. In addition, we list OCL explicitly due to its relevance as a specification language in MDSD. The sub-category of logical specification languages is further divided into approaches that specify system properties with higher-order logic, first-order logic, dynamic logic, rewriting logic, relational logic, or temporal logics (CTL, LTL, μ-calculus). A *bisimulation* is an equivalence relation that asserts whether two automata can simulate each others moves on the same input. Basically, two automata are declared *bisimilar* if there exists a bisimulation relation $\mathcal{R}$, where a pair (a, b) of states from automaton A and B is in $\mathcal{R}$ if automaton B can replicate every move $a \rightarrow a'$

by automaton A, for some state a', and automaton A can replicate every move $b \rightarrow b'$ by automaton B, for some state b', and the pair (a', b') is again in $\mathcal{R}$ [135]. In general, this relation is stronger than language equivalence, i.e., whether two automata accept the same language [135]. Graph-based specification languages define system properties by means of graph constraints, which are, essentially, graph transformations whose LHS and RHS are identical. Thus, they do not alter the system and, if their applicability is asserted, the system is declared correct w.r.t. to the constraint.

Verification Technique

Finally, we categorize approaches according to the verification technique they employ and assign them either to the category of *theorem proving*-based techniques or to the category of *model checking*-based techniques. Once assigned to either of the two verification techniques the capabilities and limitations of the different approaches become comparable with regard to the logical models and properties they can verify. In particular, theorem proving-based approaches can verify systems with infinitely many different states, but they usually require manual guidance by an expert user. Model checking-based approaches, on the contrary, are fully automatic, but can only verify finite state system descriptions. There exist, however, automatic theorem provers that either check the satisfiability of logical propositions modulo decidable first-order theories or the satisfiability of classical first-order logic, in which case the search of a proof may not terminate. Hence, we classify theorem proving-based approaches into *automatic* and *manual/interactive* approaches.

We refine the classification of model checking-based approaches by their *state space representation* and by the *type of properties* that can be verified. If the state space is explored *enumeratively*, every possible combination of different valuations for the state-defining properties is analyzed. Contrary, *symbolic* state space representations use (propositional) logic to represent states and transitions. Likewise, *abstract* state space representations use the theory of abstract interpretation [48] to conservatively over-approximate the set of possible system states. Concerning the supported types of properties, we record for each model checking-based approach whether it supports the verification of *reachability*, *safety*, or *liveness* properties.

The Feature Model

The classification described above is reflected in the feature model [104] depicted in the left half of Table 3.1. In the following presentation we use a tabular representation for our feature model, that compactly mirrors the commonly used tree-based representation (cf. [50]). The root feature, named *Software Model Verification Approach*, is decomposed into five main features named verification goal, domain representation, verification representation, and so on. These main features are further refined according to our classification described in the previous section. Note that all features in the table are mandatory. Names written in *italic* denote abstract features that are further refined by either *and*, *or*, or *xor* decompositions. An *and* (*or*, *xor*) decomposition mandates that each (at least one, exactly one) of the child features is present, used, or implemented in the verification approach in order to be classified successfully. For example, the *Verification Goal* feature is *or*-decomposed into the *Consistency*, the *Translation Correctness*, and the *Behavioral Correctness* feature. The latter is in turn *xor*-decomposed into the *Behavior by Transformation* and the *Behavior by Operation* features. Hence, an approach that asserts the behavioral correctness encodes the behavior into transformations or operation contracts. Moreover, we introduce *multi-valued features* to increase readability of the feature model. A multi-valued feature is equivalent to an abstract feature containing a child feature for each of its possible values. Thus, a multi-valued feature is always abstract and written in *italic*. For example, the multi-valued feature *Transition System* listed under the main feature *Verification Representation* has three different values: Labeled Transition System (LTS), Graph Transition System (GTS), and Abstract State Machine (ASM). Each of the possible values of a multi-valued feature is listed in the "Legend".

The right of Table 3.1 shows the classification presented in Section 3.2. This part of the table is read as follows. A check-mark in the table indicates that the feature is supported and, in case of multi-valued features, the actual value is displayed in parentheses. Approaches providing an implementation are underlined.

We purposefully deviated from the restriction governing the *xor*-decomposition in the case of Groove [106], which supports both an enumerative traversal and, since recently, an abstraction-based traversal of the state space. Although the techniques employed by Groove to implement enumerative and abstraction-based model checking are conceptually different, we decided to merge the two verification approaches into one entry because they coincide on every feature but the state space representation. Further, due to the many similarities among the model checking-based approaches for UML models, we decided to only list a representative assignment of features in the last column of Table 3.1 for all with model checking-based verification approaches that verify the consistency and behavioral correctness of UML models. In Section 3.1 we provide a more fine-grained comparison of these approaches, which are then summarized in Table 3.2.

3.3 Summary

Progress and success of formal verification techniques in hardware design and software engineering has motivated the MDSD community to adopt and apply such techniques for the verification of software models. To this end, much research efforts have been spent on lifting verification techniques from hard- and software to MDSD.

In this chapter, we surveyed the efforts made to apply formal verification techniques in the MDSD development process. We established a feature model that relates the characteristic properties of different verification approaches. By this means, we are able to categorize the different approaches in a concise manner. Overall, formal verification techniques have been used extensively in all areas of MDSD. Compared to works on formal verification in hardware and software systems, the works in MDSD are often in a very early stage and work in progress. Therefore, it is hard to give general recommendations which techniques and tools to apply for a given verification problem beyond those of the provided features. Also there is no established evaluation methodology to judge on the effectiveness of the different approaches. Nevertheless, already at this early stage the vast literature on formal verification techniques in MDSD illustrates their huge potential.

Based on the insights gained from the literature review and the subsequent classification we draw the following conclusions:

- formal verification techniques of semantic properties based on model checking and interactive theorem proving have been applied extensively to all areas of MDSD;
- compared to formal verification methods developed for hard- and software, the majority of the proposed approaches for the verification of software models is still in its infancy and (prototypical) implementations are pending;
- a large scale evaluation of the effectiveness of the proposed approaches is, at its current state, impossible due to incompatible semantic interpretations of, e.g., the UML standard [69,182], and further hindered by the lack of a common benchmarks;
- the large amount of literature on formal verification techniques in MDSD illustrates, however, their huge potential.

CHAPTER 4

Model Checking CTL-extended OCL Specifications

The first part of this chapter presents a temporal extension for OCL based on the Computation Tree Logic (CTL) [42]. This extension for OCL, called cOCL (short for CTL-*extended* OCL*)*), allows reasoning over sequences of states instead of just a pre-state and a post-state as OCL is capable of. The verification approach presented in the second part of this chapter implements an explicit-state model checker for cOCL specifications, called MocOCL (short for *Model Checking* CTL-*extended* OCL), that iteratively enumerates the set of all reachable states. These states are represented explicitly, that is, internally each state is stored as an object diagram and contains the actual values used to describe the real-world system.

Overview. In MBD, the Object Constraint Language (OCL) [148] is a popular specification language. It allows to constrain MOF based models with *invariants* and operation *contracts*, which specify pre- and postconditions for operations defined in these models. The OCL thus provides means to specify *functional behavior properties* [86] of at most two consecutive system states, i.e., the pre- and the post-state. Yet, temporal operators are necessary to observe and verify the behavior over a sequence of states. As OCL does not support the specification of temporal aspects beyond the post-state, we present the CTL-extended Object Constraint Language, or cOCL for short. This extension introduces the branching time operators from CTL to OCL, which allows us to capture the desired behavior of a system over sequences of system states. Specifications that use our extension can be verified with our explicit-state model checker.

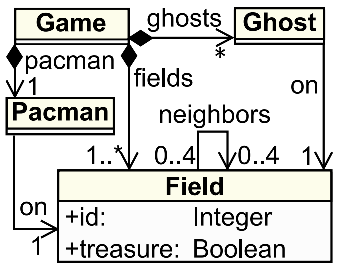
The verification of a system against its specification with our model checker for CTL-extended OCL (MocOCL for short) proceeds in seven steps. First, the modeler defines a model of the system that captures its static structure. Then, a set of model transformations is developed that specifies the behavior of the system. Next, the specification is formulated and an initial state is defined in terms of an object diagram. All of these artifacts comprise the input to the model checker that constructs the so-called state space by applying the model transformations to the initial state and successively to all models resulting from previous applications of the model transformation. For reasons of efficiency, the model checker will avoid the generation of the entire state space, which might grow huge even for small verification tasks. Thus, the model checker evaluates the specification *on-the-fly* while constructing the state space. The model checker may then stop as soon as the specification is found to hold or abort if the specification is violated. In the latter case, the model checker will extract a *counter-example* that demonstrates an execution trace of the system that violates the specification.

Running example. To illustrate the concrete syntax of cOCL (see Sec. 4.3 of this chapter) and to demonstrate a verification scenario with MocOCL (see Sec. 4.4) we present in the following a variation of the well-known Pacman game[1] that we will re-use during the evaluation discussed in Chapter 6. Our variation of the Pacman game is played on a board consisting of fields, each of which has at most four neighboring fields. Each field has a unique identification number. Some fields contain a treasure as indicated by a Boolean flag. Pacman plays against one or more ghosts. Each player, Pacman or the ghosts, is placed on one field of the board. The static structure of the game's implementation is shown in Figure 4.1a. The root element `Game` *contains* exactly one `Pacman` element, one or more `Field` elements, and zero or more `Ghost` elements. Each player, i.e., the Pacman and each ghost, is placed on exactly one field. A field is identified uniquely by its `id` attribute and has at most four neighboring fields. The Boolean `treasure` attribute indicates whether the field contains a treasure. Figure 4.1b uses the graphical, concrete syntax and shows a Pacman game instance with four fields, a treasure on field 4, Pacman on field 1, and a ghost on field 3. Note that an instance of the Pacman game, as just described, defines a particular *state* while playing the game, which is done as follows. The players move turn-wise in no fixed order. Pacman has to find a treasure, which is placed somewhere on the board. If Pacman finds one, he wins the game. If, however, Pacman moves onto a field with a ghost or if a ghost moves onto Pacman's field, Pacman looses the game. The game ends, i.e., no more moves may be made by the players, if either Pacman wins or looses the game.

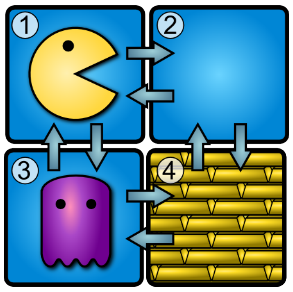
Related Work. We briefly summarize two lines of related works; first, those that extend OCL with temporal operators and, second, those that suggest a model checking based verification approach to verify temporal extensions of OCL.

Distefano *et al.* [55] present a CTL based logic, called BOTL, to specify static and dynamic properties of object-oriented systems. But instead of extending OCL, they map OCL onto BOTL; thus, they provide formal semantics for a large part of OCL based on BOTL. Ziemann and Gogolla [203] formalize an extension based on linear time logic, which is similar in nature to our CTL based solution. Bradfield *et al.* [33] embed OCL into the observational μ-calculus. They suggest the use of predefined templates

[1]`http://en.wikipedia.org/wiki/Pac-Man`



(a) Static structure of the Pacman game



(b) Instance of a Pacman game

Figure 4.1: Abstract and concrete syntax of the Pacman game

with intuitive semantics, from which the underlying μ-calculus formula is automatically generated.

Mullins and Oarga [138] present EOCL, an extension inspired by BOTL, that augments OCL with CTL operators. The operational semantics of EOCL are defined over object-oriented transition systems. They implement SOCLe, a tool that translates class, state chart, and object diagrams into an abstract state machine and checks on-the-fly if the system satisfies a given EOCL specification. Al-Lail *et al.* [117] describe systems with class diagrams and the operations' contracts, given by OCL pre- and postconditions, capture the behavior of the system. They implement a bounded model checking approach. Given an initial instance of the class diagram their tool generates sequences of instances up to a predefined depth with the USE Model Validator [187]. They use TOCL [203] to specify reachability and safety properties. Similarly, Hilken *et al.* [89] employ the USE tool to generate a depth-bounded *filmstrip* model from a UML class diagram and a set of OCL operation contracts. The filmstrip model describes a finite sequence of instances, each of which conforms to the given UML class diagram. Except for the last instance in the sequence, each instance that satisfies a precondition of an operation contract has a successor instance that satisfies the postcondition of the operation contract. The filmstrip model allows to check temporal properties of the system up to a certain bound, but the authors neither define nor restrict themselves to a temporal language for this task. The GROOVE framework [106] verifies object-oriented systems modeled as attributed, type graphs with inheritance relations. It is similar to MocOCL in that it represents system states as graphs and the system's behavior by graph transformations. But, in contrast, it uses standard CTL and LTL to formulate the system's specification.

Bibliographic Note. The OCL extension cOCL and the model checker MocOCL were first presented in [27]. A revised and extended version reflecting the current state of cOCL and MocOCL was later published in [28]. The implementation of MocOCL and the usability evaluation were first presented in [25]. Further, a demonstration of MocOCL was given at the VOLT2014 workshop [26] and at the TAP2014 conference [71].

4.1 Preliminaries

Model checking based techniques require a formal representation of the system and its behavior. Because both the UML standard [153] and the MOF standard [150] provide formal semantics only to a limited extent we use the algebraic theory of graph transformations to describe formally the system's structure with graphs and the system's behavior with graph transformations. In the following we extend the graph transformation theory presented in Section 2.2 and introduce attributed, typed graphs with inheritance and composition relations. We use the formalization of EMF models presented by Biermann *et al.* [24] to present formal syntax and semantics of OCL. This presentation is based on the work of Richters and Gogolla [170].

Algebraic Graph Transformations

Recall from Section 2.2 that a graph is quadruple $G = (V_G, E_G, src_G, tgt_G)$ with a set of vertices V_G, a set of edges E_G, and a source and a target function, $src_G : E_G \to V_G$ and $tgt_G : E_G \to V_G$, respectively. A graph morphism $f : G \to H$ between graphs G and H is a pair (f_V, f_E) of mappings $f_V : V_G \to V_H$ and $f_E : E_G \to E_H$ that preserve the source and target mapping [62] such that for all edges $e \in E_G$

$$\begin{aligned} f_V \circ src_G(e) &= src_G \circ f_E(e) \\ f_V \circ tgt_G(e) &= tgt_G \circ f_E(e). \end{aligned}$$

A node (or edge) is an *image* $h = f(g) \in V_H \cup E_H$ of a node (or edge) $g \in V_G \cup E_G$ if there exists a mapping from g to h, i.e., $(g, h) \in f$. We also write $m(G)$ to denote all images of G. A morphism is *injective* if every node in H is the image of at most one node in G, otherwise it is *surjective.* We write $f : G \hookrightarrow H$ to emphasize that a morphism is injective; otherwise, if the mapping is irrelevant, we simply write $f : G \to H$ as before. A double pushout (DPO) *graph transformation* (also: *rewriting rule*, *graph production*) $p : L \xleftarrow{l} K \xrightarrow{r} R$ describes declaratively how a graph L, the left-hand side (LHS), is rewritten into a graph R, the right-hand side (RHS) over a common interface graph K. The *interface graph* K defines the nodes that are *preserved* by transformation p, whereas all nodes in L that are not in the image of $l(K)$ are *deleted* by p and, reciprocally, all nodes present in R that are not an image of $r(K)$ are *created* by p. A *pushout* of a pair of morphisms $f : A \to B$ and $g : A \to C$ is a pushout graph G together with a pair of morphisms $f' : C \to G$ and $g' : B \to G$ with $f' \circ g = g' \circ f$ and, for all graphs X, if $f'' : C \to X$ and $g'' : B \to X$ such that $f'' \circ g = g'' \circ f$, then there exists a unique morphism $x : G \to X$ such that $x \circ f' = f''$ and $x \circ g' = g''$ (see Fig. 4.2a).

We say that a DPO graph transformation p is *applicable to* a graph G, called the *host graph*, if there exists (i) a morphism $m : L \to G$, called the *match*, that maps graph L of $p : L \xleftarrow{l} K \xrightarrow{r} R$ to G, (ii) a graph D and a pair of morphisms $k : K \to D$ and

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] \arrow[dr, phantom, "="] & B \arrow[d, "g'"] \arrow[ddr, bend left, "g''"] & \\
C \arrow[r, "f'"] \arrow[drr, bend right, "f''"'] & G \arrow[dr, dotted, "x"] \arrow[d, phantom, "="] \arrow[r, phantom, "="] & \\
 & & X
\end{tikzcd}$$

(a) Pushout construction

$$\begin{tikzcd}
L \arrow[d, "m"] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "n"] \\
G & D \arrow[l, "f"] \arrow[r, "g"'] & H
\end{tikzcd}$$

(b) Application of a graph transformation

$$\begin{tikzcd}
L \arrow[rr, "ac"] \arrow[dr, "m"'] & = & AC \arrow[dl, "c", "/" marking] \\
 & G &
\end{tikzcd}$$

(c) NAC

$f : D \to G$ with $m \circ l = f \circ k$ such that *(1)* in Fig. 4.2b is a pushout with pushout graph G together with morphisms m and f. Further, the transformation p may only be applied if both the *dangling edge condition* and the *identification condition* are satisfied. The *dangling edge condition* states that, after deleting a node, all edges adjacent to the deleted node must be deleted as well by the transformation such that no edge is left in a *dangling* state. The *identification condition* demands that, for any two distinct nodes (or edges) v, w of the LHS graph L of p, if a surjective match m maps both v and w to the same node (or edge) in host graph G, i.e., $m(v) = m(w)$, then either both are deleted or both are preserved. A graph transformation is *applied to* a graph G, thereby producing graph H, the so-called *result graph*, by removing those images $m(L)$ from G that are not preserved, that is, all nodes in L that are not an image of $l(K)$, and by adding all images of $n(R)$ to H that are created by the transformation.

In the following, we will assume that a graph transformation is a applied following the DPO approach outlined above, but we will drop the interface graph K and use the more intuitive notation $p : L \to R$ when we refer to a DPO graph transformation provided that the presentation remains unambiguous.

A graph transformation $p : L \to R$ can be further restricted by *negative application conditions* (NAC) or *positive application conditions* (PAC). An application condition consists of a graph AC and a morphism $ac : L \to AC$, i.e., a mapping of nodes and edges from the transformation's LHS to the application graph. A NAC is satisfied if its graph AC cannot be mapped injectively into the host graph G. This is the case if there exists no morphism $c : AC \hookrightarrow G$ such that $c \circ ac = m$ (see Fig. 4.2c). A PAC, in contrast, demands that such a morphism c does exist.

Finally, an *amalgamated graph transformation* matches and transforms with a single application multiple occurrences of a recurring graph pattern. An amalgamated graph transformation consists of a *kernel rule* and multiple *multi rules*. A kernel rule defines the common structure shared by all multi rules and is used to synchronize the application of the multi rules. Thus, the *kernel rule* matches a host graph exactly once, while the multi rules may be applied to the host graph *zero* or more times. An amalgamated graph transformation is defined by a triple (p_K, P_{multi}, em) with kernel rule p_K, multi rules $P_{multi} = \{p_i \mid 1 \leqslant i \leqslant n\}$, and a set of embeddings $em : p_{kernel} \hookrightarrow p_i$ that embed the kernel rule into the multi rules such that the intersection of the left-hand sides of two multi rules overlap only on elements found in the LHS of the kernel rule [23]. An application of an amalgamated rule applies the kernel rule p_K at a match m_K and the multi rules $\{p_i \mid 1 \leqslant i \leqslant n\}$ as often as there exists match $m_i : L_i \to G$ that overlaps with any other multi rule match m_j only at the kernel match m_K [23].

Operationally, a graph transformation DPO is applied to a host graph G as follows. First, search in G for an occurrence of L. If such an occurrence exists and all application conditions are satisfied, remove all nodes of G that are in L but not in R. This yields graph D (see Fig. 4.2b). Next, for every node in R that is not in L, create a node in D and attach these newly created nodes to the preserved nodes in D accordingly. This yields the resulting graph H.

Various extensions for the above presented theory of graph transformations have been developed. In the following, we briefly discuss those relevant for subsequent chapters.

Type graph. A graph may be typed over *type graph* $TG = (V_{TG}, E_{TG}, src_{TG}, tgt_{TG})$ that defines a set of well-formedness constraints. A graph that satisfies these constraints is said to be *typed over* TG and is called an *instance graphs* of TG or simply a *typed graph*. A type graph defines the static structure of nodes and edges that its instance graphs must conform to. This conformance relation is expressed by a *type morphism* $t_G : G \to TG$. A typed graph morphism $f : G \to H$ between typed graphs G and H must be type compatible, i.e., $t_H \circ f = t_G$ where $t_H : H \to TG$.

Inheritance graph. To express subtype relations between nodes of a type graph $TG = (V_{TG}, E_{TG}, src_{TG}, tgt_{TG})$, we introduce the *inheritance graph* $I = (V_I, E_I, \mathfrak{A})$ with $V_I = V_{TG}$ and $\mathfrak{A} \subseteq V_{TG}$ being the set of abstract nodes. The set E_I of inheritance edges, $E_I \subseteq V_I \times V_I$, defines a subtyping relation $subtypes_I$ that is defined as the smallest set $subtypes_I = \bigcup_{i \geqslant 0} subtypes_i$ with

$$\begin{aligned} subtypes_0 &= E_I \\ subtypes_{i+1} &= E_I \cup \{(v,w) \mid (v,x),(x,w) \in subtypes_i\}. \end{aligned}$$

On the basis of this subtyping relation we define for each node $v \in V_I$ the *inheritance clan* as the set of all sub-nodes of v, i.e., $clan_I(v) = \{w|(v,w) \in subtypes_I\} \cup \{v\}$. The typing morphism $ctm : G \to TGI$ between an instance graphs G and a type graph TGI with inheritance assigns each node and edged in G to a type in TGI such that the inheritance clan is preserved, e.g., $ctm \circ src_G(e) = clan_I(src_{TG} \circ ctm(e))$ for all $e \in E_G$. If we flatten the inheritance graph we obtain either the *abstract* or the *concrete closure* of the type graph with inheritance depending on whether abstract nodes are included or excluded from the flattened graph. We may reuse all of the established theory on the closure of type graph with inheritance. Moreover, it can be shown that there exists a bijective correspondence between the type morphism $type : G \to \overline{TGI}$ defined on the closure $\overline{TGI}$ of the type graph with inheritance and the clan type morphism ctm [62].

Attributes. The combination of a type graph and a data algebra allows us to define attributes for nodes $v \in V_G$. We refer to typed graphs with data or attribute nodes as *attributed type graph.* An algebra $\mathcal{A}$ consists of a data signature A, which defines data sorts and a set of function symbols, and an interpretation of the function symbols. Thus, an algebra $\mathcal{A}$ defines the set of values that may be assigned to the data or *attribute nodes* V_A, where $\mathcal{A} \subseteq V_A$. We introduce additional attribute edges E_A that associate a node with a value node $a \in V_A$. An attributed graph morphism $f : G \to H$ between two attributed graphs G, H must be type compatible and attribute compatible, i.e., $\forall a \in V_{A_G} : f(a) \hookrightarrow V_{A_H}$.

Containment. *Containment edges* model part-of relations and are are defined as a distinct set of containment edges $C_G \subseteq E_G$ such that (i) every node $v \in V_G$ has at most one container and (ii) no node is transitively contained by itself. A graph with containment edges that satisfies these constraints is said to be *consistent.* These constraint do not, however, apply to type graphs. For example, a type graph with inheritance edges may specify a containment edge to the top most node of an inheritance hierarchy, which obviously achieves the effect of propagating the containment edge down the hierarchy if the graph is flatten (as described above). Moreover, the type graph may define a containment loop between one or more nodes. Since these edges might result in a containment cycle in the instance graph, they are called *cycle capable* [24]. For example, in a type graph that represents a relational database schema we may nest tables such that a table may contain a sub-table, which in turn may contain another sub-table. To model this scenario we add a containment edge whose source and target are the *Table* node. To maintain the consistency of instance graphs with containments all graph transformations are required to satisfy the following constraints:

- A contained node is always deleted with its containment edge.
- The deletion of a containment edge requires the deletion of its contained node.
- Every created node is immediately connected to its container.
- The creation of a containment edge requires either the creation of a new node or the connection to an existing node whose container is changed.
- For cycle capable containment edges, the contained node may only change its container if the current and the new container are already transitively connected by containment edges.

We will in the following assume that all graphs with containment edges (not type graphs) are *rooted*, i.e., there exists *root* node that transitively contains all other nodes.

The combination of these extensions formalize EMF models as attributed, typed graphs with inheritance and containment relations [24]. To complete the picture we add the set $\mathcal{E} \subseteq (\mathbb{S} \times \mathbb{S} \times \mathbb{Z})$ of enumerations, where each enumeration is identified by a string valued name $n \in \mathbb{S}$ that is associated with a set of key-value pairs $l \in (\mathbb{S} \times \mathbb{Z})$, called *enumeration literals.* We define the terms metamodel, model, and instance model as follows and define a three-layered hierarchy, where the instance of a metamodel is a model, and an instance of a model is an instance model.

Definition 1 (Metamodel and model) *A (meta)model $\mathbb{M}$ is defined by the tuple $(T, I, \mathcal{A}, \mathcal{E}, type_{\mathbb{M}}, root_{\mathbb{M}})$ with (type) graph $T = (V_T, E_T, src_T, tgt_T)$, inheritance graph $I = (V_I, E_I)$ where $V_I = V_T$ and $E_I \subseteq V_I \times V_I$, algebra $\mathcal{A}$, enumerations $\mathcal{E}$, root node $root_{\mathbb{M}} \in V_T$, and type morphism $type_{\mathbb{M}} : T \to T$, which states that graph T is typed over itself. For graph T we further define attribute nodes $V_A \subseteq V_T$, attribute edges $E_A \subseteq E_T$ s.t. $\forall e \in E_A : src_T(e) \in V_T \backslash V_A$ and $tgt_T(e) \in V_A$, and containment edges $C \subseteq E_T$.* □

Definition 2 (Instance model) *An instance model M of $\mathbb{M} = (T, I, \mathcal{A}, \mathcal{E}, type_{\mathbb{M}}, root_{\mathbb{M}})$ is a tuple $(G, type_M, root_M)$ with instance graph $G = \{V_G, E_G, src_G, tgt_G\}$, root node $root_M \in V_G$, and type morphism $type_M : G \to T$ s.t. $type_M(root_M) = root_{\mathbb{M}}$.* □

In the following, we denote by $\mathbb{M}$ the set of instance models M of $\mathbb{M}$ and we write $M \in \mathbb{M}$ if M is an instance of model $\mathbb{M}$ and say that M *conforms to* model $\mathbb{M}$. In general, we refer to elements of $\mathbb{M}$ or M by $\mathbb{M}|_X$ or $M|_X$ with $X = \{T, I, \mathcal{A}, \mathcal{E}, root_{\mathbb{M}}\}$ or $X = \{G, root_M\}$, respectively. Moreover, for convenience, we denote by $\mathbb{M}|_{V_T}$ and $\mathbb{M}|_{E_T}$ the set V_T of vertices and the E_T of edges of the (type) graph T underlying model $\mathbb{M} = (T, I, \mathcal{A}, type_M, root_M)$, respectively, and likewise for models $M \in \mathbb{M}$, i.e., $M|_{V_G}$ and $M|_{E_G}$ denote the set V_G of vertices and the E_G of edges of the graph G underlying model $M = (G, type_M, root_M)$, respectively. We write $v \in \mathbb{M}|_{V_T}$ if v is a node in graph T of $\mathbb{M}$, i.e., $v \in V_T$, and, similarly, we write $v \in M|_{V_G}$ if v is a node in graph G of M, i.e., $v \in V_G$. Likewise for edges, we abbreviate $e \in E_T$ or $e \in E_G$ to $e \in \mathbb{M}|_{E_T}$ or $e \in M|_{E_G}$, respectively.

Syntax and Semantics of OCL

On the basis of the above defined EMF (meta-)model $\mathbb{M} = (T, I, \mathcal{A}, \mathcal{E}, type_{\mathbb{M}}, root_{\mathbb{M}})$ we introduce the core features of OCL required for the rest of this thesis. We base our presentation of OCL syntax, semantics, and typing rules on the work of Richters and Gogolla [170].

OCL expressions are always defined w.r.t. a model consisting of classes and associations between classes. Each class contains a set of attributes and operations. Such a model provides the basis for a signature $\Sigma_{\mathbb{M}}$ over which the set of valid OCL expressions is defined. In the following, we first define the set of types induced by a model $\mathbb{M}$ before we formally introduce the notion of an OCL signature $\Sigma_{\mathbb{M}}$.

Definition 3 (OCL types, type closure) *The set of* types $\mathbb{T}_{\mathbb{M}}$ *induced by a model $\mathbb{M} = (T, I, \mathcal{A}, \mathcal{E}, type_{\mathbb{M}}, root_{\mathbb{M}})$ with type graph $T = (V_T, E_T, src_T, tgt_T)$ is defined as the smallest set*

$$\begin{aligned} \mathbb{T}_{\mathbb{M}} = {} & V_T \cup E_T \\ & \cup \{Bool, Int, Real, String\} \cup \bot \\ & \cup Enum \cup Collection(t)_{t \in \mathbb{T}_{\mathbb{M}} \backslash \bot} \text{ where} \end{aligned}$$

- *the* primitive types *Bool, Int, Real, and String denote Boolean, integer, real, and string values,*

- *the* void type $\bot$ *denotes an undefined value,*
- *the* enumeration type *Enum denotes the set of enumerations, which are sets of key-value pairs identified by the name of the enumeration,*
- *and the* collection type *Collection*(t) *denotes the set of all subsets containing objects of type* $t \in \mathbb{T}_{\mathbb{M}}$*; thus, a collection type over* t *is equivalent to the power set* $\mathcal{P}(t)$.

The closure $\mathbb{T}^*_{\mathbb{M}}$ *of* $\mathbb{T}_{\mathbb{M}}$ *is defined as the smallest set* $\bigcup_{i\geqslant 0} \mathbb{T}_i$ *where*

$$\begin{aligned} \mathbb{T}_0 &= \mathbb{T}_{\mathbb{M}} \\ \mathbb{T}_{i+1} &= \mathbb{T}_{\mathbb{M}} \cup \{\alpha \to \beta \mid \alpha \in \mathbb{T}_i, \beta \in \mathbb{T}\} \end{aligned}$$

□

In the style of functional programming languages we use *curried* function types instead of the tuple based types usually found in mathematical notations. We thus treat the function type $\alpha_1 \to \cdots \to \alpha_n \to \beta$ as notationally and functionally equivalent to $\alpha_1 \times \cdots \times \alpha_n \to \beta$. Note that we may use $\alpha \to \mathit{Collection}(\beta)$, which is equivalent to $\alpha \to \mathcal{P}(\beta)$, to express a relation type $\alpha \times \beta$.

Definition 4 (OCL Signature w.r.t. a model) *An* OCL signature $\Sigma_{\mathbb{M}}$ with respect to a model $\mathbb{M} = (T, I, \mathcal{A}, \mathit{root}_{\mathbb{M}}, \mathit{type}_{\mathbb{M}})$ *is a tuple* $\Sigma_{\mathbb{M}} = (\mathbb{T}_{\mathbb{M}}, \mathbb{O}, \mathbb{F}_{\mathbb{M}}, \mathbb{V})$ *where* $\mathbb{T}_{\mathbb{M}}$ *is the set of types as defined in Def. 3,* $\mathbb{O}$ *is a set of predefined operation symbols with a given arity,* $\mathbb{F}_{\mathbb{M}}$ *is the set of binary relations called the features of* $\mathbb{M}$*, and* $\mathbb{V}$ *is the set of countable-infinite, typed variables.* □

The set $\mathbb{O}$ of predefined operation symbols contains, among others, the arithmetic operators for integers and reals, i.e., addition (+), subtraction (−), multiplication (∗), and division (/), Boolean operators for conjunction and negation, $\wedge$ and $\neg$, and the collection operator get that returns from a collection xs the pair (x, xs') consisting an element $x \in xs$ and a collection $xs' = xs \backslash x$. The set of features $\mathbb{F}_{\mathbb{M}}$ represents the set of attribute and reference relations; thus, it is equal to set E_T of $\mathbb{M}$. In the following, we will omit the reference to model $\mathbb{M}$ in $\mathbb{T}_{\mathbb{M}}$ and $\mathbb{F}_{\mathbb{M}}$ and only write $\mathbb{T}$ and $\mathbb{F}$, respectively, if the reference to $\mathbb{M}$ is unambiguously deducible from the context.

On the basis of an OCL signature $\Sigma_{\mathbb{M}}$, we define in the following, first, the abstract syntax, then a set of typing rules, and finally the semantics of the set of OCL expressions w.r.t. an instance model $M \in \mathbb{M}$, denoted by OclExpr. We start with the definition of the set of numeric OCL expressions, which are a subset of OclExpr.

Definition 5 (Numerical OCL expressions) *Let* $\Sigma_{\mathbb{M}} = (\mathbb{T}, \mathbb{O}, \mathbb{F}, \mathbb{V})$ *be an* OCL *signature over model* $\mathbb{M}$ *and let* $op_{Num} \in \{+, -, *, /\} \subseteq \mathbb{O}$*. Then, the set* NumOclExpr *over* $\Sigma_{\mathbb{M}}$ *is inductively defined as follows.*

i. NumOclExpr $\subseteq \mathbb{Z} \uplus \mathbb{R}$

ii. *If* $e_1, e_2 \in \mathsf{NumOclExpr}$*, then* $e_1 \; op_{Num} \; e_2 \in \mathsf{NumOclExpr}$. □

In the definition above, the set NumOclExpr is a subset of the disjoint union of the set of integers $\mathbb{Z}$ and the set of reals $\mathbb{R}$. This construction allows us to distinguish integer values from real values in the set of numerical OCL expressions, which will be necessary once we define the typing function for NumOclExpr (see Def. 8). Further, numerical OCL expressions may be combined by the standard set of arithmetic operators to form complex expressions.

Definition 6 (OCL expressions) *Let* $\Sigma_{\mathbb{M}} = (\mathbb{T}, \mathbb{O}, \mathbb{F}, \mathbb{V})$ *be the* OCL *signature over model* $\mathbb{M}$ *as described above. Then the set* OclExpr *of* OCL *expressions over* $\Sigma_{\mathbb{M}}$ *is inductively defined as follows.*

i. $\mathsf{undef} \in \mathrm{p}$ *(undefined).*

ii. $\mathsf{self} \in \mathsf{OclExpr}$ *(self reference).*

iii. *If* $e \in \mathsf{NumOclExpr}$*, then* $e \in \mathsf{OclExpr}$ *(numerical expression).*

iv. *If* $e \in \mathsf{OclExpr}$*,* $f \in \mathbb{F}$*, then* $\mathsf{nav}(e, f) \in \mathsf{OclExpr}$ *(navigation).*

v. *If* $v \in \mathbb{V}, e_1, e_2 \in \mathsf{OclExpr}$ *then* $(\mathsf{let}\ v = e_1\ \mathsf{in}\ e_2) \in \mathsf{OclExpr}$ *(let-expression).*

vi. *If* $op \in \mathbb{O}$ *with arity* n*,* $e_1, \ldots, e_n \in \mathsf{OclExpr}$*, then* $op(e_1, \ldots, e_n) \in \mathsf{OclExpr}$ *(operation call).*

vii. *If* $e_1, e_2, e_3 \in \mathsf{OclExpr}$ *then* $(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\ \mathsf{endif}) \in \mathsf{OclExpr}$ *(if-then-else).*

viii. *If* $v_1, v_2 \in \mathbb{V}$*,* $e_1, e_2, e_3 \in \mathsf{OclExpr}$*, then* $(e_1 \rightarrow \mathsf{iterate}(v_1;\ v_2 = e_2 \mid e_3)) \in \mathsf{OclExpr}$ *(iterate expression).* □

In the above definition, we omit *type casts* and *type membership tests* because we use the language in a statically typed manner only. Note, moreover, that parenthesis around OCL expressions may be omitted as long as the resulting expression is unambiguous.

Finally, we define the set BoolOclExpr of Boolean OCL expressions, which are a subset of OclExpr, just like the set NumOclExpr of numeric OCL constraints.

Definition 7 (Boolean OCL expressions) *Let* $\Sigma_{\mathbb{M}} = (\mathbb{T}, \mathbb{O}, \mathbb{F}, \mathbb{V})$ *be the* OCL *signature over model* $\mathbb{M}$*, and let* NumOclExpr *be the numerical expressions over* $\Sigma_{\mathbb{M}}$ *as defined above. Then, the set* $\mathsf{BoolOclExpr} \subseteq \mathsf{OclExpr}$ *over* $\Sigma_{\mathbb{M}}$ *is inductively defined as follows.*

i. $\mathsf{false} \in \mathsf{BoolOclExpr}$.

ii. *If* $e \in \mathsf{BoolOclExpr}$*, then* $\neg e \in \mathsf{BoolOclExpr}$.

iii. *If* $e_1, e_2 \in \mathsf{BoolOclExpr}$*, then* $e_1 \wedge e_2 \in \mathsf{BoolOclExpr}$ *where* $\wedge \in \mathbb{O}$.

iv. *If* $e_1, e_2 \in \mathsf{NumOclExpr}$*, then* $e_1 \; op_{Rel} \; e_2 \in \mathsf{BoolOclExpr}$*,* $op_{Rel} \in \{<, >\} \subseteq \mathbb{O}$.

v. *If* $e_1, e_2 \in \mathsf{OclExpr}$*, then* $e_1 = e_2 \in \mathsf{BoolOclExpr}$. □

The Boolean OCL expressions consist of one primitive value, namely false, negation ($\neg$), and conjunction ($\wedge$). Moreover, we define equality expressions ($=$) and strict inequality comparisons, *less than* ($<$) and *greater than* ($>$). Since negation and conjunction comprise a functionally complete set of Boolean operators, we add to the set of the Boolean OCL expressions true and $\vee$ as *syntactic sugar*, i.e., true abbreviates $\neg\mathsf{false}$ and $e_1 \vee e_2$ stands for $\neg(\neg e_1 \wedge \neg e_2)$, respectively.

In the following, we define the typing function $\mathtt{type} : \mathsf{OclExpr} \rightarrow \mathbb{T}^*$ that statically assigns a type $t \in \mathbb{T}^*$ to every OCL expression $e \in \mathsf{OclExpr}$. We denote by $\mathrm{V}_t \subseteq \mathbb{V}$ the set of variables of type $t \in \mathbb{T}$. We first define the typing function for numerical OCL expressions, then, we extend the typing function to the set $\mathsf{OclExpr}$ of OCL expressions, and finally, define the typing function for equality expressions.

Definition 8 (Typing of numerical OCL expressions) *Let $\Sigma_{\mathbb{M}} = (\mathbb{T}, \mathbb{O}, \mathbb{F}, \mathbb{V})$ be the* OCL *signature over model $\mathbb{M}$, and let $op_{Num} \in \{+, -, *, /\} \subseteq \mathbb{O}$. The typing function* $\mathtt{type} : \mathsf{NumOclExpr} \rightarrow \mathbb{T}^*$ *for numerical* OCL *expressions* $\mathsf{NumOclExpr} \subseteq \mathsf{OclExpr}$ *is defined inductively as follows.*

i. *Let $x \in \mathsf{NumOclExpr}$. If $x \in \mathbb{Z}$, then $\mathtt{type}(x) = Int$.*

ii. *Let $x \in \mathsf{NumOclExpr}$. If $x \in \mathbb{R}$, then $\mathtt{type}(x) = Real$.*

iii. $\mathtt{type}(e_1\ op_{Num}\ e_2) = Int$, *if* $\mathtt{type}(e_1) = Int, \mathtt{type}(e_2) = Int$.

iv. $\mathtt{type}(e_1\ op_{Num}\ e_2) = Real$, *if* $\mathtt{type}(e_1) = Real, \mathtt{type}(e_2) = Real$. □

Note that the typing function for numerical operations op_{Num} is only defined for numeric arguments that agree on their type, i.e., their type is undefined if called with one argument of type *Int* and the other of type *Real*. In the following definition, we provide the typing rules for OCL expressions $\mathsf{OclExpr}$, where we omitted the treatment of inheritance and dropped support for OclAny for the sake of simplicity.

Definition 9 (Typing of OCL expressions) *Let $\Sigma_{\mathbb{M}} = (\mathbb{T}, \mathbb{O}, \mathbb{F}, \mathbb{V})$ be the* OCL *signature over model $\mathbb{M} = (T, I, \mathcal{A}, root_{\mathbb{M}}, type_{\mathbb{M}})$, let $\mathrm{V}_t \subseteq \mathbb{V}$ be the set of variables of type $t \in \mathbb{T}$, and let $\alpha, \beta \in \mathbb{T}^*$ be types. The typing function* $\mathtt{type} : \mathsf{OclExpr} \rightarrow \mathbb{T}^*$ *for* OCL *expressions is defined inductively as follows.*

i. $\mathtt{type}(\mathsf{undef}) = \perp$.

ii. $\mathtt{type}(\mathsf{self}) = root_{\mathbb{M}}$.

iii. $\mathtt{type}(\mathsf{nav}(e, f)) = \beta$, *if* $\mathtt{type}(e) = \alpha$, $\mathtt{type}(f) = \alpha \rightarrow \beta$, *and* $e \in \mathsf{OclExpr}$.

iv. $\mathtt{type}(\mathsf{let}\ v = e_1\ \mathsf{in}\ e_2) = \beta$, *if* $\mathtt{type}(e_1) = \alpha$, $\mathtt{type}(e_2) = \beta$, $v \in \mathbb{V}$, *and* $e_1, e_2 \in \mathsf{OclExpr}$.

v. $\mathtt{type}(op(e_1, \ldots, e_n)) = \beta$, *if* $\mathtt{type}(op) = \alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow \beta$, $op \in \mathbb{O}$, $\mathtt{type}(e_1) = \alpha_1, \ldots, \mathtt{type}(e_n) = \alpha_n$, *and* $e_1, \ldots, e_n \in \mathsf{OclExpr}$.

vi. $\mathtt{type}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\ \mathsf{endif}) = \alpha$, *if* $\mathtt{type}(e_1) = \mathit{Bool}$, $\mathtt{type}(e_2) = \alpha$ *and* $\mathtt{type}(e_3) = \alpha$, *and* $e_1, e_2, e_3 \in \mathsf{OclExpr}$.

vii. $\mathtt{type}(e_1 \to \mathsf{iterate}(v_1; v_2 : t = e_2 \mid e_3)) = \beta$, *if* $\mathtt{type}(e_1) = \mathit{Collection}(\alpha)$, $\mathtt{type}(e_2) = \mathtt{type}(e_3) = \beta$, $t \in \mathbb{T}, t = \beta$, $v_1, v_2 \in \mathbb{V}$, *and* $e_1, e_2, e_3 \in \mathsf{OclExpr}$. □

Definition 10 (Typing of Boolean OCL expressions) *Let* $\Sigma_{\mathbb{M}} = (\mathbb{T}, \mathbb{O}, \mathbb{F}, \mathbb{V})$ *be the* OCL *signature over model* $\mathbb{M}$, *let* $\{\neg, \wedge, =\} \subseteq \mathbb{O}$, *and let* $\mathit{op}_{\mathit{Rel}} \in \{<, >\} \subseteq \mathbb{O}$. *The typing function* $\mathtt{type} : \mathsf{BoolOclExpr} \to \mathbb{T}^*$ *for Boolean* OCL *expressions* $\mathsf{BoolOclExpr} \subseteq \mathsf{OclExpr}$ *is defined inductively as follows.*

i. $\mathtt{type}(\mathsf{false}) = \mathit{Bool}$.

ii. $\mathtt{type}(\neg e) = \mathit{Bool}$, *if* $\mathtt{type}(e) = \mathit{Bool}$.

iii. $\mathtt{type}(e_1 \wedge e_2) = \mathit{Bool}$, *if* $\mathtt{type}(e_1) = \mathit{Bool}, \mathtt{type}(e_2) = \mathit{Bool}$.

iv. $\mathtt{type}(e_1\ \mathit{op}_{\mathit{Rel}}\ e_2) = \mathit{Bool}$, *if* $\mathtt{type}(e_1) = \mathit{Int}, \mathtt{type}(e_2) = \mathit{Int}$.

v. $\mathtt{type}(e_1\ \mathit{op}_{\mathit{Rel}}\ e_2) = \mathit{Bool}$, *if* $\mathtt{type}(e_1) = \mathit{Real}, \mathtt{type}(e_2) = \mathit{Real}$.

vi. $\mathtt{type}(e_1 = e_2) = \mathit{Bool}$, *if* $\mathtt{type}(e_1) = \mathtt{type}(e_2)$. □

Note again that the typing functions for expressions with inequality operations $\mathit{op}_{\mathit{Rel}}$ are only defined if the types of their arguments are the same. Finally, we informally introduce — for the sake of completeness — the types of the collection operation, get, which is defined as $\mathtt{type}(\mathsf{get}) = \mathit{Collection}(\alpha) \to \alpha \to \mathit{Collection}(\alpha)$.

An OCL expression is defined w.r.t. to a model $\mathbb{M}$ and evaluated over an instantiation of $\mathbb{M}$, referred to as an instance model M of $\mathbb{M}$. An instance model represents a *state* (also: *snapshot*) of a system and contains all currently active *objects*, the *links* among these objects, and the *values* for each attribute of an object. For the definition of the semantics of an OCL expression we fix a domain of values that an expression may evaluate to. The domain $\mathbb{D}$ consists of all objects of an instance model M, the sets of Boolean, integer, real, and string values, enumeration literals, and collection values. Recall that we denote by $M|_{V_G}$ and $M|_{E_G}$ the set V_G of vertices and the set E_G of edges of the graph G underlying model $M = (G, \mathit{type}_M, \mathit{root}_M)$, respectively. In the following, $\mathbb{Z}, \mathbb{R}, \mathbb{S}$ denote the sets of integers, reals, and strings. Note that the set $\mathbb{S}$ is the set of string literal expressions as defined in the OCL standard [148, p. 77].

Definition 11 (Domain w.r.t. an instance model) *Let* $\Sigma_{\mathbb{M}} = (\mathbb{T}, \mathbb{O}, \mathbb{F}, \mathbb{V})$ *be the* OCL *signature over model* $\mathbb{M}$ *and let* $M \in \mathbb{M}$ *be an instance model of* $\mathbb{M}$. *Moreover, let* $\mathbb{Z}, \mathbb{R}, \mathbb{S}$ *be the set of integer, real, and string values, respectively. We define the* domain $\mathbb{D}$ w.r.t. an instance model $M = (G, \mathit{type}_M, \mathit{root}_M)$ *as the union* $\mathbb{D} = \bigcup_{t \in \mathbb{T}} D_t$, *that is,*

$$\mathbb{D} = D_{V_T} \cup D_{E_T} \cup D_{\mathit{Bool}} \cup D_{\mathit{Int}} \cup D_{\mathit{Real}} \cup D_{\mathit{String}} \cup D_{\mathit{Enum}} \cup D_{\mathit{Collection}} \cup D_{\perp}$$

where

$$\begin{aligned} D_{V_T} &= M|_{V_G} & D_{E_T} &= M|_{E_G} \\ D_{Bool} &= \{true, false\} & D_{Int} &= \mathbb{Z} \\ D_{Real} &= \mathbb{R} & D_{String} &= \mathbb{S} \\ D_{Enum} &\subseteq (\mathbb{S} \times \mathbb{S} \times \mathbb{Z}) & D_{\perp} &= \perp \end{aligned}$$

and $D_{Collection} \subseteq C^*$ *where* C^* *is the smallest set* $\bigcup_{i \geqslant 0} C_i$ *with*

$$\begin{aligned} C_0 &= \bigcup_{D \in \mathbb{D} \setminus D_{\perp}} \mathcal{P}(D) \\ C_{i+1} &= \mathcal{P}(C_i). \end{aligned}$$

□

The semantics of OCL expressions may now be defined by the interpretation function

$$I[\![.]\!] : \mathsf{OclExpr} \to \mathbb{M} \to \mathbb{D}$$

that evaluates an OCL expression $e \in \mathsf{OclExpr}$ w.r.t. to a model $\sigma \in \mathbb{M}$, called a *state*, and maps it to a value in $\mathbb{D}$. In the following definition, we denote by $expr[e_1/x]$ the replacement of all occurrences of x in $expr$ by e_1.

Since a state σ is a model $M \in \mathbb{M}$, we write $\sigma|_{V_G}$ and $\sigma|_{E_G}$ to denote the vertices V_G and E_G of graph G of $M = (G, type_M, root_M)$, respectively.

Definition 12 (Semantics of OCL expressions) *Let* $\Sigma_{\mathbb{M}} = (\mathbb{T}, \mathbb{O}, \mathbb{F}, \mathbb{V})$ *be the* OCL *signature over model* $\mathbb{M} = (T, I, \mathcal{A}, root_{\mathbb{M}}, type_{\mathbb{M}})$ *with* $T = (V_T, E_T, src_T, tgt_T)$, *let* $M = (G, type_M, root_M) \in \mathbb{M}$ *be an instance model of* $\mathbb{M}$ *with graph* $G = (V_G, E_G, src_G, tgt_G)$ *and typing function* $type_M : G \to T$ *on which the* OCL *expression is evaluated, and let* $\sigma = M \in \mathcal{S}$ *be a state. Further,* $\mathsf{get} : Collection(t) \to t \to Collection(t)$ *is the function that returns a pair consisting of (i) an element from the collection, (ii) the collection resulting from the removal of the element. Note that* get *returns the elements of two equal collections* c_1, c_2 *in identical order, i.e.,* $\forall c_1, c_2 \in Collection(t), \forall n \in \mathbb{N}, n < |c_1| : c_1 = c_2 \Rightarrow \mathsf{get}^n(c_1) = \mathsf{get}^n(c_2)$. *Then, the semantics of* OCL *expressions are defined inductively as follows.*

i. $I[\![\mathsf{undef}]\!](\sigma) = \perp$.

ii. $I[\![\mathsf{self}]\!](\sigma) = root_M$.

iii. $$I[\![\mathsf{nav}(e, f)]\!](\sigma) = \begin{cases} \{tgt_G(edge) \mid edge \in \sigma|_{E_G}, \\ \qquad type_M(edge) = f, \\ \qquad src_G(edge) \in I[\![e]\!](\sigma)\} & \text{if } src_T(f) = type_M(I[\![e]\!](\sigma)), \\ \perp & \text{otherwise} \end{cases}$$

iv. $I[\![op(e_1, \ldots, e_n)]\!](\sigma) = I[\![op]\!](\sigma)(I[\![e_1]\!](\sigma), \ldots, I[\![e_n]\!](\sigma)), op \in \mathbb{O}$.

v. $I[\![\mathsf{let}\ v = e_1\ \mathsf{in}\ e_2]\!](\sigma) = I[\![e_2']\!](\sigma)$ *where* $e_2' = e_2[e_1/v]$.

vi. $I[\![\ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ \mathrm{e}_3\ \mathsf{endif}]\!](\sigma) = \begin{cases} I[\![e_2]\!](\sigma) & \textit{if } I[\![e_1]\!](\sigma) = \textit{true} \\ I[\![e_3]\!](\sigma) & \textit{if } I[\![e_1]\!](\sigma) = \textit{false} \\ \bot & \textit{otherwise.} \end{cases}$

vii. $I[\![xs \rightarrow \mathsf{iterate}(v_1; v_2 = e_2|e_3)]\!](\sigma) = I[\![xs \rightarrow \mathsf{iterate}'(I[\![e_2]\!](\sigma), e_3)]\!](\sigma)$ *where* $\mathsf{iterate}'$ *is defined as follows:*

$$I[\![xs \rightarrow \mathsf{iterate}'(acc, e_3)]\!](\sigma) = \begin{cases} acc & \textit{if } I[\![xs]\!](\sigma') = \varnothing \\ I[\![xs' \rightarrow \mathsf{iterate}'(acc', e_3)]\!](\sigma) & \textit{otherwise} \end{cases}$$

where $(x, xs') = \mathsf{get}(xs)$, $acc' = e_3[x/v_1, acc/v_2]$. □

In the above definition, $\mathsf{self} \in \mathsf{OclExpr}$ always evaluates to the root node $root_M \in V_G$ of graph G that underlies the instance model M. The evaluation of *navigation* expressions returns a set of objects, i.e., vertices $obj \in V_G$. The vertices are obtained as the target endpoints of an edge $edge \in E_G$, which is an instance of a feature edge $f \in \mathbb{F}$ whose source vertex denotes the type of an object $obj \in V_G$. The object obj is obtained as a result of evaluating expression e whose type is equal to the source of feature f. In this way, navigation can start from either a single vertex or a set of vertices all of which need to be of equal type. An operation call is evaluated to an interpreted function which is applied to the set of evaluated arguments. The evaluation of a *let-expression* first evaluates expression e_1. Then all occurrences of variable v are replaced by the thus obtained result yielding expression e_2'. The evaluation then continues with expression e_2'. The evaluation of an *if-then-else* expression tests whether e_1 is *true* or *false* and returns either the result of evaluating e_2 or e_3. Finally, the *iterate* expression takes a collection e_1, initializes the accumulator variable acc to the result of $I[\![e_2]\!](\sigma)$ and calls $\mathsf{iterate}'$. The $\mathsf{iterate}'$ expression takes three arguments, the collection e_1, the evaluation of the accumulator acc and an expression e_3. Upon execution the $\mathsf{iterate}'$ expression returns current value of the accumulator acc if the collection e_1 is empty. Otherwise, it removes the first element of e_1 and replaces all occurrences of both v_1 and v_2 with the head of e_1 and the current accumulator value, respectively, in e_3 which is subsequently evaluated and stored in the updated accumulator acc'. Then, $\mathsf{iterate}'$ is recursively called on the tail of e_1, the updated accumulator acc', and the expression e_3.

4.2 The CTL-extended Object Constraint Language

In this section, we formally introduce the syntax and semantics of the CTL-extended Object Constraint Language, cOCL for short, which enriches OCL with standard CTL [42] operators. For this purpose, we integrate our extension into the formal semantics of OCL as presented in Section 4.1.

We define the set $\mathsf{cOclExpr}$ of cOCL expressions as a superset of the OCL expressions as defined in Def. 6 and 7. In particular, the set of cOCL expressions contains all OCL expressions induced by the OCL signature $\Sigma_{\mathbb{M}}$ over a model $\mathbb{M}$ and extends the set of Boolean OCL expressions by temporal expressions resulting in the set $\mathsf{BoolcOclExpr}$ of Boolean cOCL expressions as follows.

Definition 13 (Syntax of cOCL) *Let $\Sigma_{\mathbb{M}}$ be the* OCL *signature over model $\mathbb{M}$ and let $\mathsf{OclExpr}$ be the set of* OCL *expressions induced by $\Sigma_{\mathbb{M}}$. Then, the set $\mathsf{cOclExpr}$ of cOCL expression is defined inductively as follows.*

i. If $e \in \mathsf{OclExpr}$, then $e \in \mathsf{cOclExpr}$.

ii. Let $\mathsf{BoolOclExpr} \subseteq \mathsf{OclExpr}$ be the set of Boolean OCL *expressions as defined in Def. 7. Then, let $\mathsf{BoolcOclExpr} \subseteq \mathsf{cOclExpr}$ be the smallest set of Boolean cOCL expressions such that*

- $\mathsf{BoolOclExpr} \subseteq \mathsf{BoolcOclExpr}$,
- *if $e \in \mathsf{BoolcOclExpr}$, then $\neg e, \mathsf{AX}\, e, \mathsf{EX}\, e \in \mathsf{BoolcOclExpr}$, and*
- *if $e_1, e_2 \in \mathsf{BoolcOclExpr}$, then $e_1 \wedge e_2$, $\mathsf{A}\, e_1 \,\mathsf{W}\, e_2$, $\mathsf{E}\, e_1 \,\mathsf{W}\, e_2$, $\mathsf{A}\, e_1 \,\mathsf{U}\, e_2$, $\mathsf{E}\, e_1 \,\mathsf{U}\, e_2 \in \mathsf{BoolcOclExpr}$.*

Our extension introduces three temporal operators, *next* (X), *weak until* (W), and (strong) *until* (U), which are quantified either existentially (E) or universally (A). We define two additional operators, *eventually* (F) and *globally* (G), by the following equivalences: $\mathsf{EF}\varphi \equiv \mathsf{E}\, \mathit{true} \,\mathsf{U}\, \varphi$ and $\mathsf{AF}\varphi \equiv \mathsf{A}\, \mathit{true} \,\mathsf{U}\, \varphi$, and $\mathsf{EG}\varphi \equiv \mathsf{E}\, \varphi \,\mathsf{W}\, \mathit{false}$ and $\mathsf{AG}\varphi \equiv \mathsf{A}\, \varphi \,\mathsf{W}\, \mathit{false}$. Note that *next*, *eventually*, and *globally* have a single subformula as argument, whereas the *weak until* and *until* operators have two subformulas. Before we define the semantics of the temporal operators, we formally introduce the term *transition system* which describes all possible executions of a system. A transition system is defined over a model $\mathbb{M}$ (see Def. 1) and it consists of a set of states and transitions between these states. The transitions are labeled with model transformations and a morphism that identifies the subgraph, to which the transformation is applied. A distinguished state is marked initial.

Definition 14 (Transition System) *The transition system $\mathcal{K}_{\mathbb{M}}$ of a model $\mathbb{M}$ is a tuple $(\mathcal{S}, \mathcal{R}, \mathcal{M}, \mathcal{T}, \iota)$ consisting of a finite set $\mathcal{S}$ of states, a finite set $\mathcal{R}$ of model transformations, a set $\mathcal{M}$ of morphisms with domain and co-domain $\mathbb{M}$, a transition relation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{R} \times \mathcal{M} \times \mathcal{S}$, and an initial state $\iota \in \mathcal{S}$.*

A state $\sigma \in \mathcal{S}$ is a model that conforms to $\mathbb{M}$, i.e., $\mathcal{S} \subseteq \mathbb{M}$. A transformation $p : L \xleftarrow{l} K \xrightarrow{r} R, p \in \mathcal{R}$ defines a transformation for a model $M \in \mathbb{M}$ and a morphism $m : L \to G$ defines a mapping of the LHS graph L of p into graph G of an instance model $M = (G, \mathit{type}_M, \mathit{root}_M) \in \mathbb{M}$. Because a state σ is an instance model M, we define that an action $\alpha = (p, m)$ rewrites a state $\sigma_s \in \mathcal{S}$ into a state $\sigma_t \in \mathcal{S}$ at match m and we write $\sigma_s \overset{p,m}{\Longrightarrow} \sigma_t$. For a sequence of transitions, so-called *execution traces*, we define the term *path* as follows.

Definition 15 (Path) *Let $\mathcal{K}_{\mathbb{M}} = (\mathcal{S}, \mathcal{R}, \mathcal{M}, \mathcal{T}, \iota)$ be the transition system of a model $\mathbb{M}$. A* path *π is a finite or infinite sequence of states $(\sigma_0\sigma_1\sigma_2\ldots)$ with $\sigma_i \in \mathcal{S}$ and $(\sigma_i, p, m, \sigma_{i+1}) \in \mathcal{T}$ and $\sigma_i \overset{p,m}{\Longrightarrow} \sigma_{i+1}$ for all $i \geqslant 0$. For a path $\pi = (\sigma_0\sigma_1\sigma_2\ldots)$, we define the projection function $\pi(i) = \sigma_i$. The length of a path $|\pi| = n \in \mathbb{N}$ for finite paths $\pi = (\sigma_0 \ldots \sigma_n)$, and $|\pi| = \infty$ for infinite paths $\pi = (\sigma_0\sigma_1\sigma_2\ldots)$. By $\Pi_{\mathcal{K}_{\mathbb{M}}}$ we denote the set of all paths of $\mathcal{K}_{\mathbb{M}}$.*

We are now able to define the semantics of cOCL as follows.

Definition 16 (Semantics) *Let $\mathcal{K}_{\mathbb{M}} = (\mathcal{S}, \mathcal{R}, \mathcal{M}, \mathcal{T}, \iota)$ be the transition system associated with model $\mathbb{M}$ as defined above. The semantics of a cOCL expression w.r.t. a state $\sigma \in \mathcal{S}$ is defined by the rules i.–vii. originating from Definition 12 and six additional rules viii.–xiii. for the temporal extension as follow.*

viii. $I[\![\mathsf{A}\, e_1\, \mathsf{U}\, e_2]\!](\sigma) = \{\pi \mid \pi \in \Pi_{\mathcal{K}_{\mathbb{M}}}, \pi(0) = \sigma, \nexists n \in \mathbb{N}, n \leqslant |\pi| :$
$(I[\![e_2]\!](\pi(n)) = \mathit{true} \wedge \forall\, 0 \leqslant i < n : I[\![e_1]\!](\pi(i)) = \mathit{true})\} = \varnothing$

ix. $I[\![\mathsf{E}\, e_1\, \mathsf{U}\, e_2]\!](\sigma) = \{\pi \mid \pi \in \Pi_{\mathcal{K}_{\mathbb{M}}}, \pi(0) = \sigma, \exists n \in \mathbb{N}, n \leqslant |\pi| :$
$(I[\![e_2]\!](\pi(n)) = \mathit{true} \wedge \forall\, 0 \leqslant i < n : I[\![e_1]\!](\pi(i)) = \mathit{true})\} \neq \varnothing$

x. $I[\![\mathsf{A}\, e_1\, \mathsf{W}\, e_2]\!](\sigma) = \{\pi \mid \pi \in \Pi_{\mathcal{K}_{\mathbb{M}}}, \pi(0) = \sigma, \nexists n \in \mathbb{N}, n \leqslant |\pi| :$
$(I[\![e_1]\!](\pi(n)) = \text{false} \rightarrow \exists i \in \mathbb{N}, i \leqslant n : I[\![e_2]\!](\pi(i)) = \mathit{true})\} = \varnothing$

xi. $I[\![\mathsf{E}\, \phi\, \mathsf{W}\, \psi]\!](\sigma) = \{\pi \mid \pi \in \Pi_{\mathcal{K}_{\mathbb{M}}}, \pi(0) = \sigma, \exists n \in \mathbb{N}, n \leqslant |\pi| :$
$(I[\![e_1]\!](\pi(n)) = \text{false} \rightarrow \exists i \in \mathbb{N}, i \leqslant n : I[\![e_2]\!](\pi(i)) = \mathit{true})\} \neq \varnothing$

xii. $I[\![\mathsf{E}\, \mathsf{X}\, e]\!](\sigma) = \{\pi \mid \pi \in \Pi_{\mathcal{K}_{\mathbb{M}}}, \pi(0) = \sigma \wedge |\pi| \geqslant 1 \wedge I[\![e]\!](\pi(1)) = \mathit{true}\} \neq \varnothing$

xiii. $I[\![\mathsf{A}\, \mathsf{X}\, e]\!](\sigma) = \{\pi \mid \pi \in \Pi_{\mathcal{K}_{\mathbb{M}}}, \pi(0) = \sigma \wedge |\pi| \geqslant 1 \wedge I[\![e]\!](\pi(1)) = \text{false}\} = \varnothing$

The above definitions for formulas with allquantified path operators A may be read as follows: first, search in the set $\Pi_{\mathcal{K}_{\mathbb{M}}}$ of all paths of $\mathcal{K}_{\mathbb{M}}$ for a counterexample, i.e., a path that does not satisfy the desired condition, e.g., condition for $\mathsf{A}\, e_1\, \mathsf{U}\, e_2$. If no such path exists, the formula evaluates to *true*; otherwise, the set of counterexamples is not empty and hence the formula evaluates to *false*. For example, in case of $\mathsf{A}\, e_1\, \mathsf{U}\, e_2]\!](\sigma)$ we find a counterexample if for all states $\sigma_i, i \leqslant |\pi|$ either e_2 never evaluates to *true* or there exists a state $\sigma_j, j \leqslant i$ where e_1 does not hold. In case of a formula with an existential path operator, we search for a path that satisfies the desired condition, i.e., the set of paths that satisfy the condition is not empty. Note that the semantics of the *eventually* and *globally* operators follow directly from the above definitions. We define the *satisfiability* of cOCL expressions as follows.

Definition 17 (Satisfiability) *A cOCL expression ϕ is* satisfiable *w.r.t. a transition system $\mathcal{K}_{\mathbb{M}}$ with initial state ι iff it holds in the initial state ι, i.e., $I[\![\phi]\!](\iota) = \mathit{true}$.*

In the remainder of this chapter, we discuss how the enumerative, explicit-state model checker MocOCL verifies cOCL specifications. We discuss its model checking algorithm and present its web based user interface.

natural language	**cOCL expression**
Initially, there is a field containing a treasure.	`self.fields->exists(field | field.treasure)`
The game is over/not over.	`Always Next false/Exists Next true`
The game will surely be over sometimes.	`Always Eventually (Always Next false)`
Pacman will find the treasure in all cases.	`Always Eventually self.pacman.on.treasure`
If the treasure is next to Pacman, he can always find it in the next turn.	`Always Globally self.pacman.on.neighbor-> exists(field.treasure) implies (Exists Next self.pacman.on.treasure)`
As long as not all fields next to Pacman are occupied by ghosts, there is a possibility that the game is not over after the next turn.	`Always Globally self.pacman.on.neighbor-> exists(field | self.ghosts-> forAll(g | field <> g.on) implies (Exists Next (Exists Next true)))`
As long as the game is not over, every ghost may move to at least two different positions.	`Always self.ghosts-> forAll(g | g.on.neighbor->select(field | Exists Next g.on = field)->size() >= 2) Unless (Always Next false)`

Table 4.1: Examples of cOCL expressions in the concrete syntax of MocOCL.

4.3 MocOCL — A model checker for cOCL specifications

The implementation of MocOCL consists of three parts, (i) a concrete syntax for cOCL expressions, (ii) a backend that implements the enumerative, explicit-state model checking algorithm, and (iii) a frontend realized as a web based user interface that allows the user to interact with the backend.

Concrete Syntax

The *concrete syntax* enhances the readability of cOCL expressions. It allows us to write the temporal operators in their familiar long forms, i.e., $\mathsf{X}\varphi$, $\mathsf{F}\varphi$, $\mathsf{G}\varphi$, $\varphi\,\mathsf{W}\,\psi$, and $\varphi\,\mathsf{U}\,\psi$ become **Next** φ, **Eventually** φ, **Globally** φ, φ **Unless** ψ, and φ **Until** ψ. The universal and existential path quantifiers preceding the temporal operators become **Always** and **Exists** or, alternatively, **Sometimes**. In our implementation, we extend the Xtext[2] grammar of the concrete syntax of OCL, which provides us with a Java API and an editor that performs syntax highlighting for cOCL expressions. Table 4.1 shows examples of MocOCL expressions in the concrete syntax. The first expression in this table is an ordinary OCL expression without any temporal operators; it is thus evaluated only in the current state. Note that the `exits` keyword does not refer to the temporal (path) operator but to the set operator of OCL. In comparison, the existential path operator **Exists** is capitalized—just like all the other temporal operators that the concrete syntax of cOCL introduces. The expressions `Always Next false` and `Exists Next true` can be used to test for termination and non-termination, respectively. Here, `Always Next false` evaluates to `false` whenever there exists a next state along some path π, because `false` evaluates to *false* in every expression. But it evaluates to *true* whenever there exists no next state. This is due to the evaluation of a $\mathsf{AX}\varphi$ expression that checks, among others, whether all path π starting in the current state are of length greater than zero. If this condition is not satisfied the entire expression evaluates to *true* regardless of the evaluation of φ. Hence, the expression `Always Next false` is suitable to test for a terminating path. In contrast, the expression `Exists Next true` evaluates to *true* if there exists a path π (starting in the current state) whose length is greater than zero. Expression 5–7 showcase how cOCL expressions can be nested to form more complicated properties. In particular, expression 5 and 6 specify invariants that have to be satisfied in every state, while expression 7 specifies an *Always-Until* formula, i.e., $\mathsf{A}\,\varphi_1\,\mathsf{W}\,\varphi_2$, that demands that either φ_1 holds invariantly unless the game ends (φ_2).

Backend

The backend consists of a parser for the textual concrete syntax of cOCL and the model checker MocOCL that verifies cOCL specifications. Details on the implementation, which is based on the Eclipse Modeling Framework [190], are presented in the following.

The prototypical, EMF based[3] implementation of the MocOCL model checker performs the actual verification task as follows. Given an Ecore-conformant model, an instance model that represents the system's initial state, a set of model transformations, and a cOCL specification, MocOCL generates the state space, verifies the cOCL specification, and finally reports to the modeler information on the reason of the verification result. Although the actual implementation of our model checker generates the state space iteratively and verifies the specification on-the-fly the explanation of the imple-

[2] `http://www.eclipse.org/Xtext/`

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

mentation in this section is more formal and omits these optimizations in favor of a comprehensible presentation.

In MocOCL, the state space consists of a set of instance models that conform to some model $\mathbb{M}$. Each instance model is represented internally as an attribute, typed graph with inheritance and composition edges (see Sec. 4.1. Consequently, each state in the state space is a model $M = (G, \mathit{type}_M, \mathit{root}_M)$ with graph $G = (V_G, E_G, \mathit{src}_G, \mathit{tgt}_G)$. Given a set $\mathcal{R}$ of graph transformations, a set $\mathcal{M}$ of morphisms, and a state $\sigma \in \mathbb{M}$, we use the successor function $\textbf{succ} : \mathcal{S} \to \mathcal{P}(\mathcal{S})$ to perform a step-wise exploration of the state space. The successor function returns the set of successor states reachable from state σ and is defined as $\textbf{succ}(\sigma_s) := \bigcup_{p \in \mathcal{R}} \{\sigma_t \mid \sigma_s \overset{p,m}{\Longrightarrow} \sigma_t, m \in \mathcal{M}\}$. The set of states $\mathcal{S}$ can now be defined as the closure of the successor function applied to the designated initial state ι; it is defined as the smallest set $\mathcal{S} = \bigcup_{i>0} \mathcal{S}_i$ where

$$\begin{aligned} \mathcal{S}_0 &= \iota \\ \mathcal{S}_{i+1} &= \iota \cup \{\textbf{succ}(\sigma) \mid \sigma \in \mathcal{S}_i\}. \end{aligned}$$

The transition relation $\mathcal{T}$ is then defined as the set $\{(\sigma_s, p, m, \sigma_t) \mid \sigma_s \overset{p,m}{\Longrightarrow} \sigma_t, p \in \mathcal{R}, m \in \mathcal{M}\}$. Thus, the set $\mathcal{R}$ of graph transformations together with the successor function applied to the initial state ι give rise to a transition system $\mathcal{K}_{\mathbb{M}} = (\mathcal{S}, \mathcal{R}, \mathcal{M}, \mathcal{T}, \iota)$. We evaluate a cOCL expression on the paths induced by this transition system.

The algorithm for *evaluating* cOCL expressions of the form $(\mathsf{A}|\mathsf{E})\, \phi\, (\mathsf{U}|\mathsf{W})\, \psi$ is shown in Figure 4.2. To ease the presentation we drop intermediate checks that allow the algorithm to stop the evaluation once enough states have been explored to conclude whether the formula holds. The algorithm assigns each state to one of three sets. If $I[\![\varphi]\!](\sigma)$ evaluates to *true*, then φ is assigned to $\mathit{\Phi}$. Similarly, if $I[\![\psi]\!](\sigma)$ holds, then state σ is assigned to Ψ. If a state σ is reachable from a φ-state, i.e., a state where φ holds, but neither φ nor ψ hold in σ, then state σ is assigned to η. All states that have not been processed yet are stored in the worklist ω. The algorithm proceeds as follows. The algorithm initializes the worklist with the start state σ_{start} and uses the **succ** function to iteratively expand the set of reachable states. It evaluates φ and ψ in each state σ and assigns σ to the corresponding sets $\mathit{\Phi}$ or Ψ, or to η if neither φ or ψ hold. Once every reachable state is assigned to either $\mathit{\Phi}$, Ψ, or η, the algorithm constructs the set Δ, which contains all states from $\mathit{\Phi}$ whose successor states lie on a path that leads to a state outside of $\mathit{\Phi}$. That is, Δ contains all states that are not part of a circular sequence of φ-states. Then, $I[\![\mathsf{A}\, \phi\, \mathsf{U} \psi]\!](\sigma)$ holds if η is empty, and $\mathit{\Phi}$ contains neither cycle nor deadlock; $I[\![\mathsf{E}\, \phi\, \mathsf{U} \psi]\!](\sigma)$ holds if Ψ is not empty; $I[\![\mathsf{A}\, \phi\, \mathsf{W} \psi]\!](\sigma)$ holds if η is empty; and $I[\![\mathsf{E}\, \phi\, \mathsf{W} \psi]\!](\sigma)$ holds if Ψ is not empty or $\mathit{\Phi}$ contains a cycle. Expressions $(\mathsf{A}|\mathsf{E})\, \mathsf{X} \phi$ are implemented as $I[\![(\mathsf{A}|\mathsf{E})\, \mathsf{X} \phi]\!](\sigma) := (\forall|\exists) \tau \in \mathrm{succ}(\sigma) : I[\![\phi]\!](\tau) = \mathit{true}$, where we check if all (at least one) successor of the current state satisfies φ.

The evaluation of a cOCL expression yields a *report* that, besides returning the result of the evaluation, contains a *cause* or explanation for the result. A cause is associated with a cOCL expression. It stores the result of the evaluation of the associated expression and, for each relevant sub-expression, a sub-cause. A sub-expression is *relevant* if it influences the result of its super-expression. For example, if the sub-expression φ in

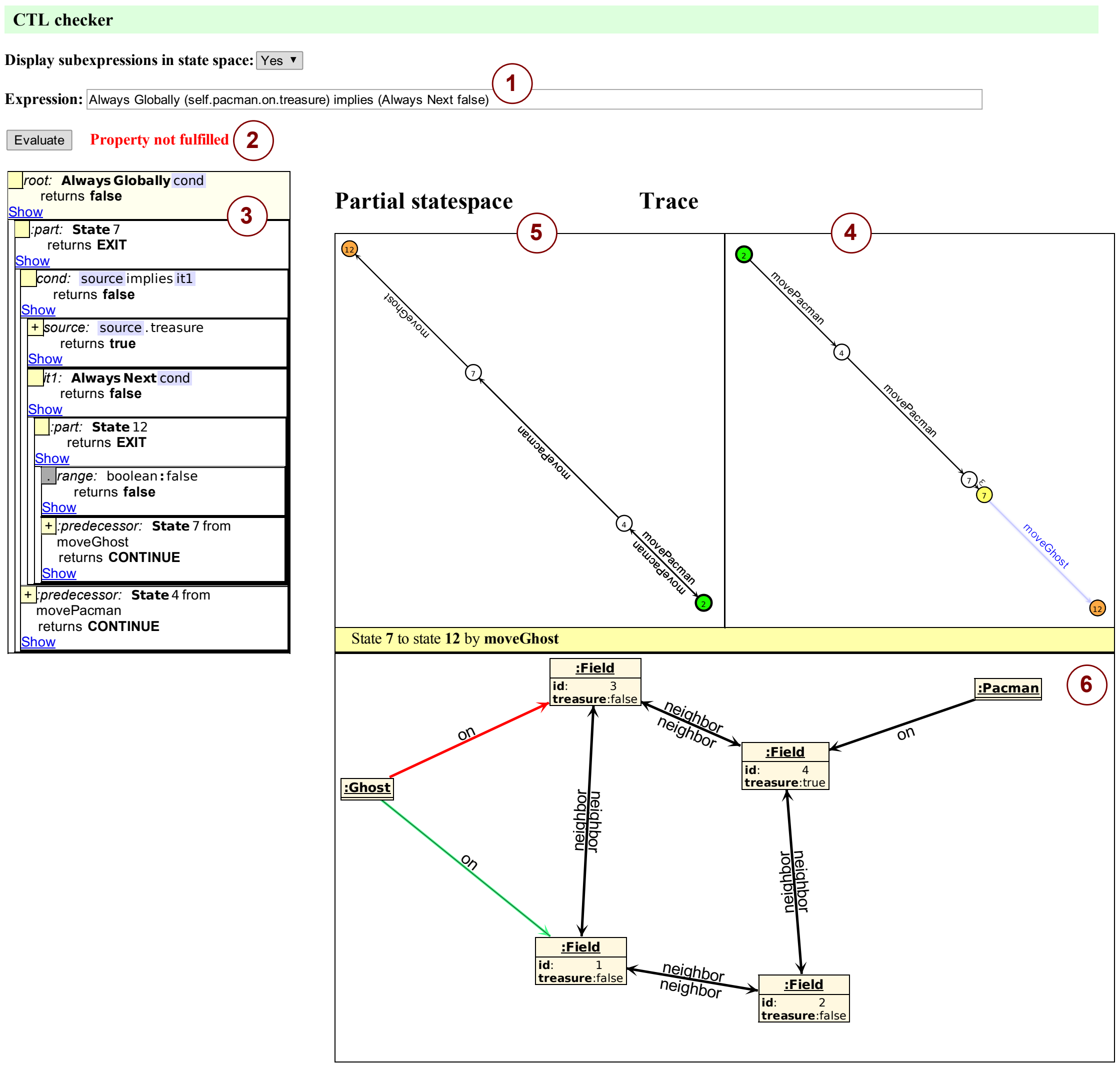
```
**function** evaluate($\sigma_{start}$, $(P\ \phi\ T\ \psi)$) : Bool
  **Description:** Evaluates the cOCL expression $(P\ \phi\ T\ \psi)$ in state $\sigma_{start}$.
  **Parameters:** $\sigma_{start}$: the start state; $P$: Path operator, *Always* or *Exists*; $T$: Temporal operator, *Until* or *Unless*;
  **Returns**: *true* iff $(P\ \phi\ T\ \psi)$ holds in $\sigma_{start}$
$\omega := \{\sigma_{start}\}$;
$\varPhi := \emptyset$;
$\Psi := \emptyset$;
$\eta := \emptyset$;
**while** $\omega \neq \emptyset$
    **pick** $\sigma \in \omega$;
    $\omega := \omega \backslash \{\sigma\}$;
    **if** $I[\![\phi]\!](\sigma)$ **then**
        $\varPhi := \varPhi \cup \{\sigma\}$;
        $\omega := \omega \cup \mathrm{succ}(\sigma) \backslash (\varPhi \cup \Psi \cup \eta)$;
    **else if** $I[\![\psi]\!](\sigma)$ **then**
        $\Psi := \Psi \cup \{\sigma\}$;
    **else**
        $\eta := \eta \cup \{\sigma\}$
    **end if**
**end while**
$\Delta := \emptyset$;
$\Delta_l := \emptyset$;
**repeat**
    $\Delta_l := \Delta$;
    $\Delta := \{\sigma \in \varPhi \mid \mathrm{succ}(\sigma) \neq \emptyset$ **and**
                        $\mathrm{succ}(\sigma) \cap (\varPhi \backslash \Delta_l) = \emptyset\}$
**until** $\Delta = \Delta_l$

**switch** $(P, T)$
  **case** (*Always*, *Until*):
    **return** $\varPhi = \Delta$ **and** $\eta = \emptyset$;
  **case** (*Always*, *Unless*):
    **return** $\eta = \emptyset$
  **case** (*Exists*, *Until*):
    **return** $\varPhi \neq \emptyset$;
  **case** (*Exists*, *Unless*):
    **return** $\varPhi \neq \emptyset$ **or** $\varPhi \neq \Delta$;
**end switch**
```

Figure 4.2: Until/Unless Algorithm Pseudo Code.

φ **or** ψ evaluates to **true** then no sub-cause is generated for ψ as the evaluation of φ uniquely determines the result of φ **or** ψ. If, however, both φ and ψ evaluate to **false**, then a sub-cause for each of the two sub-expressions is generated and stored in the cause of φ **or** ψ. Note that the cause generation is not necessarily deterministic, as is the case, for example, if both φ and ψ evaluate to **true** in φ **or** ψ.

Frontend

The MocOCL frontend provides (i) an interface for the parametrization of the verification engine, i.e., the backend described in the previous section, and (ii) visualization and report generation support that provides useful information for the modeler on the reason of a specific verification result. The MocOCL user interface consists of the following parts: (1) an input field for the cOCL specification, (2) the result of the verification, i.e., whether the cOCL specification is satisfied or not, (3) the cause that textually describes (4) the trace of the evaluation, which is embedded in (5) the partial state space. Further, upon clicking on a state or transition from (3) the cause, (4) the trace, or (5) the partial state space, the selected state or transition is visualized in (6) the object diagram pane. The changes caused by a transition are highlighted in red and green indicating



CTL checker

Display subexpressions in state space: Yes

Expression: Always Globally (self.pacman.on.treasure) implies (Always Next false)

Evaluate

Property not fulfilled

```
root:  Always Globally cond
   returns false
Show
 :part:  State 7
    returns EXIT
 Show
  cond:  source implies it1
     returns false
  Show
   + source:  source.treasure
        returns true
   Show
   it1:  Always Next cond
        returns false
   Show
    :part:  State 12
       returns EXIT
    Show
     range:  boolean : false
        returns false
     Show
     + :predecessor:  State 7 from
       moveGhost
       returns CONTINUE
     Show
  + :predecessor:  State 4 from
    movePacman
    returns CONTINUE
  Show
```

Partial statespace

Trace

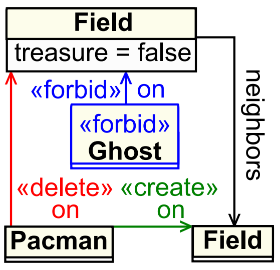
State 7 to state 12 by moveGhost

Figure 4.3: Visualization of a cause in the MocOCL tool

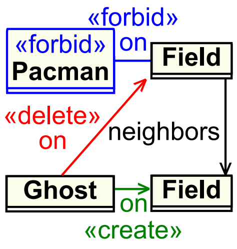
the deletion and creation of an association, respectively.

4.4 MocOCL in Action

In this section we implement the Pacman game as described in the beginning of this chapter. In our initial implementation we deliberately introduced a bug for the purpose of demonstrating how MocOCL can be used to debug an erroneous implementation.



(a) *Move Pacman* rule



(b) *Move Ghost* rule

Figure 4.4: Implementation of the behavior of the Pacman game.

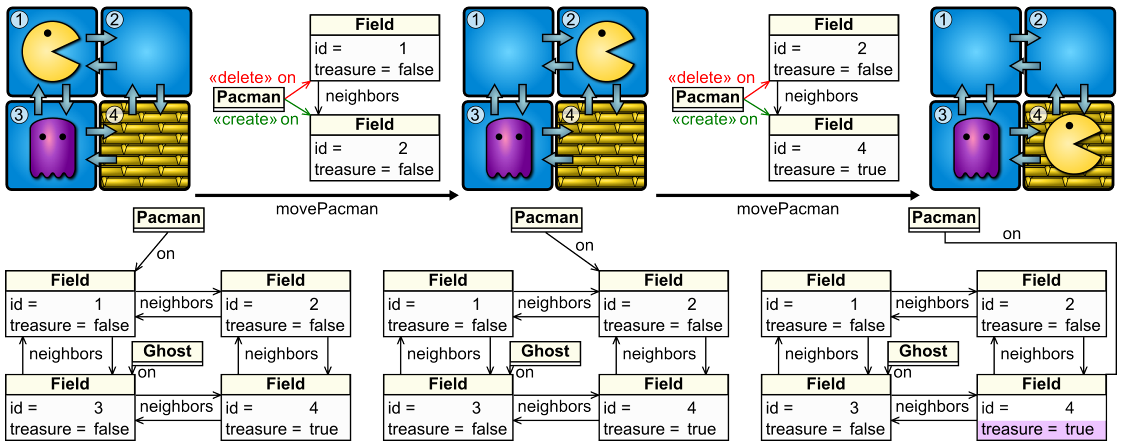


Figure 4.5: Example for Transformations.

Implementation. We outlined the static structure of our implementation of the Pacman game depicted in Fig. 4.1a in the beginning of this chapter already. To implement the described behavior we define two graph transformations. The first transformation, *Move Pacman*, is depicted in Fig. 4.4a and describes a single move of Pacman. The second transformation, *Move Ghost* (see Fig. 4.4b), describes a single move of a ghost. Pacman and the ghosts are only allowed to move if the game is not over yet, that is, a player must not move if Pacman is on a treasure field or if one of the ghosts caught Pacman. Note, however, that the first restriction is not enforced by the *Move Ghost* rule; hence, ghosts may still move if Pacman already found the treasure.

In Figure 4.5 we exemplarily illustrate two applications of the *Move Pacman* graph transformation and the subsequent changes to the current state. First, Pacman moves from Field 1 to Field 2 and in the next round Pacman moves from Field 2 to Field 4, which contains the treasure. In this scenario, Pacman wins the game. The lower part of Fig. 4.5 displays the states resulting from applying the *Move Pacman* rule to the current state, which is shown leftmost, and we witness that the `on` reference of `Pacman` is updated accordingly, while the `Ghost` remains stationary.

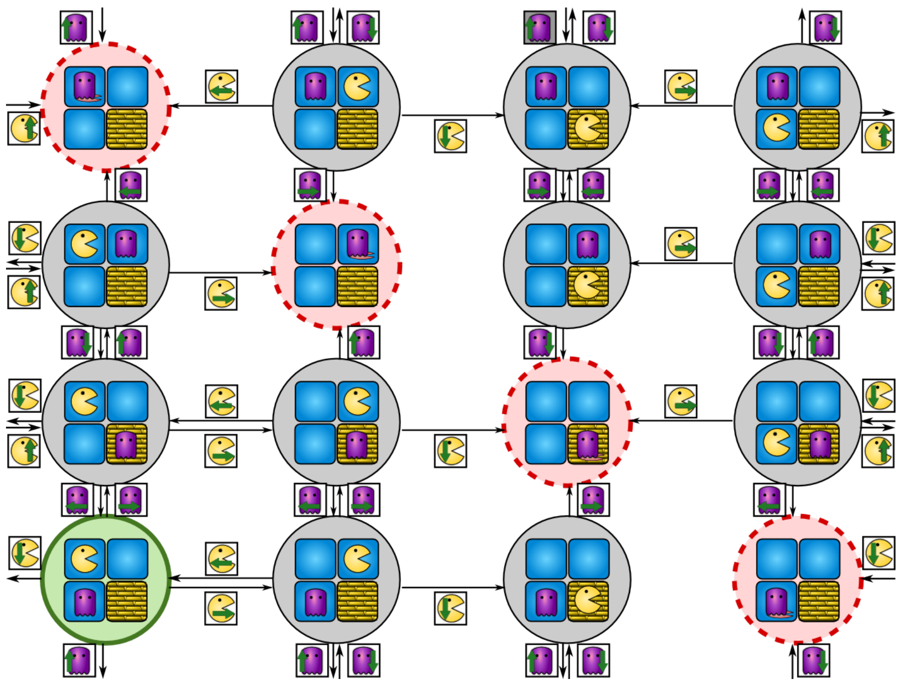


Figure 4.6: State Space of the Pacman Game.

Verification Task. To verify whether the game stops whenever Pacman wins or looses we formulate two cOCL expressions. The first expression, `Always Globally (self.pacman.on.treasure) implies (Always Next false)`, states that the game terminates if Pacman moves onto a treasure field, while the second expression, `Always Globally ghosts->exists(on=pacman.on) implies (Always Next false)`, checks if the game terminates whenever there exists a ghost who resides on the same field as Pacman. We may now supply MocOCL with the static structure of the Pacman game (Fig.4.1a), an initial state (e.g. Fig. 4.1b), the behavior of the players as graph transformations (see Fig. 4.4), and one of the two cOCL specifications to initiate the verification. MocOCL then constructs the state space of the Pacman game iteratively. In our implementation, we use the graph transformation tool Henshin [6] that explores the state space by recursively applying all matching graph transformation rules to the user-provided initial model. The full state space resulting from recursively applying the rules *Move Pacman* and *Move Ghost* to the initial model (Fig. 4.1b) is depicted in Figure 4.6. The *initial state* in the bottom-left corner of the figure is highlighted in green with a bold border and the end states are marked red with a dashed border. The transitions between the states show possible moves of Pacman and the ghost. Overall there are $4 * 4 = 16$ different states (the ghost has to be placed on each field and Pacman has to be placed on each field).

Figure 4.3 depicts a screenshot of MocOCL that displays the verification result for the first cOCL expression, `Always Globally (self.pacman.on.treasure) implies`

`(Always Next false)`. Note that out of the 16 states only four needed to be explored to conclude that the expression does not hold, i.e., the game does not end if Pacman finds a treasure. The cause shows a scenario where Pacman finds the treasure in two moves starting from the initial state (state 2) and moving first to state 4 and then to state 7. There exists, however, a transition `moveGhost` leading from state 7 to state 12. If we select this transition in (4) the trace pane, it is highlighted in blue. The changes associated with the transition are then displayed in (6) the object diagram pane. The deletion and creation of the `on` relation between the ghost and two adjacent fields describes the ghost's move. Consequently, the ghost may perform moves after Pacman already resides on the treasure field, in this case, field 4. Thus, the implementation does not satisfy the specification of the game and needs to be fixed by introducing an additional *Negative Application Condition* for the *Move Ghost* rule such that a ghost may no longer move once Pacman found the treasure. A corrected version of the *Move Ghost* transformation is shown in Section 6.1 in Fig. 6.2c.

A demo version of MocOCL is available as a browser version at

`http://www.modelevolution.org/mococl/`

and can be used without any installation efforts. In the demo version the initial model is fixed to the 2×2 board shown in Fig. 4.1b due to memory limitations on the server. A browser based version for custom installations, which is not restricted to the Pacman model, is available for download at

`http://www.modelevolution.org/prototypes/mococl`.

4.5 Summary

In this chapter we presented a novel approach to verify rich, temporal specifications directly at the level of models. For this purpose we extend the Object Constraint Language (OCL) by temporal operators from the Computation Tree Logic (CTL). This extensions, called cOCL, allows us to formulate temporal OCL specification and we presented its formal syntax and semantics. Moreover, we described the implementation of a state-of-the-art model checker, called MocOCL, that verifies a system w.r.t. its cOCL specification. The static structure of the system is hereby defined as Ecore model, which is the *de facto* reference implementation of the MOF standard developed within the Eclipse Modeling Framework (EMF). The behavior of the system is defined by algebraic graph transformations. The state space of the system is computed by applying the set of graph transformation to a given graph that represents the initial state of the system and subsequently to all graphs resulting from these applications of the graph transformations. MocOCL performs the state space exploration iteratively and evaluates the cOCL specification on-the-fly. It comes with a web based graphical user interface that reports the results of a verification run back to the user.

In the next chapter, we will present and elaborate on a complementary approach that uses a symbolic representation of the state space based on the relational calculus [95,194] that we developed to improve the performance of the verification of safety properties. Finally, in chapter 6 we compare both of these approaches in an evaluation consisting of four benchmarks.

CHAPTER 5

Symbolic Model Checking of Safety Properties with Gryphon

In this chapter we present our symbolic model checker GRYPHON that verifies safety properties of the form $\mathsf{AG}\,\neg\phi$, where ϕ is denotes a bad state, or reachability properties of the form $\mathsf{EF}\,\phi$, where ϕ denotes a good state. In contrast to model checkers like MocOCL, that store states explicitly in memory, a symbolic model checker represents states and transitions *symbolically* by means of (propositional) formulas that allow an efficient representation of large state spaces.

Overview. Similar to MocOCL, our symbolic model checker GRYPHON expects as input an EMF (meta-)model $\mathbb{M}$, a set $\mathcal{R}$ of graph transformations, an initial state M_ι, and a set Φ of safety properties. These safety properties are expressed by means of *graph constraints*, which can be thought of as degenerate graph transformations with identical left-hand and right-hand sides. Thus, graph constraints may only match a (sub-)graph but do not change it; hence, we use graph constraints to describe desired and undesired states of a system under verification. Moreover, GRYPHON requires a map $\Gamma : \mathbb{M}|_{V_T} \rightarrow \mathbb{N}$ that bounds the maximal number of objects per class; this is necessary to obtain a finite representation of the system. For the purpose of verifying a set of safety constraints, the EMF (meta-)model $\mathbb{M}$, the set $\mathcal{R}$ of graph transformations, the initial state M_ι, the graph constraints Φ, and the upper bound map Γ are translated into a *sequential circuit.* The steps required to translate the inputs into a sequential circuit are depicted in Figure 5.1. The translation starts by converting $\mathbb{M}$, $\mathcal{R}$, and Φ into a problem of first-order, relational logic consisting of a finite universe bounded by Γ, a set *Rel* of relations derived from elements in $\mathbb{M}$, and relational formulas that encode the behavior of the system. These formulas are subsequently translated into Boolean functions and, extracted therefrom, we obtain a sequential circuit, whose initial state is defined by M_ι. We represent the generated sequential circuit as an *And-Inverter Graph* (AIG) that can be compactly stored in the AIGER format [19,22]. The AIGER format is

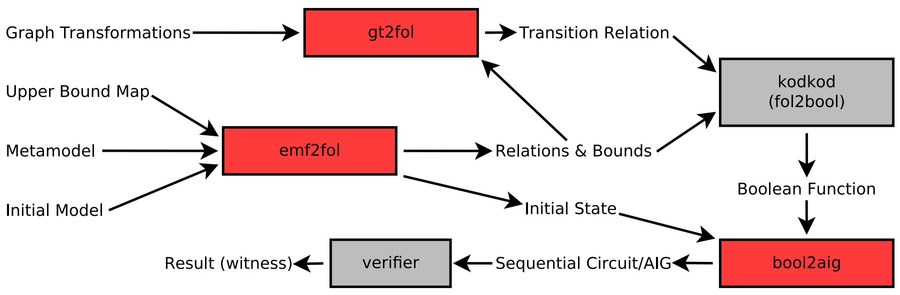


Figure 5.1: GRYPHON workflow. Labeled arrows represent data flows between the components of GRYPHON. Those components that are highlighted in red are contributions of this thesis.

the standardized input format used for the Hardware Model Checking Competition [21] and is thus supported by many well-known model checkers. By using a standardized representation we decouple our encoding from the verification procedure and can thus apply any of the existing, highly-specialized model checking algorithms.

Chapter Structure. We again start with a section on preliminaries where we introduce bounded, first-order relational logic and sequential circuits. We then discuss the translation of models and graph transformations into sequential circuits in detail; the workflow of this translation is depicted in Figure 5.1. Finally, we briefly discuss the implementation of GRYPHON and outline future extensions and optimizations.

5.1 Preliminaries

In the following we provide a summary of the concepts employed to perform the translation from EMF models and graph transformations to and-inverter graphs (AIG). That is, we explain syntax and semantics of bounded, first-order relational logic, and sequential circuits and their representation as and-inverter graphs in the AIGER format as used in the Hardware Model Checking Competition [21]. Note that this section builds in parts on the preliminaries presented in Section 4.1 that are not repeated here. In particular, we will make use of Definitions 1 and 2 that define the notion of (meta-)models and instance models, denoted by $\mathbb{M}$ and M, respectively.

First-Order Relational Logic

Relational logic[1] is an extension of propositional logic that introduces *relations* and relational *variables* of a given arity, a *universe* of atoms (or objects), and *quantifiers*. If the universe is *finite*, the relational logic is said to be *bounded*. In the following, we will focus our treatment on bounded, first-order relational logic. In this restricted

[1] Our presentation of the logic follows the one presented for ALLOY [95] and KODKOD [198] that in turn are based on Tarski's exposition of the relational calculus [194].

problem := univ relation* formula

univ := {obj[, obj]*}
relation := $\text{rel}_{:\text{arity}}$[lower[, upper]?]

varDecl := var : expr
lower := constant
upper := constant
constant := {tuple*}
tuple := $\langle$obj[, obj]*$\rangle$

arity := $\mathbb{N}$
obj := ID
rel := ID
var := ID

expr := rel | var | unary | binary | comprehension
unary := expr unop
unop := $^{+}$ | $^{-1}$
binary := expr binop expr
binop := $\cup$ | $\cap$ | $\setminus$ | . | $\times$
comprehension := {varDecl | formula}

formula := atomic | composite | quantified
atomic := expr $\subseteq$ expr
composite := $\neg$formula | formula logop formula
logop := $\wedge$ | $\vee$
quantified := quantifier varDecl | formula
quantifier := $\forall$ | $\exists$

(a) Syntax

P: problem $\to$ binding $\to$ boolean
R: relation $\to$ binding $\to$ boolean
M: formula $\to$ binding $\to$ boolean
E: expr $\to$ binding $\to$ constant
binding: (var $\cup$ rel) $\to$ constant

$\text{P}[\![\ U\ \ r_1 \ldots r_n\ \ \text{F}\ ]\!]\text{b} = \text{R}[\![r_1]\!]\text{b} \wedge \cdots \wedge \text{R}[\![r_n]\!]\text{b} \wedge \text{M}[\![\text{F}]\!]\text{b}$

$\text{R}[\![r\ :\ [\text{L}]]\!]\text{b} = \text{R}[\![r\ :\ [\text{L}, \text{L}]]\!]\text{b}$
$\text{R}[\![r\ :\ [\text{L}, \text{U}]]\!]\text{b} = \text{L} \subseteq \text{b}(r) \subseteq \text{U}$

$\text{M}[\![\text{p} \subseteq \text{q}]\!]\text{b} = \text{E}[\![\text{p}]\!]\text{b} \subseteq \text{E}[\![\text{q}]\!]\text{b}$
$\text{M}[\![\neg\text{F}]\!]\text{b} = \neg\text{M}[\![\text{F}]\!]\text{b}$
$\text{M}[\![\text{F}\ op\ \text{G}]\!]\text{b} = \text{M}[\![\text{F}]\!]\text{b}\ op\ \text{M}[\![\text{G}]\!]\text{b}$ where $op = \{\wedge, \vee\}$
$\text{M}[\![\forall\, v : \text{p} \mid \text{F}]\!]\text{b} = \bigwedge_{x \in \text{E}[\![\text{p}]\!]\text{b}} \text{M}[\![\text{F}]\!](\text{b} \oplus [v \mapsto x])$
$\text{M}[\![\exists\, v : \text{p} \mid \text{F}]\!]\text{b} = \bigvee_{x \in \text{E}[\![\text{p}]\!]\text{b}} \text{M}[\![\text{F}]\!](\text{b} \oplus [v \mapsto x])$

$\text{E}[\![\text{p}\ op\ \text{q}]\!]\text{b} = \text{E}[\![\text{p}]\!]\text{b}\ op\ \text{E}[\![\text{q}]\!]\text{b}$ where $op = \{\cup, \cap, \setminus\}$
$\text{E}[\![\text{p} \,.\, \text{q}]\!]\text{b} = \{\langle \text{p}_1, \ldots, \text{p}_{n-1}, \text{q}_2, \ldots, \text{q}_m \rangle \mid$
$\quad \langle \text{p}_1, \ldots, \text{p}_n \rangle \in \text{E}[\![\text{p}]\!]\text{b} \wedge \langle \text{q}_1, \ldots, \text{q}_m \rangle \in \text{E}[\![\text{q}]\!]\text{b}$
$\quad \wedge \text{p}_n = \text{q}_1\}$
$\text{E}[\![\text{p} \times \text{q}]\!]\text{b} = \{\langle \text{p}_1, \ldots, \text{p}_n, \text{q}_1, \ldots, \text{q}_n \rangle \mid$
$\quad \langle \text{p}_1, \ldots, \text{p}_n \rangle \in \text{E}[\![\text{p}]\!]\text{b} \wedge \langle \text{q}_1, \ldots, \text{q}_n \rangle \in \text{E}[\![\text{q}]\!]\text{b}\}$
$\text{E}[\![\text{p}^{-1}]\!]\text{b} = \{\langle \text{p}_2, \text{p}_1 \rangle \mid \langle \text{p}_1, \text{p}_2 \rangle \in \text{E}[\![\text{p}]\!]\text{b}\}$
$\text{E}[\![\text{p}^{+}]\!]\text{b} = \{\langle x, y \rangle \mid \exists \text{p}_1, \ldots, \text{p}_n |$
$\quad \langle x, \text{p}_1 \rangle, \langle \text{p}_1, \text{p}_2 \rangle, \ldots, \langle \text{p}_n, y \rangle \in \text{E}[\![\text{p}]\!]\text{b}\}$
$\text{E}[\![\{v : \text{p} \mid \text{F}\}]\!]\text{b} = \{x \in \text{E}[\![\text{p}]\!]\text{b} \mid \text{M}[\![\text{F}]\!](\text{b} \oplus [v \mapsto x])\}$
$\text{E}[\![r]\!]\text{b} = \text{b}(r)$
$\text{E}[\![x]\!]\text{b} = \text{b}(x)$

(b) Semantics

Figure 5.2: Syntax and semantics of relational logic [198]

logic, each relation is assigned an upper bound and, optionally, a lower bound. Syntax and semantics of this bounded, first-order relational logic is provided in Figure 5.2 and follows closely the presentation of [196]. A relational problem P in bounded, first-order relational logic is a tuple $(U, Rel, \Phi, \bigsqcap, \bigsqcup, ar)$ that consists of

(i) a *universe* of discourse U, which is a *sequence* of uninterpreted *atoms*,

(ii) a set Rel of relations,

(iii) a relational formula Φ,

(iv) a map $ar : Rel \to \mathbb{N}$ that assigns an arity to each relation $r \in Rel$, and

(v) maps $\bigsqcap : Rel \to \mathcal{P}(U^n)$ and $\bigsqcup : Rel \to \mathcal{P}(U^n)$ that define lower and upper bounds for relations. Lower and upper bounds of a relation $r \in Rel$ are defined as tuples over the set of atoms from the universe, with n being the arity $ar(r)$ of relation r.

We further define a relational *constant* as a *set* of tuples of a given arity including the empty set. The set of relational *expressions* is recursively defined as the smallest set consisting of the empty set and the set of all atoms, i.e., the universe U, the relations $r \in$

Rel, and all expressions resulting from applying either (i) a unary operator like *transitive closure* ($^+$) or *transposition* ($^{-1}$) to another expression or (ii) a binary operator, *union* ($\cup$), *intersection* ($\cap$), *join* (.), *difference* ($\setminus$), or *product* ($\times$), to the former and another expression. The evaluation of an expression yields a relational constant. An atomic *relational formula* is a sentence constructed over two relational *expressions* connected by the subset $\subseteq$ operator. Formulas can be quantified and composed into composite formulas using the usual logical connectives, *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). A *solution*, or *model*, of a relational problem is an assignment, i.e., a *binding*, of tuples to relations such that (1) the assigned tuples lie within the lower and upper bounds of the relations such that (2) the formula evaluates to true. Note that our treatment of relational logic is untyped; admissible bindings to relations are solely defined by their lower and their upper bounds. Further note that we treat relations and relational variables as mentioned in Figure 5.2 as *varDecls* alike because relational variables are bound from above by a relational expression, which evaluates to a tuple, upon declaration. That is, relational variables can be thought of as relations with an empty lower bound.

The logic also supports reasoning over integer values. The range of supported integers can be conveniently specified by providing a *bitwidth*, i.e., the upper bound on the number of bits used to represent integers in the universe. Using, for example, a two's complement binary number representation and N bits, the numbers in the range -2^{N-1} to $2^{N-1}-1$ can be represented in the universe. In this case, the bounds for the predefined Int relation, which represents integer values, are defined as

$$\sqcap(\mathsf{Int}) = \sqcup(\mathsf{Int}) = \{\langle -2^{N-1}\rangle, \ldots, \langle 2^{N-1}-1\rangle\}.$$

Example. In the following we formulate a simple relational problem to determine whether every woman's husband has the woman as his wife and vice versa. We define two binary relations, $\mathsf{Husband}_{:2}$ and $\mathsf{Wife}_{:2}$, and the universe U of discourse is populated with atoms $\{adam, eve, john, marry, larry\}$. The upper bounds are set as follows,

$$\begin{aligned}\sqcup(\mathsf{Wife}) = \{&\langle adam, eve\rangle, \langle adam, marry\rangle, \langle larry, eve\rangle,\\ &\langle larry, marry\rangle, \langle john, eve\rangle, \langle john, marry\rangle\}\end{aligned}$$

and

$$\begin{aligned}\sqcup(\mathsf{Husband}) = \{&\langle eve, adam\rangle, \langle eve, larry\rangle, \langle eve, john\rangle,\\ &\langle marry, adam\rangle, \langle marry, larry\rangle, \langle marry, john\rangle\}.\end{aligned}$$

In addition we set the lower bound on Husband to $\sqcap(\mathsf{Husband}) = \{\langle eve, adam\rangle\}$, that is, a solution to the relational problem is required to contain at least the tuple $\langle eve, adam\rangle$ in the Husband relation. The requirement that "*X is Y's husband if and only if Y is X's wife.*" may now be written in relational logic as shown in Eq. 5.1.

$$\Phi := \mathsf{Wife} = \mathsf{Husband}^{-1} \tag{5.1}$$

Here, we essential define that the Husband be symmetric to Wife. Alternatively, we may use the predefined binary identity relation id, which contains tuples $\langle adam, adam\rangle$,

$\langle eve, eve\rangle, \langle john, john\rangle, \ldots$, i.e., for each atom a in the universe a tuple $\langle a, a\rangle$, and first-order quantifiers to formulate the requirement as

$$\forall \mathsf{x} \in \mathsf{Husband}, \exists \mathsf{y} \in \mathsf{Wife} \mid \mathsf{x.y} \subseteq \mathsf{id} \wedge$$
$$\forall \mathsf{y} \in \mathsf{Wife}, \exists \mathsf{x} \in \mathsf{Husband} \mid \mathsf{y.x} \subseteq \mathsf{id}$$

Note that we use *equality* ($=$) in formula $\mathsf{x} = \mathsf{y}$ as a short-hand notation for $\mathsf{x} \subseteq \mathsf{y} \wedge \mathsf{y} \subseteq \mathsf{x}$ and, similarly, *implication* ($\Rightarrow$) in formula $\mathsf{x} \Rightarrow \mathsf{y}$ as a short-hand notation for $\neg\mathsf{x} \vee \mathsf{y}$.

The relational problem $\mathrm{P}[\![\ U \ \ \mathsf{Husband}_{:2}, \mathsf{Wife}_{:2} \ \ \Phi \ ]\!]\mathrm{b}$ evaluates to *true* under binding $\mathrm{b} = \{\mathsf{Wife} = \{\langle adam, eve\rangle\}, \mathsf{Husband} = \{\langle eve, adam\rangle\}\}$ such that $\mathrm{E}[\![\mathsf{Husband}^{-1}]\!]\mathrm{b}$ evaluates to $\{\mathsf{adam}_{:2} eve\}$ which is equal to the binding of the Wife relation. Thus the formula $\mathsf{Wife} = \mathsf{Husband}^{-1}$ evaluates to *true*, i.e., $formeval \mathsf{Wife} = \mathsf{Husband}^{-1} = true$, and formula Φ is satisfied. Note that other bindings may also satisfy this formula. For example, binding $\mathrm{b}_1 = \{\mathsf{Wife} = \{\langle adam, eve\rangle, \langle adam, marry\rangle\}, \mathsf{Husband} = \{\langle eve, adam\rangle, \langle marry, adam\rangle\}\}$, although legally prohibited, satisfies Φ, too, as we did specify that a marriage may only bind exactly two people. To forbid such models additional constraints are necessary.

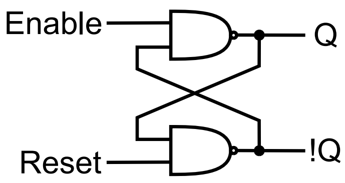
Kodkod

KODKOD provides a sophisticated API and model finder to formulate and solve formulas of bounded first-order logic. In KODKOD, a relational problem consists of a *universe*, a set of *relational variable declarations*, and a *formula*. It translates such a relational problem into Boolean functions represented in Conjunctive Normal Form (CNF), which can be solved by off-the-shelf SAT solvers like MINISAT or LINGELING. For this purpose, KODKOD provides an interface to SAT solvers that feeds the Boolean circuit in conjunction normal form to the desired SAT solver. The SAT solver then searches for a satisfying assignment of truth values to the variables of the CNF formula. If it fails to find such an assignment and KODKOD was configured to request a proof of unsatisfiability, KODKOD maps the minimal unsatisfiable subset of the clauses in the CNF formula back to the corresponding fragments of the relational formula. In this way, contradicting elements of a specification may be identified. If the solver, however, finds a satisfying assignment, KODKOD produces an *instance* that assigns to the declared relational variable proper bounds, i.e., relational constants, such that the (relational) formula is satisfied.

The KODKOD API was developed to overcome the deficiencies of its predecessor, the ALLOY analyzer [197]. In particular, KODKOD now supports the specification of partial solutions that had to be encoded directly into the relational formulas. Moreover, relations in KODKOD are untyped and require explicit bounds in form of relational constants, whose tuples are drawn from the explicitly defined universe of discourse. In contrast, ALLOY only demands integer bounds on the maximum number of instances per *signature*, and these upper bounds implicitly define the universe. While ALLOY provides libraries to support Boolean values, integers, lists, *etc.*, KODKOD only supports signed, fixed-width integers, which are represented in two's complement notation. Similar to other relational variables, integers need to be bounded as well, that is, a maximum bit-width needs to

```
aag 7 2 1 2 4
2              input 0      'enable'
4              input 1      'reset'
6 8            latch 0      Q next(Q)
6              output 0     Q
7              output 1     !Q
8 4 10         AND gate 0   reset & (enable ^ Q)
10 13 15       AND gate 1   enable ^ Q
12 2 6         AND gate 2   enable & Q
14 3 7         AND gate 3   !enable & !Q
```

(a) AIGER file (taken from [19])



(b) Logic diagram

Figure 5.3: A flip-flop in AIGER format and its logic diagram. Note that the explanations to the right of the circuit elements are not part of the AIGER file.

be provided. The reasoning over integers is than bit-precise, and operations on integers, like addition, multiplication, and subtraction, are translated into their corresponding binary circuit operations, implemented as, e.g., adders and binary multipliers. Clearly, the use of integers in KODKOD impose penalties on solving time quickly as the generated Boolean formulas grow larger with increasing bit-widths.

Combinational Circuits, Sequential Circuits, and the AIGER Format

Combinational circuits consist of a set of primitive logical gates having one or more inputs and at least one output. The outputs of a combinational circuit depend solely on the current input values, that is, the gates define a direct mapping from the input values to the output values of the circuit. A Boolean function is a special case of a combinational circuit with only a single output. In contrast to *memoryless* combinational circuits, *sequential circuits* are *stateful* as their output may depend on previous input values that reflect the state of the circuit. In the following we will use the simplest form of memory to represent the state of a circuit, namely *latches*. Latches store a single bit, 0 or 1, *true* or *false*. Usually, the initial state of each latch is set to 0. Each latch is associated with a Boolean function that determines the next state of the latch as a function of the current inputs and the current state of the circuit. In a *synchronous* sequential circuit a clock sends out *signals* at discrete times that trigger a state change.

The AIGER format provides a standardized textual and binary representation for combinational and sequential circuits, of which only the textual, ASCII format will be discussed in the following. The AIGER format is published as a draft in version 1.9.4 [22]

```
(
    { '0' | '1' | '2' } <newline>          # status
  ( { 'b' | 'c' } <index> )+ <newline>      # properties
    { '0' | '1' | 'x' }* <newline>          # initial state
  ( { '0' | '1' | 'x' }* <newline> )+       # input vector(s)
  '.'
)*
```

Figure 5.4: Witness format

and available from `http://fmv.jku.at/hwmcc11/beyond1.pdf`. The AIGER format represents circuits compactly by And-Inverter Graphs (AIG), that is, circuits are formed as acyclic, directed graphs whose nodes are either NOT or AND gates with one or two inputs, respectively, and a single output. The format defines four mandatory and four optional sections. The mandatory sections encompass the inputs, the latches, one or more outputs, and a set of AND gates. Figure 5.3 displays an AIGER file of a simple flip-flop and its schematic circuit layout. An AIGER file in ASCII format starts with a header `aag M I L O A` where `M` defines the number of variables in the circuit, `I` defines the number of inputs, `L` the number of latches, `O` the number of outputs, and `A` the number of AND gates with $\mathtt{M} = \mathtt{I} + \mathtt{L} + \mathtt{A}$. The mandatory sections may be extended by four optional sections, which are declared by the quadruple `B C J F` in the header with `B` defining the number of *bad state* properties and `C`, `J`, and `F` defining the number of invariant constraints, justice, and fairness properties, respectively [22]. The header is followed by the literals that identify the circuit's inputs. For each input a variable is allocated. A variable is converted into a literal by multiplying it by two which is written into the AIGER file; an even literal identifies a positive variable, while an odd literal represents a negated variable. Here, literal 0 represents `false` and 1 represents `true`. The inputs are followed by latch definitions that consist of a triple `lit nxt ini` where `lit` identifies a latch's current state, `nxt` references the root gate of a latch's next state function, and `ini` defines a latch's initial state, either 0, 1, or undefined if left blank. The latch definitions are followed by the outputs, if any, which reference the gate whose output define an output of the circuit. If any of the optional sections are used, they follow right after the outputs. The definitions of the AND gates appears last. An AND gate is defined by the triple `LHS RHS1 RHS2` where `LHS` is the (even) literal that identifies the gate, and `RHS1` and `RHS2` are the inputs of the AND gate. Valid inputs to an AND gate are (i) an input declared in the input section, (ii) a latch declared in the latch section, (iii) another AND gate, (iv) 0 or 1.

In Gryphon we use *bad state* properties in the formulation of the safety specification. A bad state property $b \equiv \neg\varphi$ is essentially obtained by negating the *good* property φ of $\mathsf{AG}\,\phi$. If a bad state is reachable, the AIGER format defines the result of the verification as *satisfied* and requires the solver to return a *witness*. A witness describes a trace from the initial state to the bad state as a sequence of latch and input valuations. The format of a witness is reproduced from [22] in Figure 5.4. A *status* of `1` or `0` indicates satisfiability or unsatisfiability, while `2` indicates *unknown*. If the status is `1` a witness has been found for property identified in the next line of the output; in case a witness has been found for a bad property the line contains a `b` followed by the index of the property. The following lines then describe the trace to the bad state, starting with the valuation of the latches followed the valuations of the inputs. Here, an `x` indicates a "don't care" value.

5.2 Translating EMF Models and Graph Transformations to Sequential Circuits

A model based implementation of the system under verification, which consists of

- an EMF (meta-)model $\mathbb{M}$ that describes the static structure of the system,
- an instance model M_ι that defines the initial state of the system,
- a set $\mathcal{R}$ of graph transformations that describes the system's behavior,
- the specification Φ of graph constraints, and
- a set Γ of upper bounds that define the maximal number of objects per class

is translated into a sequential circuit in three steps. First, the provided inputs are translated into bounded, first-order relational logic: the upper bounds define the elements of the universe; classes, references, and attributes of model $\mathbb{M}$ are translated into relations whose bounds are derived from Γ; and the graph transformations and the graph constraints are translated into relational formulas. Viewed logically, a graph transformation is an if-then-else expression, where the nodes of the LHS correspond to existentially quantified variables: "*If* there exists a match of the LHS in host graph G, *then* rewrite G to match the RHS *else* perform no changes." These if-then-else expressions are then translated into a set of Boolean functions of the form

$$\begin{aligned} f_1(a_1, b_1, \ldots, x_1, \ldots, x_n, x'_1, \ldots, x'_n) &\wedge \\ f_2(a_2, b_2, \ldots, x_1, \ldots, x_n, x'_1, \ldots, x'_n) &\wedge \\ &\vdots \\ f_m(a_m, b_m, \ldots, x_1, \ldots, x_n, x'_1, \ldots, x'_n)&, \end{aligned}$$

where each function $f_1, \ldots, f_m$ corresponds to one graph transformation $p \in \mathcal{R}$ and the Boolean variables $x_1, \ldots, x_n$ and $x'_1, \ldots, x'_n$ reflect the current and the next state of the system, respectively. These current and next state variables can be mapped back to the bounds of a relation, which in turn can be mapped back to the objects, references, and attributes of a class. Further, the Boolean variables $a, b, \ldots$ are allocated to represent the nodes in the LHS, the RHS, the PACs, and the NACs of the graph transformation. In the final step, *next state* functions $\hat{f}_1, \ldots, \hat{f}_m$ are extracted from functions $f_1, \ldots, f_m$. In contrast to functions $f_1, \ldots, f_m$, the next state functions receive the current state

$(x_1, \ldots, x_n)$ and the node variables $a, b, \ldots$ for rule $p \in \mathcal{R}$ as their input and produce the next state $(x'_1, \ldots, x'_n)$ as their output:

$$\begin{aligned}
(x'_1, \ldots, x'_n) &= \hat{f}_1(x_1, \ldots, x_n), \\
(x'_1, \ldots, x'_n) &= \hat{f}_2(x_1, \ldots, x_n), \\
&\vdots \\
(x'_1, \ldots, x'_n) &= \hat{f}_m(x_1, \ldots, x_n).
\end{aligned}$$

From functions $\hat{f}_1, \ldots, \hat{f}_m$ we build the sequential circuit, that is, the pairs of current and next state variables correspond to the latches of the circuit and the union of all node variables $a_i, b_i, \ldots, i > 0$ corresponds to the input of the circuit. Finally, we store the constructed sequential circuit in the AIGER format for further processing through the external model checker. Depending on the result of the verification the witness is translated back to a sequence of relational bindings that correspond to a sequence of graphs that starts with the initial state M_ι and ends at the desired or undesired final state.

Each step of this translation is explained in detail in the following exposition, for which we informally define that a class C is an element of the set of type nodes in (meta-)model $\mathbb{M}$, i.e., $C \in \mathbb{M}|_{V_T}$, and the notion of the *object space Obj*, i.e., the set of all objects, as the set of all pairs $(C, i) \in \mathbb{M}|_{V_T} \times \mathbb{N}$. Moreover, we define $\Gamma : \mathbb{M}|_{V_T} \cup \mathbb{Z} \rightarrow \mathbb{N}$ as the upper bound map that defines for each class $C \in \mathbb{M}|_{V_T}$ the maximal number of objects and for the set of integers the maximal bitwidth.

The following notation will be used throughout the section; upper case letters C, D refer to classes and C, D denote (corresponding) relations; enumerations are denoted by E and E; relational variables range over $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$ and are sometimes numbered, e.g., $\mathsf{a1}, \mathsf{a2}$, to indicate that they are bound by the same (relational) expression.

From Models and Graph Transformations to Relational Logic

The translation of models and graph transformation into relational logic is performed in two steps. First, from a consistent EMF (meta-)model $\mathbb{M}$ a set *Rel* of relations is generated, which includes the relation Int of integers. Further, the universe U is derived from the upper bound map Γ, and for each relation $r \in \mathit{Rel}$ generated from a class $C \in \mathbb{M}|_{V_T}$ the upper bound $\bigsqcup(r)$ is deduced from the upper bound map Γ and, likewise, the range of integer values that bound relation Int is calculated from the bitwidth as given by $\Gamma(\mathbb{Z})$. Moreover, for each relation $r \in \mathit{Rel}$ with arity n, a function $\iota : \mathit{Rel} \rightarrow U^n$ is derived from the initial state M_ι. This function maps a relation to a set of n-ary tuples that define the relation's lower bounds in the initial state. In the second step, the symbolic transition relation is derived from the set $\mathcal{R}$ of graph transformation. The transition relation is a conjunction of relational formulas of the form $\mathit{LHS} \Rightarrow \mathit{RHS}$ where LHS and RHS are relational formulas that resemble that matching conditions and the effects of a graph transformation.

Relations, Universe, and Bounds. The translation generates for each class C in $\mathbb{M}|_{V_T}$ a unary relation C and likewise for each enumeration E in $\mathbb{M}|_{\mathcal{E}}$ a unary relation E. For each attribute *attr* in C the translation generates a binary relation $\mathsf{C\_attr}$ and, for each reference *ref* of (source) class C to some (target) class D, it creates a binary relation $\mathsf{C\_ref}$.

The universe consists of a sequence of uninterpreted atoms and its content, which is derived from the upper bound map Γ, is defined as $U = \bigcup_{C \in \mathbb{M}|_{V_T}} \bigcup_{0<i\leqslant\Gamma(C)}(C,i)$, that is, the universe consists of a number of objects for each class from the object space Obj; hence, $U \subseteq Obj$. Moreover, if bitwidth N returned by $\Gamma(\mathbb{Z})$ is greater than zero, a range $\langle -2^{N-1}\rangle, \ldots, \langle 2^{N-1}-1\rangle$ of integer representative is added to the universe given that a two's complement binary number representation is used, which we assume in the following.

Next, upper bounds for each relation are derived. Again, the number of objects per class are extracted from the upper bound map Γ; these define the upper bounds $\bigsqcup(\mathsf{C}) = \{(C,i) \mid 0 < i \leqslant \Gamma(C)\}$ of a relation $\mathsf{C}_{:1}$ generated for class $C \in \mathbb{M}|_{V_T}$. The relation Int representing integers is bound by the bitwidth $N = \Gamma(\mathbb{Z})$ and is assigned tuples ranging from $\langle -2^{N-1}\rangle$ to $\langle 2^{N-1}-1\rangle$. Boolean values and enumeration literals are mapped to integers as well. The relation Bool is bound by $\{\langle -1\rangle, \langle 0\rangle\}$ that represents *true* and *false*, respectively. An enumeration relation $\mathsf{E}_{:1}$ that was generated for an enumeration $\varepsilon = (e,l,i) \in (\mathbb{S} \times \mathbb{S} \times \mathbb{Z}) = \mathcal{E}$, where e denotes the name of the enumeration and l and i range over the key-value pairs of the enumeration, is bound by the (integer) values of its keys, i.e., $\bigsqcap(\varepsilon) = \bigsqcup(\varepsilon) = \{\langle i\rangle \mid (e,l,i) \in (\mathbb{S} \times \mathbb{S} \times \mathbb{Z})\}$. The upper bound of an attribute relation $\mathsf{C\_attr}$ is constructed from the product of the upper bounds of the class C that contains the attribute and the domain of the attribute, i.e., $\bigsqcup(\mathsf{C\_attr}) = \bigsqcup(\mathsf{C}) \times \bigsqcup(\mathbb{D})$ where domain $\mathbb{D} = \{\mathsf{Int}, \mathsf{Bool}, \mathsf{E}\}$ with E being an enumeration relation. Likewise, the upper bound of a reference relation is constructed from the product of the source and the target class's upper bounds, i.e., $\bigsqcup(\mathsf{C\_ref}) = \bigsqcup(\mathsf{C}) \times \bigsqcup(\mathsf{D})$ with source class C and target class D. Finally, we implement the inheritance hierarchy at the level of the relation's bounds. For this purpose denote by $S(C)$ the set of subclasses of class C. Then, the upper bound of C is defined as the union of all upper bounds of its subclasses, i.e., $\bigsqcup(C) = \bigcup_{s \in S(C)} \bigsqcup(s)$. Since lower bounds provide partial solutions for a relational problem, we omit their declaration for all relations except for the integer relation Int, the Boolean relation Bool, which are not expected to change.

Finally, the initial state map ι is constructed from the initial model M_ι and returns

(a) for every unary relation in *Rel* that was generated for a class C, all objects (C,i) in the initial model or the empty set if no such object is present;

(b) for every binary relation that was generated for a reference between classes C and D, all tuples $\langle (C,i),(D,j)\rangle$ if there exists a reference between objects (C,i) and (D,j) with $i,j > 0$; and

(c) for every binary relation that was generated for an attribute *attr* of class C, all tuples $\langle (C,i), \mathrm{val}(attr)\rangle$ if *attr* is set to the value returned by $\mathrm{val}(attr)$.

Class	Relation	Upper Bound
Game	$\mathsf{Game}_{:1}$	$\{\langle \mathit{Game1}\rangle\}$
pacman	$\mathsf{Game\_pacman}_{:2}$	$\{\langle \mathit{Game1}, \mathit{Pacman1}\rangle\}$
ghosts	$\mathsf{Game\_ghosts}_{:2}$	$\{\langle \mathit{Game1}, \mathit{Ghost1}\rangle\}$
fields	$\mathsf{Game\_fields}_{:2}$	$\{\langle \mathit{Game1}, \mathit{Field1}\rangle, \ldots, \langle \mathit{Game1}, \mathit{Field4}\rangle\}$
Pacman	$\mathsf{Pacman}_{:1}$	$\{\langle \mathit{Pacman1}\rangle\}$
on	$\mathsf{Pacman\_on}_{:2}$	$\{\langle \mathit{Pacman1}, \mathit{Field1}\rangle, \ldots, \langle \mathit{Pacman1}, \mathit{Field4}\rangle\}$
Ghost	$\mathsf{Ghost}_{:1}$	$\{\langle \mathit{Ghost1}\rangle\}$
on	$\mathsf{Ghost\_on}_{:2}$	$\{\langle \mathit{Ghost1}, \mathit{Field1}\rangle, \ldots, \langle \mathit{Ghost1}, \mathit{Field4}\rangle\}$
Field	$\mathsf{Field}_{:1}$	$\{\langle \mathit{Field1}\rangle, \ldots, \langle \mathit{Field4}\rangle\}$
neighbors	$\mathsf{Field\_neighbors}_{:2}$	$\{\langle \mathit{Field1}, \mathit{Field1}\rangle, \ldots, \langle \mathit{Field2}, \mathit{Field3}\rangle, \ldots, \langle \mathit{Field3}, \mathit{Field2}\rangle, \ldots, \langle \mathit{Field4}, \mathit{Field4}\rangle\}$
treasure	$\mathsf{Field\_treasure}_{:2}$	$\{\langle \mathit{Field1}, \mathit{-1}\rangle, \langle \mathit{Field1}, \mathit{0}\rangle, \ldots, \langle \mathit{Field4}, \mathit{-1}\rangle, \langle \mathit{Field4}, \mathit{0}\rangle\}$

Table 5.1: Generated relations and bounds for the Pacman game

Example. The translation of the Pacman game, introduced in Section 6.1, proceeds as follows. For the classes in the model of the Pacman game, i.e., *Game*, *Pacman*, *Ghost*, and *Field*, unary relations Game, Pacman, Ghost, and Field are generated. For each attribute and reference of these classes binary relation are generated, e.g., for the *neighbors* reference of the *Field* class a binary relation Field_neighbors is generated. Table 5.1 lists all generated relations together with their upper bounds, whose derivation is outlined in the following.

Given an upper bound map $\Gamma = \{(\mathit{Game}, 1), (\mathit{Pacman}, 1), (\mathit{Ghost}, 1), (\mathit{Field}, 4)\}$ we derive first the universe $U = \{\mathit{Game1}, \mathit{Pacman1}, \mathit{Ghost1}, \mathit{Field1}, \ldots, \mathit{Field4}\}$. Next, upper bounds for each of the above generated relations are derived accordingly. For example, for the Field relation we derive upper bounds $\bigsqcup(\mathsf{Field}) = \{\langle \mathit{Field1}\rangle, \ldots, \langle \mathit{Field4}\rangle\}$. The upper bound of the Field_neighbors relation is defined as

$$\begin{aligned}\bigsqcup(\mathsf{Field\_neighbors}) =& \bigsqcup(\mathsf{Field}) \times \bigsqcup(\mathsf{Field})\\ =&\{\langle \mathit{Field1}, \mathit{Field1}\rangle, \langle \mathit{Field1}, \mathit{Field2}\rangle, \ldots, \langle \mathit{Field2}, \mathit{Field4}\rangle, \ldots,\\ &\langle \mathit{Field3}, \mathit{Field2}\rangle, \ldots, \langle \mathit{Field4}, \mathit{Field4}\rangle\}.\end{aligned}$$

The initial state map for the Pacman game, which is depicted in Figure 4.1b, is defined by the graph

$$\begin{aligned}\iota = \{&(\mathsf{Game}, \{\langle \mathit{Game1}\rangle\}), (\mathsf{Pacman}, \{\langle \mathit{Pacman1}\rangle\}), (\mathsf{Ghost}, \{\langle \mathit{Ghost1}\rangle\}),\\ &(\mathsf{Field}, \{\langle \mathit{Field1}\rangle, \ldots, \langle \mathit{Field4}\rangle\}), (\mathsf{Game\_pacman}, \{\langle \mathit{Game1}, \mathit{Pacman1}\rangle\}),\\ &(\mathsf{Pacman\_on}, \{\langle \mathit{Pacman1}, \mathit{Field2}\rangle\}), (\mathsf{Ghost\_on}, \{\langle \mathit{Ghost1}, \mathit{Field1}\rangle\}),\\ &(\mathsf{Field\_treasure}, \{\langle \mathit{Field4}, \mathit{-1}\rangle\}), \ldots\}\end{aligned}$$

Graph Transformations. Graph transformations describe conditional modifications on graphs, that is, if the LHS and all of its application conditions are satisfied, the RHS deletes and/or creates nodes, edges, and attributes. As we use graphs to model states of our system under verification, graph transformations describe how the system transitions from the current to the next state. It is thus necessary to capture these conditional modifications and express them over relations when translating graph transformations into relational logic. In essence, we translate graph modifying instructions into relation modifying *formulas*. In contrast to graph modifying instructions that update the structure of the graph in-place, relations and relational variables are *immutable*. For example, observe that the relational formula $\exists\, \mathsf{a} : \mathsf{A} \mid \mathsf{a.n} = 1 \Rightarrow \mathsf{A} = \mathsf{A} - \mathsf{a}$ yields a contradiction under every binding where $\mathsf{a.n} = 1$ evaluates to true. Thus, for each relation $r \in \mathit{Rel}$ we introduce a primed relation, $r' \in \mathit{Rel}'$, that reflects the next state of the system after the relation modifying actions have been performed. Then, the relational formula above becomes $\exists\, \mathsf{a} : \mathsf{A} \mid \mathsf{a.n} = 1 \Rightarrow \mathsf{A}' = \mathsf{A} - \mathsf{a}$, where the upper bound of relation A' is set to the upper bound of relation A.

Example. The *MovePacman* transformation (see Fig. 4.4a) alters Pacman's *on* reference, which is represented by the $\mathsf{Pacman\_on}$ relation. The transformation rewrites the *on* reference to point to a field other than the one Pacman currently resides on. Provided that the $\mathsf{Pacman\_on}$ relation is currently bound to the tuple $\langle \mathit{Pacman1}, \mathit{Field2} \rangle$ and *Field3* is a neighboring field, the effect of applying the *MovePacman* transformation can be described by removing the tuple $\langle \mathit{Pacman1}, \mathit{Field2} \rangle$ from the $\mathsf{Pacman\_on}$ relation and adding the updated tuple $\langle \mathit{Pacman1}, \mathit{Field3} \rangle$, i.e., $\mathsf{Pacman\_on}' = \mathsf{Pacman\_on} - \{\langle \mathit{Pacman1}, \mathit{Field2} \rangle\} + \{\langle \mathit{Pacman1}, \mathit{Field3} \rangle\}$.

Formally, we translate a graph transformation into a first-order, relational formula as follows. Given sets *Rel* and *Rel'* of current and next state relations generated for a (meta-)model $\mathbb{M}$ as described above, from each (double pushout) graph transformation $p : \mathit{Cond} \leftarrow \mathit{Lhs} \rightarrow \mathit{Rhs}$, where the application condition *Cond* can consist of positive application conditions (*Pac*) and/or negative application conditions (*Nac*), a formula

$$\mathrm{F}_p := \mathit{Pre}(\mathit{Lhs}, \mathit{Pac}, \mathit{Nac}, \mathit{Rhs}) \implies \mathit{Post}(\mathit{Lhs}, \mathit{Rhs}) \tag{5.2}$$

is derived where $\mathit{Pre} : G \times G \times G \times G \rightarrow \mathbb{F}$ is a function that generates from a quadruple of graphs a conjunction of relational formulas $f \in \mathbb{F}$ that mimic the match conditions of the transformation's LHS. Function $\mathit{Post} : G \times G \rightarrow \mathbb{F}$, on the other hand, generates a conjunction of relational formulas from a pair of graphs, i.e., the LHS and RHS, that mimic the effects of the transformation's RHS. Here, the set G of graphs is typed by $T \in \mathbb{M}$.

Function *Pre* generates the following conjuncts from the transformation's LHS. For each node obj_C of class C in the LHS we allocate a fresh, existentially quantified node variable c whose domain is bound by relation $\mathsf{C}_{:1}$ that was generated for class C. This yields the relational formula $\exists\, \mathsf{c} : \mathsf{C}$. If obj_C of class C has a reference *ref* to a target object obj_D of D and under the assumption that (relational) node variables c, d have been allocated for obj_C and obj_D, and relation $\mathsf{C\_ref}_{:2}$ was generated for reference *ref*, then the condition $(\mathsf{c}{\rightarrow}\mathsf{d}) \subseteq \mathsf{C\_ref}$ is generated where $(\mathsf{c}{\rightarrow}\mathsf{d})$ denotes the product of the tuples bound to c and d.

If an attribute *attr* of obj_C is assigned an expression e, then *Pre* generates formula $(\mathsf{c}{\rightarrow}expr(e)) \in \mathsf{C\_attr}$ where function $expr : \mathbb{Z} \cup \mathbb{B} \cup (\mathbb{S} \times \mathbb{S} \times \mathbb{N}) \rightarrow \mathrm{Rexpr}$ converts an integer, Boolean, or enumeration expression into a relational expression. This expression describes an additional constraint that a matching subgraph must satisfy. Thus, we generate a condition that requires the attribute value of the object that is bound to c to evaluate to the same value as $expr(e)$. For an overview of the conditions generated by *Pre* see Table 5.2.

If the graph transformation contains PAC patterns, they are translated into formulas of relational logic like the LHS pattern because they, too, demand the existence of nodes, edges, or matching attribute expressions. Thus, the translation of LHS and PAC patterns follow the same procedure as described above.

Negative application conditions, in contrast to LHS and PAC patterns, describe forbidden patterns that must not be satisfied by any matching subgraph. As such we generate equivalent relational formulas as for the LHS, but negate them such that the formula $\neg\exists\, \mathsf{n} : \mathsf{N}$ is generated assuming that node variable n was allocated for a NAC object obj_N. Note that none of the conditions generated for references and attributes in the NAC graph need to be negated and are thus equivalent to those generated for the LHS graph.

Finally, the injectivity and the dangling edge conditions, generated by *Pre*, are necessary to faithfully translate the graph modifying instructions into relational logic. The injectivity condition ensures that all elements of the LHS, the NACs, and the PACs are mapped to exactly one element in a matching host graph and each variable must be bound to a distinct object of the universe. The generated *injectivity* condition performs a pairwise test of inequality on all variables bound by the same expression or relation. For example, given variables $\mathsf{c1}, \mathsf{c2}$, both of which are bound by $\mathsf{C}_{:1}$, the condition $\neg(\mathsf{c1} = \mathsf{c2})$ is generated to ensure that $\mathsf{c1}$ and $\mathsf{c2}$ are assigned to two different objects. The second condition ensures that no *dangling edges* are left behind after deleting nodes from the graph. This implies that all possible references to and from a node that is scheduled for deletion need to be deleted explicitly. We translate this requirement into a condition that checks whether the set of all possible references from and to an object that is scheduled for deletion coincides with the set of actually deleted references. For example, in Figure 5.5b object obj_D, an instance of class D, is deleted together with references coming from two objects, $obj_{C,1}$ and $obj_{C,2}$, and one reference to object obj_E. Class D may have references coming from objects of class C and references to object of class E. In the following we assume that the translation generates unary relations $\mathsf{C}_{:1}$, $\mathsf{D}_{:1}$, and $\mathsf{E}_{:1}$ for classes C, D, and E, binary relations $\mathsf{C\_toD}_{:2}$ and $\mathsf{D\_toE}_{:2}$, and allocates existentially quantified variables $\mathsf{c1}$, $\mathsf{c2}$, d, and e (see Fig. 5.5 for a simplified object diagram). The formula $\mathsf{C\_toD.d} - \{\mathsf{c1}, \mathsf{c2}\} = \varnothing$ resembles the dangling edge condition for reference between objects of class C and class D. It consists of the following components:

- The relation $\mathsf{C\_toD}$ is bound to the set of all tuples $\langle obj_c, obj_d\rangle$ with $obj_c \in \bigsqcup(\mathsf{C})$

	LHS/RHS element	Formula
Preserve/Delete	Object $c \in C$	$\exists$c : C
	Reference *ref* with $src(ref) = c \in C$, $tgt(ref) = d \in D$	(c$\rightarrow$d) $\subseteq$ C_ref
	Attribute *attr* with $src(attr) = c \in C$, expression e	(c$\rightarrow expr(e)$) $\subseteq$ C_attr
Forbid	Object $c \in C$	$\neg\exists$c : C
	Reference *ref*	*same as above*
	Attribute *attr*	*same as above*
Create	Object $c \in C$	$\exists$c : C$'$ $\wedge$ c $\nsubseteq$ C
	Reference *ref*	—
	Attribute *attr*	—

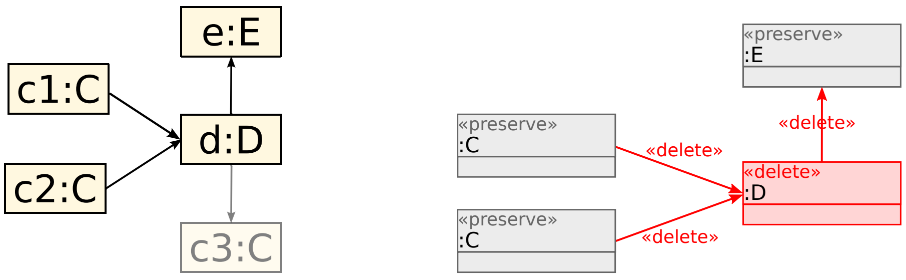
Table 5.2: Relational formulas generated by function *Pre*

and $obj_d \in \bigsqcup(\mathsf{D})$ having a *toD* reference;

- the expression C_toD.d represents the set of all *possible* objects of class C that have a *toD* reference to the object bound to d;
- the set {c1, c2} represents the *actually deleted* objects.

Thus, the formula above checks whether the set of all actually deleted objects {c1, c2} is equal to the set of all actual objects of class C that have a reference to the object bound to d. If, however, a third reference had pointed from c3 to d, the expression C_toD.d − {c1, c2} would evaluate to {c3} and thus violate {c3} $\neq \emptyset$. In the latter case the graph transformation must not be applied.

The RHS describes the effects of the graph transformation; once a matching subgraph of the LHS is found it is rewritten according to the RHS that specifies which nodes, edges, and attributes are created and/or deleted. The function *Post* generates relational formulas over *Rel* and *Rel$'$* that mimic the modifications of the transformation's RHS as follows. If the transformation creates an object obj_C of class C, two conditions are created, one by *Pre* and one by *Post*. First, function *Pre* checks for the non-existence of an object bound to relational variable c in the current state with i.e., $\exists$c : C$'$ $\wedge$ c $\nsubseteq$ C, i.e., the formula asserts that the object bound to c is *inactive* in the current state relation C. Second, function *Post* generates a condition that adds the new object (bound to c) to relation C such that the next state relation is set to C$'$ = C + c. The procedure for the



(a) Object diagram (b) A graph transformation

Figure 5.5: Dangling edge example

	LHS/RHS element	**Formula**
Delete	Object $c \in C$	$\mathsf{C}' = \mathsf{C} - \mathsf{c}$
	Reference *ref* with $src(ref) = c \in C$, $tgt(ref) = d \in D$	$\mathsf{C\_ref}' = \mathsf{C\_ref} - (\mathsf{c} \rightarrow \mathsf{d})$
	Attribute *attr* with $src(attr) = c \in C$, expression e	$\mathsf{C\_attr}' = \mathsf{C\_attr} - (\mathsf{c} \rightarrow expr(e))$
Create	Object $c \in C$	$\mathsf{C}' = \mathsf{C} + \mathsf{c}$
	Reference *ref* with $src(ref) = c \in C$, $tgt(ref) = d \in D$	$\mathsf{C\_ref}' = \mathsf{C\_ref} + (\mathsf{c} \rightarrow \mathsf{d})$
	Attribute *attr* with $src(attr) = c \in C$, expression e	$\mathsf{C\_attr}' = \mathsf{C\_attr} + (\mathsf{c} \rightarrow expr(e))$

Table 5.3: Relational formulas generated by function *Post*

deletion of an object bound to c is similar except that (i) *Pre* checks for the existence of an object that is scheduled for deletion, i.e. $\exists \mathsf{c} : \mathsf{C}$ and (ii) *Post* updates the next state relation to reflect the removal of the object (bound to c), i.e., $\mathsf{C}' = \mathsf{C} - \mathsf{c}$. Addition and deletion of (multiple) objects to and from a relation can be combined, i.e., the condition $\mathsf{C}' = \mathsf{C} + \{\mathsf{c}_1, \ldots, \mathsf{c}_m\} - \{\mathsf{c}_{m+1}, \ldots, \mathsf{c}_n\}$ states that C' is equivalent to C except that all objects bound to variables $\mathsf{c}_1, \ldots, \mathsf{c}_m$ are added to C', while objects bound to variables $\mathsf{c}_{m+1}, \ldots, \mathsf{c}_n$ are removed from C. Note that the addition and deletion of references and attributes proceed analogous to the addition and deletion of objects. For example, *Post* generates for the deletion of a reference *ref* from an object bound to c pointing to an object bound to d the formula $\mathsf{C\_ref}' = \mathsf{C\_ref} - (\mathsf{c} \rightarrow \mathsf{d})$. The formulas generated by *Post* are summarized in Table 5.3. In addition, the *Post* function also generates conditions for those relations that do not change, as otherwise arbitrary tuples could be bound to these relations.

The encodings outlined in Tables 5.2 and 5.3 translate a graph transformation $p \in \mathcal{R}$ over relations *Rel*, which is fixed w.l.o.g to $Rel = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}$ for the following explanations, into a relational formula following the scheme outlined in Figure 5.6. Here, function *match* returns constraints that mimic the transformation's LHS and control the creation of new nodes, while functions *inj* and *dec* generate injectivity constraints over nodes of the transformation's LHS and PACs/NACs and dangling edge conditions over LHS nodes, respectively.

$$\underbrace{\exists \mathsf{a1} : \mathsf{A}, \exists \mathsf{a2} : \mathsf{A}', \exists \mathsf{b} : \mathsf{B}, \exists \mathsf{c} : \mathsf{C}}_{\text{LHS,RHS, and PAC nodes}}, \underbrace{\neg\exists \mathsf{d} : \mathsf{D}}_{\text{NAC}} \mid$$

$$\underbrace{match(\mathsf{a1}, \mathsf{a2}, \mathsf{b}, \mathsf{c}, \mathsf{d})}_{\text{match constraints}} \wedge \underbrace{inj(\mathsf{a1}, \mathsf{b}, \mathsf{c}, \mathsf{d})}_{\substack{\text{injectivity}\\ \text{constraints}}} \wedge \underbrace{dec(\mathsf{a1}, \mathsf{b}, \mathsf{c})}_{\substack{\text{dangling edge}\\ \text{constraints}}} \implies$$

$$\underbrace{\mathsf{A}' = \mathsf{A} - \mathsf{a1} + \mathsf{a2} \wedge \mathsf{B}' = \mathsf{B} - \mathsf{b}}_{\text{modification constraints}} \wedge \underbrace{\mathsf{C}' = \mathsf{C} \wedge \mathsf{D}' = \mathsf{D} \wedge \mathsf{E}' = \mathsf{E}}_{\text{non-modification constraints}}$$

Figure 5.6: Scheme of a relational formula produced from a graph transformation

Example. For the *MovePacman* transformation as shown in Figure 4.4a the *Pre* and the *Post* functions generate the following set of conditions that reproduce the matching and the application of the transformation:

$$\begin{aligned}
&\exists\, \mathsf{p} : \mathsf{Pacman}, \exists\, \mathsf{f1}, \mathsf{f2} : \mathsf{Field}, \neg\exists\, \mathsf{g} : \mathsf{Ghost} \mid \\
&\quad (\mathsf{p}{\rightarrow}\mathsf{f1}) \subseteq \mathsf{Pacman\_on} \wedge (\mathsf{f1}{\rightarrow}\mathsf{f2}) \subseteq \mathsf{Field\_neighbors} \wedge \\
&\quad (\mathsf{f1}{\rightarrow}0) \subseteq \mathsf{Field\_treasure} \wedge (\mathsf{g}{\rightarrow}\mathsf{f1}) \subseteq \mathsf{Ghost\_on} \wedge \mathsf{f1} \neq \mathsf{f2} \\
&\implies \\
&\quad \mathsf{Pacman\_on}' = \mathsf{Pacman\_on} + (\mathsf{p}{\rightarrow}\mathsf{f2}) - (\mathsf{p}{\rightarrow}\mathsf{f1}) \wedge \mathsf{Game}' = \mathsf{Game} \wedge \\
&\quad \mathsf{Pacman}' = \mathsf{Pacman} \wedge \mathsf{Ghost}' = \mathsf{Ghost} \wedge \mathsf{Field}' = \mathsf{Field} \wedge \\
&\quad \mathsf{Ghost\_on}' = \mathsf{Ghost\_on} \wedge \mathsf{Game\_pacman}' = \mathsf{Game\_pacman} \wedge \\
&\quad \mathsf{Game\_ghosts}' = \mathsf{Game\_ghosts} \wedge \mathsf{Game\_fields}' = \mathsf{Game\_fields} \wedge \\
&\quad \mathsf{Field\_neighbors}' = \mathsf{Field\_neighbors} \wedge \mathsf{Field\_treasure}' = \mathsf{Field\_treasure}.
\end{aligned}$$

Graph Constraints. In contrast to graph transformations, a graph constraint does not alter a matching host graph; it may thus be used to describe a desired or an undesired pattern in a graph, i.e., a good or a bad state of the system. Thus, graph constraints are graph transformations with identical left-hand and right-hand sides. Formally, a graph constraint $c : Cond \leftarrow Lhs, c \in \Phi$ is translated into a relational formula

$$\mathrm{F}_c := Pre_c(Lhs, Pac, Nac), \tag{5.3}$$

where function $Pre_c : G \times G \times G \to \mathbb{F}$ translates the triple LHS, PAC, and NAC into a conjunction of relational formulas $f \in \mathbb{F}$. For this purpose, the encodings presented in Table 5.2 are re-used to translate a graph constraint into a relational formula. The scheme of the relational formula generated from graph constraint $c \in \Phi$ is depicted in Figure 5.7. It coincides with that of a graph transformation (see Fig. 5.6) in all but two aspects, the absence of the implication, i.e., there is no RHS, and the dangling edge condition, which is omitted because a graph constraint may not delete elements.

$$\underbrace{\exists \mathsf{a} : \mathsf{A}, \exists \mathsf{b} : \mathsf{B}, \exists \mathsf{c} : \mathsf{C}}_{\text{LHS and PAC nodes}}, \underbrace{\neg\exists \mathsf{d} : \mathsf{D}}_{\text{NAC}} \mid$$

$$\underbrace{match(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d})}_{\text{match constraints}} \wedge \underbrace{inj(\mathsf{a1}, \mathsf{b}, \mathsf{c}, \mathsf{d})}_{\substack{\text{injectivity} \\ \text{constraints}}}$$

Figure 5.7: Scheme of a relational formula produced from a graph constraint

From Relational Logic to Boolean Functions

The translation from bounded, first-order logic to propositional logic is well-known and discussed extensively in [196]. For the sake of completeness we summarize the main ideas of this relational to propositional translation briefly. The translation procedure receives as input a relational problem, consisting of a universe, a set of bounded relations, and a relational formula. It produces thereof a Boolean function in three steps, encompassing a *relation-to-matrix* translation, *symmetry detection*, and *circuit transformation* [196]. In the following we focus on the first step of the translation procedure as the latter two are not used by our approach.

The overarching idea of the translation is that relations over a finite universe can be expressed as matrices over Boolean values [95, 196]. That is, the translation of a relation of arity n allocates a matrix with n-dimensions, where the size or length of each dimension is equal to the number of elements in the universe. For example, given a universe $U = \{obj_1, \ldots, obj_m\}$ with m entries and a relation of arity n the translation produces an

$$\underbrace{m \times m \times \cdots \times m}_{n\text{-times}} = m^n \text{ matrix.}$$

In fact, all relational expressions are represented by matrices, that is, the application of a relational operator like union, product, or transitive closure to its arguments is translated into the application of a corresponding matrix operation to the matrix representations of its arguments. In this regard, relational variables are represented by matrices, too. That is, the evaluation of the expression that the relational variable is bound to upon declaration is translated into a matrix, too. Consequently, all operations on relations and relational expression are translated into operations on matrices and their application yields a Boolean matrix. Moreover, formulas over relational expressions become formulas of propositional logic over the entries of the matrices.

The entries in the matrix are determined by the lower and upper bounds of the relation. If an entry in matrix m_r generated for relation r is identified by $m_r[i_1 i_2 \dots i_n]$ with $i_j \in [1..m], 0 < j \leqslant n$ then

$$m[i_1 i_2 \dots i_n] = \begin{cases} 1 & \text{if } \langle obj_{i_1}, \dots, obj_{i_n} \rangle \in \bigsqcap(r) \\ var(r, i_1 i_2 \dots i_n) & \text{if } \langle obj_{i_1}, \dots, obj_{i_n} \rangle \in \bigsqcup(r) \backslash \bigsqcap(r) \\ 0 & \text{otherwise} \end{cases}$$

where function $var(r, i_1 \dots i_n)$ returns a Boolean variable for relation r at index $i_1 \dots i_n$. In the following, we denote by m_r the matrix generated for relation r. Further, the set of all indices in matrix m_r is given by $Idx(r)$ and a Boolean variable in matrix m_r is identified as $x_{r,i_1 \dots i_n} = var(r, i_1 \dots i_n) \in Idx(r)$ or, alternatively, r_i if index i is clear from the context. Due to the absence of lower bounds the number of Boolean variables x_i allocated for a relation X and relational variable, that is bound by relation X, is equal to the number of elements in the upper bound $\bigsqcup(\mathsf{X})$, that is, $0 < i \leqslant \text{card}(\bigsqcup(\mathsf{X}))$.

Recall from Figure 5.6 the schematic structure of a relational formula generated from a graph transformation. Each existentially quantified node variable, i.e., $\exists \mathsf{a} : \mathsf{A}$, is bound by matrix m_a, which differs from matrix m_A only at indices set to 1 or for which Boolean variables have been allocated, i.e.,

$$m_a[i_1 i_2 \dots i_n] = \begin{cases} var(a, i_1 i_2 \dots i_n) & \text{if } m_A[i_1 i_2 \dots i_n] = 1 \text{ or} \\ & \quad m_A[i_1 i_2 \dots i_n] = var(A, i_1 i_2 \dots i_n) \\ 0 & \text{otherwise.} \end{cases}$$

Note that $var(a, i_1 i_2 \dots i_n) \neq var(A, i_1 i_2 \dots i_n)$. Moreover, for each existentially quantified node variable a condition is generated that tests whether one of the Boolean variables is *active*, i.e., set to *true*. Essentially, this test is realized by an exclusive-or operation over all Boolean variables allocated for the existentially quantified node variable. The *match constraints* (see Fig. 5.6) consist of conjunctions of conditions that test for the existence of a reference between two objects. These tests are of the form $(\mathsf{a} \rightarrow \mathsf{b}) \subseteq \mathsf{A\_toB}$ with a, b being two existentially quantified relational variables whose bounds are defined by relations A and B. The *product* of two relations $(\mathsf{a} \rightarrow \mathsf{b})$ is translated into a matrix multiplication $m_a \times m_b$ that forms the conjunction of the multiplied elements. The subset-inclusion test $\mathsf{a} \subseteq \mathsf{A}$ is translated into a propositional formula $\bigwedge_{i \in Idx(A)} \neg m_a[i] \vee m_A[i]$, i.e., the existence of element $m_a[i]$, that is, $m_a[i]$ is set to *true*, implies the existence of $m_A[i]$. Note that the subset-inclusion test is only defined on relations of equal arities, as only relations of equal arity are translated into matrices of equal dimensions. Finally, the modification constraints $\mathsf{A}' = \mathsf{A} + \mathsf{a1} - \mathsf{a2}$ are translated into propositional formulas $\bigwedge_{i \in Idx(A)} m_{A'}[i] \iff m_A[i] \vee m_{a1}[i] \wedge \neg m_{a2}[i]$.

Example. Given the universe $U = \{Game1, Pacman1, Ghost1, Field1, \dots, Field4\}$ the relations $\mathsf{Game}_{:1}$, $\mathsf{Pacman}_{:1}$, $\mathsf{Ghost}_{:1}$, and $\mathsf{Field}_{:1}$ are translated into the following matrices:

$$\mathsf{Game}_{:1} = \begin{bmatrix} G_1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \mathsf{Pacman}_{:1} = \begin{bmatrix} 0 \\ P_1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \mathsf{Ghost}_{:1} = \begin{bmatrix} 0 \\ 0 \\ H_1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \mathsf{Field}_{:1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix}$$

We test with $\exists \mathsf{f1} : \mathsf{Field}, \exists \mathsf{f2} : \mathsf{Field}$ whether there exist two fields in the current state of the system. The translation of the condition produces, first, the matrix representations of the variables $\mathsf{f1}$ and $\mathsf{f2}$ and, second, an exclusive-or constraint that activates one Boolean variable from each matrix.

$$\mathsf{f1}_{:1} = \begin{bmatrix} 0 & 0 & 0 & \mathit{f1}_1 & \mathit{f1}_2 & \mathit{f1}_3 & \mathit{f1}_4 \end{bmatrix}^T \qquad \mathsf{f2}_{:1} = \begin{bmatrix} 0 & 0 & 0 & \mathit{f2}_1 & \mathit{f2}_2 & \mathit{f2}_3 & \mathit{f2}_4 \end{bmatrix}^T$$

$$\begin{array}{ll}
(\mathit{f1}_1 \wedge \neg \mathit{f1}_2 \wedge \neg \mathit{f1}_3 \wedge \neg \mathit{f1}_4)\vee & (\mathit{f2}_1 \wedge \neg \mathit{f2}_2 \wedge \neg \mathit{f2}_3 \wedge \neg \mathit{f2}_4)\vee \\
(\neg \mathit{f1}_1 \wedge \mathit{f1}_2 \wedge \neg \mathit{f1}_3 \wedge \neg \mathit{f1}_4)\vee & (\neg \mathit{f2}_1 \wedge \mathit{f2}_2 \wedge \neg \mathit{f2}_3 \wedge \neg \mathit{f2}_4)\vee \\
(\neg \mathit{f1}_1 \neg \wedge \mathit{f1}_2 \wedge \mathit{f1}_3 \wedge \neg \mathit{f1}_4)\vee & (\neg \mathit{f2}_1 \wedge \neg \mathit{f2}_2 \wedge \mathit{f2}_3 \wedge \neg \mathit{f2}_4)\vee \\
(\neg \mathit{f1}_1 \wedge \neg \mathit{f1}_2 \wedge \neg \mathit{f1}_3 \wedge \mathit{f1}_4) & (\neg \mathit{f2}_1 \wedge \neg \mathit{f2}_2 \wedge \neg \mathit{f2}_3 \wedge \mathit{f2}_4)
\end{array}$$

The relational product $\mathsf{p}{\rightarrow}\mathsf{f1}$ between relational variables p and $\mathsf{f1}$ is translated into a product operation on matrices, that is:

$$\begin{bmatrix} 0 & p_1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 0 \\ \mathit{f1}_1 \\ \mathit{f1}_2 \\ \mathit{f1}_3 \\ \mathit{f1}_4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & p_1 \wedge \mathit{f1}_1 & p_1 \wedge \mathit{f1}_2 & p_1 \wedge \mathit{f1}_3 & p_1 \wedge \mathit{f1}_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Finally, the relational formula $(\mathsf{p}{\rightarrow}\mathsf{f1}) \subseteq \mathsf{Pacman\_on}$ is translated into a conjunction of implications that logically reflect the subset-inclusion relation.

$$\bigwedge \left(\neg \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & p_1 \wedge \mathit{f1}_1 & p_1 \wedge \mathit{f1}_2 & p_1 \wedge \mathit{f1}_3 & p_1 \wedge \mathit{f1}_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \vee \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & Po_1 & Po_2 & Po_3 & Po_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \right) =$$

$$(\neg(p_1 \wedge \mathit{f1}_1) \vee Po_1) \wedge (\neg(p_1 \wedge \mathit{f1}_2) \vee Po_2) \wedge (\neg(p_1 \wedge \mathit{f1}_3) \vee Po_3 \wedge (\neg(p_1 \wedge \mathit{f1}_4) \vee Po_4)$$

The modification condition $\mathsf{Pacman\_on}' = \mathsf{Pacman\_on} + (\mathsf{p}{\rightarrow}\mathsf{f2}) - (\mathsf{p}{\rightarrow}\mathsf{f1})$ is translated into a conjunction of logical equivalences between the (Boolean) next and the current state variables with conditions for addition and deletion of *on* references.

$$\bigwedge \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & Po'_1 & Po'_2 & Po'_3 & Po'_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Longleftrightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & Po_1 & Po_2 & Po_3 & Po_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \vee$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & pf_1 & pf_2 & pf_3 & pf_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \wedge$$

$$\neg \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \hat{pf}_1 & \hat{pf}_2 & \hat{pf}_3 & \hat{pf}_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} =$$

$$Po'_1 \Longleftrightarrow (Po_1 \vee pf_1 \wedge \neg\hat{pf}_1) \wedge \ldots \wedge Po'_4 \Longleftrightarrow (Po_4 \vee pf_4 \wedge \neg\hat{pf}_4)$$

For an in-depth treatment of the translations and the rules that govern it we refer the interested reader to [196] which also explains techniques to store sparse matrices compactly. Moreover, the remaining translation steps, *symmetry detection* and *circuit transformation*, are discussed there.

From Boolean Functions to Sequential Circuits

The translation steps outlined above produce a Boolean function $f_p(\mathbf{v}_p, \mathbf{x}, \mathbf{x}')$ from a relational formula F_p (see Eq. 5.2), which encodes the matching behavior and the effects of a graph transformation $p \in \mathcal{R}$, and, similarly, a Boolean function $f_c(\mathbf{v}_c, \mathbf{x})$ from a relational formula F_c, which describes a desired or undesired state expressed by a graph constraint $c \in \Phi$. Here, the inputs $\mathbf{v}_{\{p,c\}} = (v_{a,1}, \ldots, v_{a,k}, v_{b,k+1}, \ldots)$ of f_p and f_c range over Boolean variables allocated for the existentially quantified node variables $\mathsf{a}, \mathsf{b}, \ldots$ as in $\exists \mathsf{a} : \mathsf{A}, \mathsf{b} : \mathsf{B}, \ldots$, while inputs $\mathbf{x} = (\overrightarrow{x_r}, \overrightarrow{x_s}, \ldots)$ and $\mathbf{x}' = (\overrightarrow{x_r}', \overrightarrow{x_s}', \ldots)$, with $\overrightarrow{x_r}' = \{x_{r,i} = var(r,i) \mid i \in Idx(r)\}, r \in Rel$, are Boolean variables allocated for all current and all next state relations $r, s, \ldots \in Rel$ and $r', s', \ldots \in Rel'$, respectively.

The structure of function f_p is depicted in Figure 5.8. Note that function f_c contains no modifying conditions, i.e., $f_c = pre_c(\mathbf{v}_c, \mathbf{x})$ with $\mathbf{v}_c$ being the existentially quantified node variables in graph constraint $c \in \Phi$. The function $pre_{\{p,c\}}(\mathbf{v}_{\{p,c\}}, \mathbf{x})$ returns the conjunction of constraints that, on the one hand, mimic the match conditions of the transformation's LHS, and, on the other hand, include the supplementary constraints to enforce injective matching of nodes and consistent deletion of nodes to prevent dangling edges. In essence, function $pre_{\{p,c\}}(\mathbf{v}_{\{p,c\}}, \mathbf{x})$ returns the propositional constraints generated from the relational formulas of the *Pre* function in the previous translation step

$$
\begin{aligned}
f_p(\mathbf{v}_p, \mathbf{x}, \mathbf{x}') = \mathit{pre}_p(\mathbf{v}_p, \mathbf{x}) &\implies \bigwedge_{x_{r,i} \in \mathbf{x}} \left(x'_{r,i} \Leftrightarrow \underbrace{x_{r,i} \vee \mathit{create}(\mathbf{v}_p, i) \wedge \neg \mathit{delete}(\mathbf{v}_p, i)}_{\mathit{post}_p(\mathbf{v}_p, x_{r,i})} \right) \\
\wedge \\
\neg \mathit{pre}_p(\mathbf{v}_p, \mathbf{x}) &\implies \bigwedge_{x_{r,i} \in \mathbf{x}} x'_{r,i} \Leftrightarrow x_{r,i}
\end{aligned}
$$

Figure 5.8: Structure of f_p

(see Table 5.2). The right-hand side of the implication consists of conjunctions of conditions that define the next state as it results from applying the transformation. Functions $\mathit{create}(\mathbf{v}_p, i)$ and $\mathit{delete}(\mathbf{v}_p, i)$ look-up node variables and are defined as follows:

$$
\mathit{create}(\mathbf{v}_p, i) = \begin{cases} v_{a,j} & \text{if } \exists v_{a,j} \in \mathbf{v}_p \colon j = i, \\ \mathit{false} & \text{otherwise.} \end{cases} \quad \mathit{delete}(\mathbf{v}_p, i) = \begin{cases} v_{a,j} & \text{if } \exists v_{a,j} \in \mathbf{v}_p \colon j = i, \\ \mathit{false} & \text{otherwise.} \end{cases}
$$

These functions return a Boolean variable with index i from $\mathbf{v}_p$ that is allocated for an existentially quantified node variable and is either created or deleted by transformation p, If no such variable exists in $\mathbf{v}_p$, *false* is returned. As there must not be an object that is both created and deleted the condition $\forall i \in \mathit{Idx}(r)\colon \mathit{create}(\mathbf{v}_p, i) \neq \mathit{delete}(\mathbf{v}_p, i)$ holds for all relations $r \in \mathit{Rel}$. Finally, note that the satisfiability of f_p indicates whether transformation p can be applied with the current assignments to $\mathbf{v}_p$, $\mathbf{x}$, $\mathbf{x}'$ or not, i.e., the assignments are inconsistent. In the following, we extract from $f_p(\mathbf{v}_p, \mathbf{x}, \mathbf{x}')$ the *next state function* $\mathbf{x}' = \hat{f}_p(\mathbf{v}_p, \mathbf{x})$ that returns as its output the next state $\mathbf{x}'$ that results from applying the transformation p. The extraction is based on the observation that function f_p be can be rewritten into a *logically equivalent* formula by distributing the condition $\mathit{pre}_p(\mathbf{v}_p, \mathbf{x})$ over the conjunction of $x_{r,i} \in \mathbf{x}$, which is subsequently expanded to yield conjunctions of formulas $\mathit{pre}_p(\mathbf{v}_p, \mathbf{x}) \Rightarrow (x'_{r,i} \Leftrightarrow \mathit{post}_p(\mathbf{v}_p, x_{r,i})) \wedge \neg \mathit{pre}_p(\mathbf{v}_p, \mathbf{x}) \Rightarrow (x'_{r,i} \Leftrightarrow x_{r,i})$. Finally, the next state variable $x'_{r,i}$ is factored out yielding formula

$$
f_p(\mathbf{v}_p, \mathbf{x}, \mathbf{x}') = \bigwedge_{x_{r,i} \in \mathbf{x}} x'_{r,i} \iff \big((\mathit{pre}_p(\mathbf{v}_p, \mathbf{x}) \Rightarrow \mathit{post}_p(\mathbf{v}_p, x_{r,i})) \wedge (\neg \mathit{pre}_p(\mathbf{v}_p, \mathbf{x}) \Rightarrow x_{r,i}) \big)
$$

As a consequence of the equality in the above equation the next state function $\hat{f}_p$ returns an n-ary tuple of next state formulas, i.e., for each of the $n = \sum_{i=1, r \in \mathit{Rel}}^{\mathrm{card}(\mathit{Rel})} \bigsqcup(r)$ state variables the right-hand side of the above equation is returned. Then, $\hat{f}_p$ takes on the structure depicted in Figure 5.9, where $r, s, t \in \mathit{Rel}$ and $i, k, j \in \mathit{Idx}(x)_{x \in \{r,s,t\}}$.

In all but the most trivial systems the behavior is described by multiple graph transformations. To build the *latch function* $\ell_{x_{r,i}}(\mathbf{V}, \mathbf{x})$, that determines the next state value of each Boolean variable $x_{r,i}$ based on the inputs $\mathbf{V} = \bigcup_{p \in \mathcal{R}} \mathbf{v}_p$ and the current state $\mathbf{x}$,

$$\begin{aligned}\hat{f}_p(\mathbf{v}_p,\mathbf{x}) = ((&pre_p(\mathbf{v}_p,\mathbf{x}) \Rightarrow post_p(\mathbf{v}_p,x_{r,i})) \wedge (\neg pre_p(\mathbf{v}_p,\mathbf{x}) \Rightarrow x_{r,i}),\\ &\vdots\\ (&pre_p(\mathbf{v}_p,\mathbf{x}) \Rightarrow post_p(\mathbf{v}_p,x_{s,j})) \wedge (\neg pre_p(\mathbf{v}_p,\mathbf{x}) \Rightarrow x_{s,j}),\\ &\vdots\\ (&pre_p(\mathbf{v}_p,\mathbf{x}) \Rightarrow post_p(\mathbf{v}_p,x_{t,k})) \wedge (\neg pre_p(\mathbf{v}_p,\mathbf{x}) \Rightarrow x_{t,k}))\end{aligned}$$

Figure 5.9: Structure of the next state function $\hat{f}_p$.

the outputs from the next state functions $f_{p_1},\ldots,f_{p_n}$ need to be conjoined accordingly. That is, the conjunction of all pre-post implications $pre_p(\mathbf{v}_p,\mathbf{x}) \Rightarrow post_p(\mathbf{v}_p,x_{r,i}), p \in \mathcal{R}$ and the conjunction of all negated preconditions, $pre_p(\mathbf{v}_p,\mathbf{x}), p \in \mathcal{R}$, that effect no modification on the state variable $x_{r,i}$ are formed to yield the following formula, where $n = \text{card}(\mathcal{R})$:

$$\begin{aligned}\ell_{x_{r,i}}(\mathbf{V},\mathbf{x}) = pre_1(\mathbf{v}_1,\mathbf{x}) \implies post_1(\mathbf{v}_1,x_{r,i}) \wedge \cdots \wedge pre_n(\mathbf{v}_n,\mathbf{x}) \implies post_n(\mathbf{v}_p,x_{r,i}) \wedge\\ (\neg pre_1(\mathbf{v}_1,\mathbf{x}) \wedge \neg pre_2(\mathbf{v}_1,\mathbf{x}) \wedge \cdots \wedge \neg pre_n(\mathbf{v}_n,\mathbf{x})) \implies x_{r,i}\end{aligned}$$

From Sequential Circuits to AIGER Specifications

The construction of the AIGER specification is straightforward; first, initialize for each variable $v_{r,i} \in \mathbf{V}$ an input variable in the AIGER specification and, second, declare a latch variable for each current state variable in $\mathbf{x}$. Then, translate each latch function $\ell_{x_{r,i}}(\mathbf{V},\mathbf{x})$ into an AIG by iteratively applying DeMorgan's laws to convert implications into combinations of conjunctions and their negations. Likewise, the graph constraints are converted into AIGs and registered in the AIGER specification as a *bad property.* In general, a graph constraint $c \in \Phi$ may describe a reachable state of the system, which may not necessarily be a bad state; but for the purpose of analyzing the correctness of a system the latter will most often be of more interest because if c is proven unreachable it follows that $\mathsf{AG}\,\neg c$ holds. Finally, the verification problem in the AIGER specification can be analyzed with a model checker. For each bad property in the AIGER specification the model checker verifies whether the described state is reachable. If the bad state is reachable, the model checker returns a witness. A witness lists the sequences of input values that, if inserted into the sequential circuit, can be used to reconstruct the execution trace starting in the initial state and leading to the bad state as follows. By inserting and evaluating each set of input values on sequence of their appearance in the reported witness, the states, i.e., the latch values, can be recorded to construct a counter example trace. For this purpose, each set latch variable is mapped to its corresponding upper bound element of a relation, which in turn is mapped to an object, a reference, or an attribute of a class from model $\mathbb{M}$. Thus, the sequence of recorded states from the sequential circuit can be mapped back to the relational representation and eventually to the EMF model and the applications of graph transformations. In case the state described by $c \in \Phi$ is found unreachable, the model checker reports the satisfiability of $\mathsf{AG}\,\neg c$.

5.3 Summary

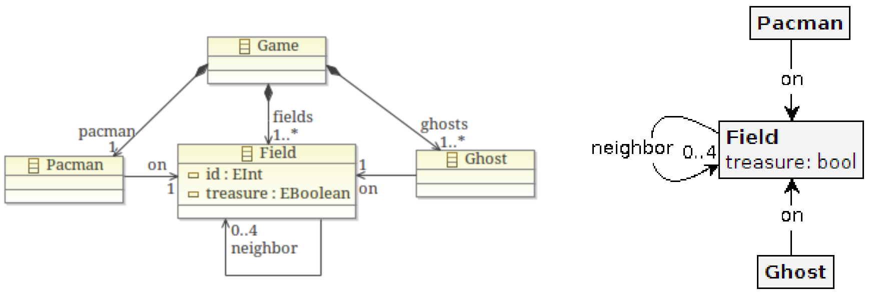
In this chapter, we introduced our symbolic model checking approach that proves a system safe from reaching some bad state. Our approach takes as input an EMF model that describes the structure of the system, a set of graph transformations that defines the behavior of the system, an initial state of the system, and a specification that consists of a set of graph constraints that describe desired and undesired states to be proven reachable or unreachable, respectively. Moreover, a set of upper bounds needs to be provided in order to construct finite representations of the system. The inputs are translated into relation logic formulas, which are subsequently converted into Boolean functions. A final translation step extracts a sequential circuit from the Boolean functions that is then stored in the AIGER format. The AIGER format, being the input format of the Hardware Model Checking Competition, allows us to take advantage of the some of the fastest model checkers currently available. Once the verification is completed by the externally invoked model checker, the result is analyzed. In case a bad state was reached, the model checker returns a witness that is translated back into a counterexample trace of instance models, i.e., system states. The above described procedure is implemented in GRYPHON that is evaluated in the next chapter and compared to GROOVE [106] and MOCOCL.

CHAPTER 6

Evaluation

In this chapter we compare and evaluate MocOCL, Gryphon, and Groove on a set of three benchmarks: the Pacman game presented in Section 6.1, the Dining Philosophers problem [52], and the verification of real-world railway interlockings inspired by [98]. We present first the Pacman game benchmark that, in contrast to the version in Section 6.1, terminates correctly. Then, we discuss the Dining Philosophers benchmark, and finally introduce the benchmark for the verification of railway interlockings. For each benchmark, we describe its implementation in terms of EMF models and Henshin transformations as well as in terms of attributed, typed graphs and corresponding graph transformations as used by Groove. For the evaluation we use two groups of specifications. The first group contains safety specifications of the form $\mathsf{AG}\,\neg\phi$, where ϕ denotes a bad state. Thus, all three tools participate in this comparison. The second group contains arbitrary safety and liveness properties formulated in CTL. Because Gryphon supports currently neither arbitrary safety specifications nor liveness properties, only MocOCL and Groove are compared on this second set. Finally, we record the results from running the benchmarks and briefly discuss them.

Benchmark setup. The benchmarks were run on an Intel$^{\text{TM}}$Core i5 M580 2.67GHz CPU with 8GB of RAM running Gentoo 2.2 (Linux kernel 3.14.14). For the benchmarks we use the Oracle$^{\text{TM}}$Java$^{\text{TM}}$SE 7 Runtime Environment (build 1.7.0_71-b14), the Henshin API in version 0.0.1, and Groove in version 5.5.2 (build: 20150324114640). The heap size for each benchmark run was set to at most 6GB and the timeout was set to 720 seconds. The runtimes of each benchmark were averaged over 10 consecutive runs. Each run of Groove is executed separately and executed from the command line with `java -jar Generator.jar -a` *formula graphgrammar startgraph*. In contrast, MocOCL and Gryphon are executed in batch mode, which allows to execute multiple runs with a single call to `java -jar mococl.jar benchmark.config` and `java -jar gryphon.jar benchmark.cfg`, respectively. Note that due to the small number of executions the just-in-time compilation provided by the Java Virtual Machine is negligible in



(a) EMF Model (b) Groove Model

Figure 6.1: The Pacman Game model

the comparison of the three tools. Further, note that MocOCL supports enumerations only on the back-end, while the web front-end will terminate with an exception. Due to this restriction, all EMF models and the corresponding Henshin implementation of their behavior presented in the following sections use integer values instead of enumeration literals.

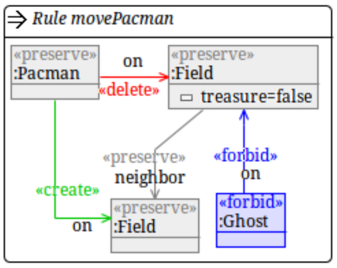
6.1 The Pacman Game

The Pacman Game is played on a board consisting of an arbitrary number of fields. A field has at most four neighboring fields and may host a treasure. The game is played turn wise, but players may choose not to move. The game is over if either of the following termination conditions are satisfied. Pacman wins the game, if he finds a treasure, and he looses the game, if a ghost meets him on the same field.

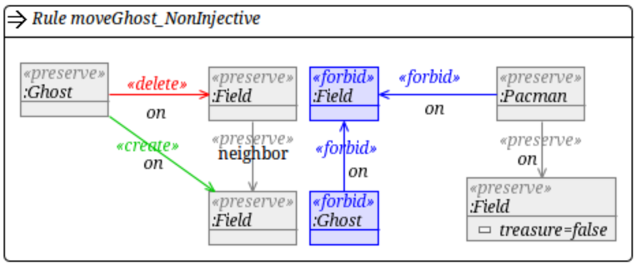
Modeling the Pacman Game

The static structure of the game's implementation follows the above description closely. The `Game` class acts as the container for the fields, the ghosts, and Pacman. Each `Field` instance is identified by an `id` attribute. A Boolean flag indicates whether the field contains a `treasure`. The `neighbors` reference points to at most four adjacent fields. Both `Pacman` and an arbitrary number of `Ghosts` are assigned to a field through their `on` reference. The static structure of the Pacman Game is depicted in Fig. 6.1 presenting both the implementation in EMF (Fig. 6.1a) and in Groove (Fig. 6.1b). In contrast to the EMF based implementation, the implementation in Groove does not require a root container; hence, the `Game` class was removed for Groove.

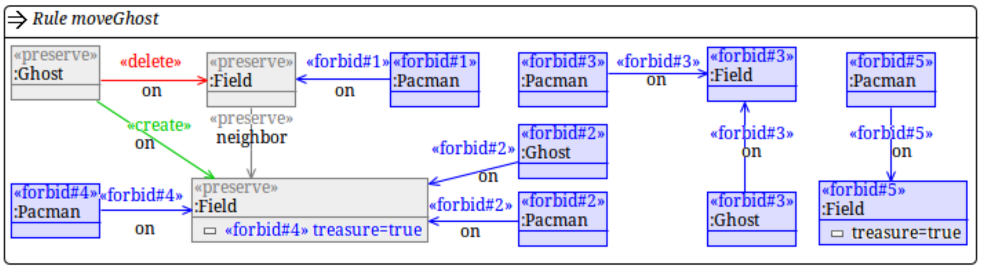
The behavior of Pacman and the ghosts is modeled by two graph transformations, *MovePacman* and *MoveGhost*, which are shown in Figure 6.2. The *MovePacman* transformation moves Pacman from its currently occupied field to an arbitrary, i.e., nondeterministically chosen, neighboring fields if he resides neither on a treasure field nor on



(a) *MovePacman*



(b) Non-Injective *MoveGhost*



(c) *MoveGhost*

Figure 6.2: *MovePacman* and *MoveGhost* transformations implemented with Henshin

the same field as a ghost. Similarly, the *MoveGhost* transformation moves an arbitrarily chosen ghost to a neighboring field if neither of the termination conditions is satisfied. In contrast to the previously presented erroneous implementation, the transformation shown in Figure 6.2c has been corrected such that a ghost may no longer move if either Pacman found a treasure or if Pacman and some ghost reside on the same field. This requires five negative application conditions, i.e., *forbid#1* through *forbid#5*, each of which prevents the movement of the ghost (#1) if the ghost has already found Pacman, (#2) if another ghost found Pacman on the neighboring field, (#3) if another ghost found Pacman on a field other than the current field of the ghost or its neighboring field, (#4) if Pacman found the treasure on the neighboring field, (#5) if Pacman found the treasure on a field other than the neighboring field. By disabling injective matching of rule elements the *MoveGhost* transformation can be simplified as displayed in Figure 6.2b. In this later implementation of the *MoveGhost* transformation all case distinction necessary to determine whether a ghost and Pacman share the same field collapse into one negative application condition. Further, the match is extended to ensure that Pacman resides on a non-treasure field.

The Groove implementation closely follows the implementation of the Henshin implementation as depicted in Figure 6.3 with the notable difference that Groove uses a non-injective matching algorithm by default. That is, two nodes of equal type in the LHS of a transformation may be mapped to the same node in the host graph. Further, note

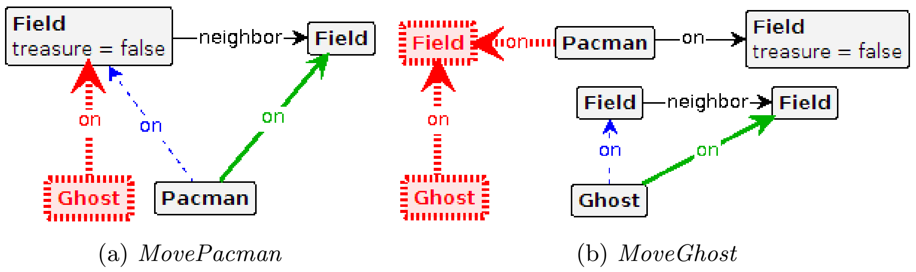


Figure 6.3: *MovePacman* and *MoveGhost* transformations implemented with GROOVE

that GROOVE uses a slightly different notation than HENSHIN: nodes and edges marked for deletion are colored in blue, while elements of negative application conditions are are indicated by thick, dashed red lines. Moreover, GROOVE by default uses a non-injective matching engine.

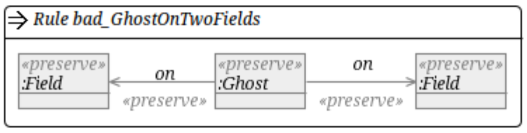
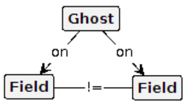
Specification

We define the following invariants for the Pacman Game, which we derive from the rules of the game:

- No ghost may be assigned to two different fields at any point in time.
- Pacman and each ghost are always assigned to one field.
- A field has no more than four different neighbors.

In Table 6.1, we list the formulation of the above properties for MocOCL, GRYPHON, and GROOVE. For each property, we provide an identifying keyword and the natural language formulation followed by the implementation of the property either as cOCL expression for MocOCL, as a graph constraint expressing either an invariant or a bad property for GRYPHON, or a combination of CTL operators and graph constraints in case of GROOVE. We compare the different implementations in the following.

The implementation of the `TWO_FIELDS` property as a graph constraint is arguably the simplest to implement as we define a *bad state* pattern of a ghost that maintains two `on` references to two different fields. Note that this pattern also encompasses bad states where a ghost maintains more than two `on` references to different fields. In GRYPHON, we then test for unreachability of this bad state pattern. In GROOVE we negate the graph constraint, which is equivalent to the one implemented for GRYPHON, in the formula. Note the *inequality* edge labeled with `!=` between the two `Field` nodes. As GROOVE uses a non-injective matching algorithm that may map two or more nodes of the same type to a single node (of matching type) in the host graph, we need to either explicitly state with inequality edges that the two (or more) nodes must be mapped to different nodes in the host graph or, alternatively, enable the `match injective` property for a

`TWO_FIELDS`: *No ghost may be assigned to two different fields at any point in time.*	
cOCL	`Always Globally ghosts->forAll(on->size()=1)`
GRYPHON	 Rule bad_GhostOnTwoFields «preserve» :Field on «preserve» «preserve» :Ghost on «preserve» «preserve» :Field
GROOVE	`AG !ghostOnTwoFields` with `ghostOnTwoFields` = 
`ON_FIELD`: *Pacman and each ghost are always assigned to one field.*	

cOCL	`Always Globally not pacman.on.oclIsUndefined() and`
	`  ghosts->forAll(on->size()=1)`

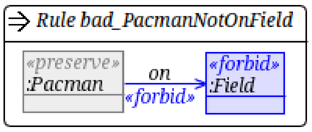
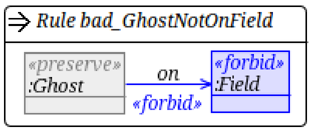
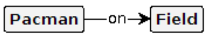
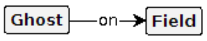
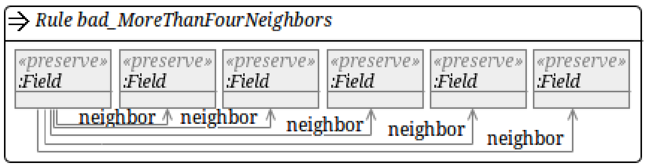
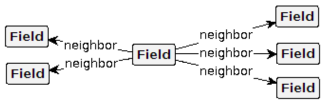
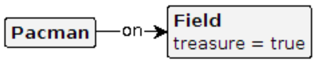
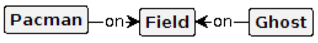
GRYPHON	 Rule bad_PacmanNotOnField «preserve» :Pacman on «forbid» «forbid» :Field  Rule bad_GhostNotOnField «preserve» :Ghost on «forbid» «forbid» :Field
GROOVE	`AG ((pacmanOnField) & (ghostOnField))`, where graph constraint `pacmanOnField` is defined as  and graph constraint `ghostOnField` is defined as .
`NEIGHBORS`: *A field has no more than four different neighbors.*	
cOCL	`fields->forAll(neighbors->size()<5)`
GRYPHON	 Rule bad_MoreThanFourNeighbors «preserve» :Field «preserve» :Field «preserve» :Field «preserve» :Field «preserve» :Field «preserve» :Field neighbor neighbor neighbor neighbor neighbor
GROOVE	`AG !moreThanFourNeighbors`, where `moreThanFourNeighbors` is defined as  Field Field neighbor neighbor Field neighbor neighbor neighbor Field Field Field. Note that we enable rule property `match injective` as this constraint requires the matching algorithm to map nodes injectively.

Table 6.1: Invariant properties formulated for use in MocOCL, Gryphon, and Groove

`GAME_OVER`: *The game ends if either Pacman found the treasure or if Pacman and a ghost move to the same field.*	

cOCL	`Always Globally (pacman.on.treasure or`
	`  ghosts->exists(g|g.on=pacman.on)) implies`
	`    Always Next false`

GROOVE	`AG !(pacmanOnTreasure | ghostOnPacman) | AX false`, where graph constraints `pacmanOnTreasure` and `ghostOnPacman` denote graphs  and .

`FAIR_MOVE`: *If the game is not over yet and Pacman resides on a field that has a neighbor, he will move to this neighboring field at some point in the future.*	

cOCL	`Always Globally not pacman.on.treasure and`
	`  ghosts->forAll(g|g.on<>pacman.on) implies`
	`    pacman.on.neighbors->`
	`      forAll(n|Exists Eventually pacman.on=n)`

GROOVE	—

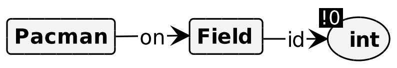
Table 6.2: Safety and liveness properties formulated for use in MocOCL and GROOVE

transformation [166, p. 22] that switches the matching algorithm into injective node mapping mode. The cOCL expression, in contrast, is arguably the more pragmatic solution: we simply check whether the `on` reference points, at all times, to exactly one field. For the implementation of the second property, named `ON`, in cOCL, we check that Pacman is always placed on a field, i.e., its `on` reference is never *undefined*, and reuse the cOCL expression of the first property to assert that all ghost's `on` reference points to exactly one field. In GRYPHON, we implement the property with two *bad* patterns and, similarly, we use two positively formulated graph constraints in GROOVE's implementation. The implementation in cOCL of the final property, named `NEIGHBORS`, again exploits the `size()` function provided by OCL that all fields have no more than for pointers to neighboring fields. In GRYPHON and GROOVE, we explicitly model the bad pattern that shall be unreachable, i.e., a field with five (or more) neighbors. Note that in case of GROOVE we enable the injective matching algorithm via the `match injective` property (see [166, p. 22]) instead of using ten inequality edges between the `Field` nodes.

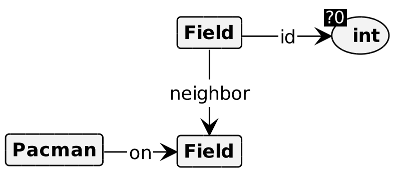
Further, we analyze the following two properties that ensure that the game is reasonably fair and that it terminates correctly.

- The game ends if either Pacman found the treasure or if Pacman and a ghost move to the same field.

- If the game is not over yet and Pacman resides on a field that has a neighbor, he will move to this neighboring field at some point in the future.

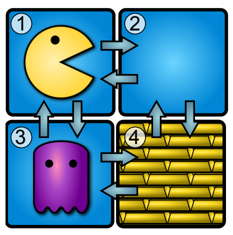


(a) `pacmanOnFieldWithID(id)`

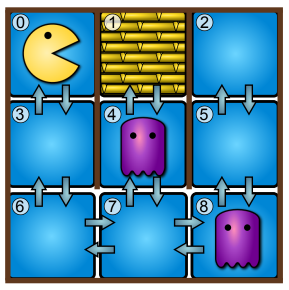


(b) `pacmanOnNeighborOfFieldWithID(id)`

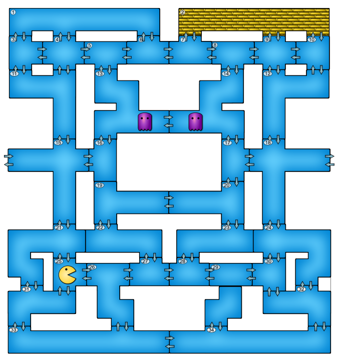
Figure 6.4: Parameterized graph constraints for GROOVE's `FAIR_MOVE` property



(a) 2x2 board



(b) 3x3 board



(c) Large board

Figure 6.5: Game fields used in the performance evaluation

In Table 6.2, we list the implementation of these properties for MOCOCL and GROOVE. The implementations of the `GAME_OVER` property are similar in nature: first, we check if Pacman found the treasure or a ghost found Pacman. Both of these condition define a terminating state as they entail that Pacman either won or lost game, respectively. Then, we ensure with $AX false$ that there exists no follow-up state after such a terminating state. For the cOCL implementation of the `FAIR_MOVE` property, we first check whether the current state is a non-terminating state which implies that Pacman should be able to move to one of his neighboring fields at some point in the future. An implementation of this property in GROOVE was deemed impractical to realize because GROOVE does not allow the direct passing of parameters to transformation or graph constraints in CTL formulas yet. Otherwise, we could have formulated the property as `AG !(!pacmanOnTreasure & !ghostOnPacman) | (pacmanOnFieldX(id) & EF pacmanOnNeighborOfFieldX(id))` where graph constraints `pacmanOnTreasure` and `ghostOnPacman` are reused from the `GAME_OVER` implementation (see Table 6.2). Moreover, the implementation requires two parameterized graph constraints, `pacmanOnField-WithID(id)` and `pacmanOnNeigborOfFieldWithID(id)`. The first constraint matches the field that Pacman currently occupies and stores its `id` value in the out-parameter of the same name, while the second constraint checks whether Pacman occupies a field which is a neighbor of the field whose `id` value is equal to the in-parameter of the same name. Both of these constraints are depicted in Figure 6.4.

		Groove		MocOCL		Gryphon	
		Total	σ	Total	σ	Total	σ
TWO_FIELDS	Large (0 ghosts) St: 36 Tr: 107	2495	789.98	104	1.65	249	280.92
	Large (1 ghosts) St: 1269 Tr: 7381	4766	1507.69	5623	132.76	9232	191.72
	Large (2 ghosts) St: 25902 Tr: 209934	24564	7773.57	(TO)	(TO)	86745	799.30
ON_FIELD	Large (0 ghosts) St: 36 Tr: 107	2471	784.80	3142	23.97	558	286.91
	Large (1 ghosts) St: 1269 Tr: 7381	4760	1508.33	5603	222.87	8526	594.97
	Large (2 ghosts) St: 25902 Tr: 209934	24645	7800.45	(TO)	(TO)	5648	150.01
NEIGHBORS	Large (0 ghosts) St: 36 Tr: 107	2512	797.44	3634	18.99	3877	24.83
	Large (1 ghosts) St: 1296 Tr: 7381	4817	1533.49	6270	294.26	3634	15.77
	Large (2 ghosts) St: 25902 Tr: 209934	24667	7805.50	(TO)	(TO)	3651	24.89
GAME_OVER	Large (0 ghosts) St: 36 Tr: 107	2483	788.67	116	0.82	—	—
	Large (1 ghosts) St: 1296 Tr: 7381	4800	1519.17	149	0.96	—	—
	Large (2 ghosts) St: 25902 Tr: 209934	24839	7868.44	772	124.02	—	—
FAIR_MOVE	Large (0 ghosts) St: 36 Tr: 107	—	—	169	1.29	—	—
	Large (1 ghosts) St: 1296 Tr: 7381	—	—	12960	365.78	—	—
	Large (2 ghosts) St: 25902 Tr: 209934	—	—	(TO)	(TO)	—	—

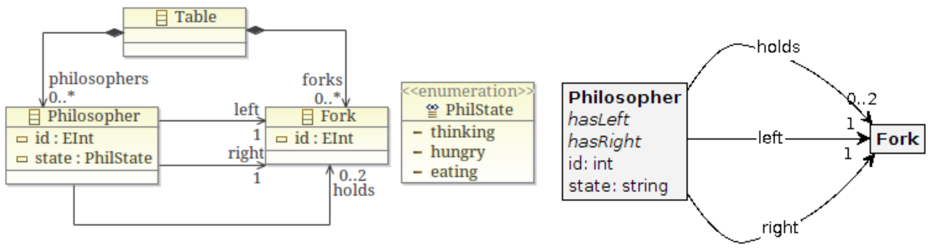
Table 6.3: Pacman game benchmarks

Benchmark Results

For the performance evaluation we define three different initial states of varying board sizes. These are depicted in Figure 6.5. The initial state depicted in Figure 6.5a is a simple 2x2 game board with one ghost and one treasure field. Similarly, the initial state shown in Figure 6.5b is a 3x3 game board with two ghosts, and barriers around the single treasure field. The third board is a simplification of a real Pacman level with 32 fields, seven treasure fields, and two ghosts. The runtime results are provided in Table 6.3.

6.2 The Dining Philosophers Problem

The Dining Philosophers problem was originally proposed by Dijkstra [52] and popularized by Hoare [92] as a figurative example for mutual exclusion and synchronization problems. In the problem's setup a group of philosophers sits around a round table, a plate in front of each of them, and a fork on each side of the plate. Each philosopher requires two forks to commence eating. These forks, however, are shared with the philosophers sitting to the left and to the right of each philosopher. In order to obtain



(a) EMF Model (b) GROOVE Model

Figure 6.6: Dining Philosophers Model

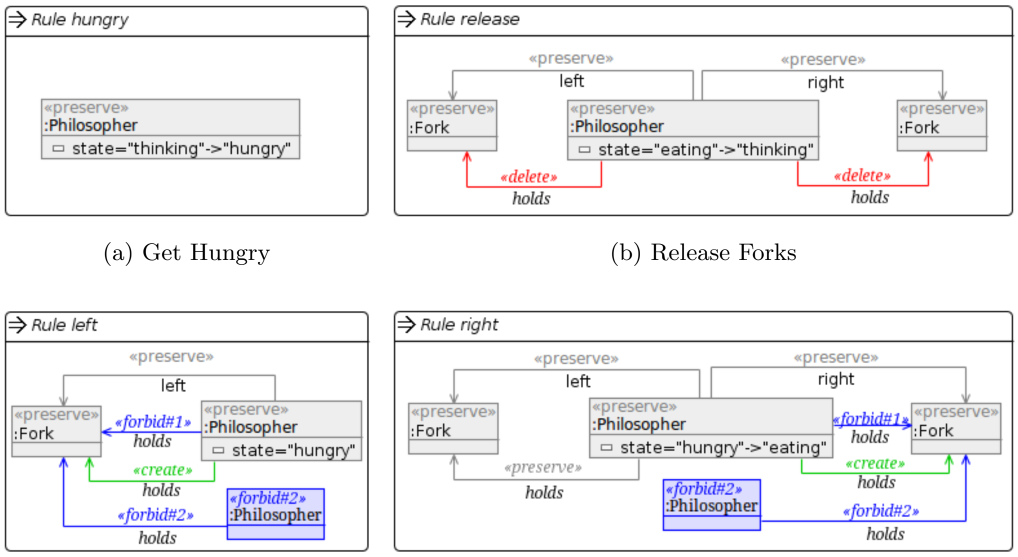
two forks, each philosopher transitions through a sequence of three states: *thinking*, *hungry*, and *eating*. Once they finish eating they go back into thinking. The classical specification proposed with the Dining Philosophers problem asserts whether the system may run into a deadlock.

Modeling the Dining Philosophers Problem

The static structure of the Dining Philosophers Problem's implementation follows the described problem setup. A `Table` class acts as a container for forks and philosophers. Each `Philosopher` is identified by an `id` and may be either in `state` *thinking*, *hungry*, or *eating* as defined by the `PhilState` enumeration. Further, each philosopher has a `left` and `right` fork and the `holds` reference points to the fork(s) currently held by the philosopher. Finally, each `Fork` is identified by an `id`. The resulting model implemented in EMF and in GROOVE is shown in Figure 6.6. In contrast to the EMF model, the implementation in GROOVE does not require a container object; hence, the `Table` has been dropped from the GROOVE model. Further, since GROOVE does not support enumerations, the `state` is realized as string-valued attribute. Although both the philosopher's and the fork's `id` field do not provide immediate value in the construction of instance models[1] or graph transformations we retain the philosopher's `id` field for the purpose of simplifying the formulation of the specification (see below). Moreover, we introduce two flags for the philosophers, `hasLeft` and `hasRight`, to indicate that left and the right fork, respectively, have been picked up. The metamodel, i.e., the type graph, used for the implementation of the graph transformation in GROOVE is depicted in Figure 6.6b.

The behavior of each philosopher is defined by four graph productions depicted in Fig. 6.7. The transformation shown in Figure 6.7a rewrites the philosopher's `state` attribute to transition from the state *thinking* to state *hungry*. A hungry philosopher needs to eat and attempts to acquire a left fork first; this is achieved by applying the transformation displayed in Figure 6.7c that establishes `hold` reference between a *hungry* philosopher and her `left` fork. If the philosopher already holds the left fork, the right fork may be picked up and the `state` changes from *hungry* to *eating*. This transition is

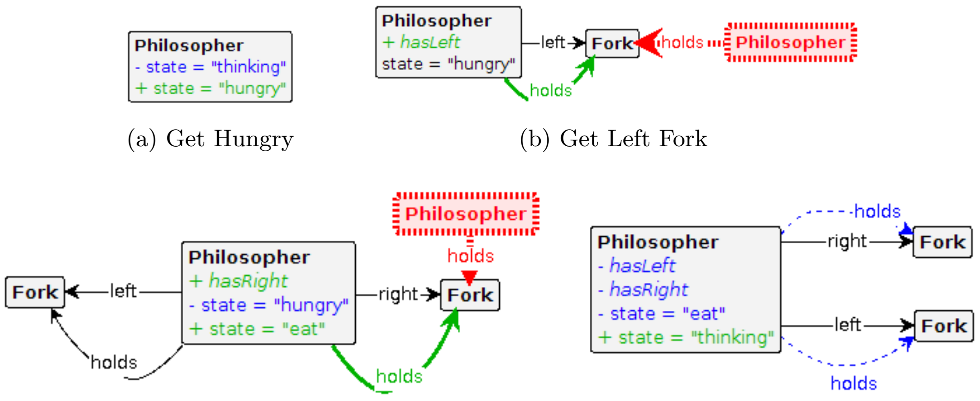
[1]In case of the EMF model we employ the `id` to ease the construction of instance models.



(a) Get Hungry (b) Release Forks

(c) Get Left Fork (d) Get Right Fork & Eat

Figure 6.7: Graph Transformations for the Dining Philosophers implemented in HENSHIN



(a) Get Hungry (b) Get Left Fork

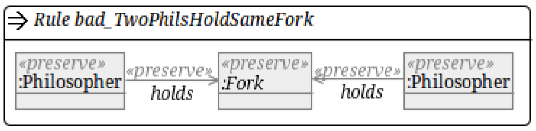
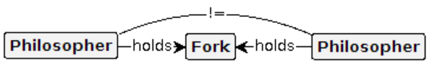
(c) Get Right Fork & Eat (d) Release Forks

Figure 6.8: Graph Transformations for the Dining Philosophers implemented in GROOVE

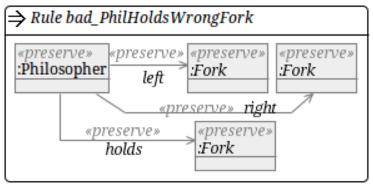
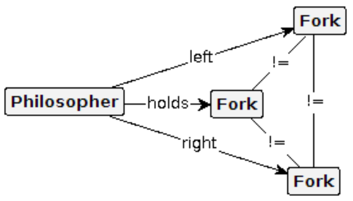
performed by the transformation depicted in Figure 6.7d that creates a `holds` reference between a *hungry* philosopher and her `right` fork and changes her state from *hungry* to *eating* provided that she already `holds` her `left` fork. Once the philosopher is done eating, the forks are put back on the table and the philosopher switches to `state` *thinking* again; the corresponding transformation is shown in Figure 6.7b. This last transformation achieves the described effect by deleting all `hold` references between a philosopher and her `right` and `left` forks and changes her `state` from *eating* to *thinking*.

SAME_FORK: *No two philosophers hold the same fork.*	

cOCL	`Always Globally philosophers->forAll(p1,p2｜p1<>p2 implies`
	`  (not p1.holds->includes(p2.holds) and`
	`    not p2.holds->includes(p1.holds)))`

GRYPHON	 Rule bad_TwoPhilsHoldSameFork «preserve» :Philosopher «preserve» holds «preserve» :Fork «preserve» holds «preserve» :Philosopher
GROOVE	`AG !philsHoldSameFork` where `philsHoldSameFork` is defined as .
LEFT_RIGHT: *If a philosopher holds a fork, its either her left or her right fork.*	

cOCL	`Always Globally philosophers->`
	`  forAll(p｜let left=p.left in let right=p.right in`
	`    p.holds->forAll(f｜(f=left or f=right)))`

GRYPHON	 Rule bad_PhilHoldsWrongFork «preserve» :Philosopher «preserve» left «preserve» :Fork «preserve» :Fork «preserve» right «preserve» holds «preserve» :Fork
GROOVE	`AG !philHoldsWrongFork` where `philHoldsWrongFork` is defined as  Fork left != Philosopher holds Fork != right != Fork
DEADLOCK: *The philosophers do not deadlock.*	

cOCL	`Always Globally not philosophers->`
	`  forAll(p｜p.holds=Set{p.left})`

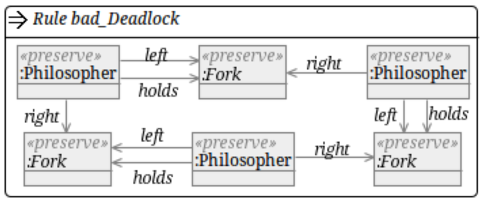
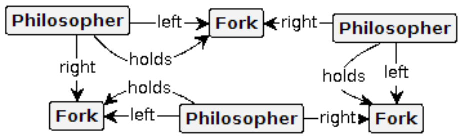
GRYPHON	 Rule bad_Deadlock «preserve» :Philosopher left holds «preserve» :Fork right «preserve» :Philosopher right left holds «preserve» :Fork left holds «preserve» :Philosopher right «preserve» :Fork
GROOVE	`AG !deadlock` where `deadlock` is defined as  Philosopher left Fork right Philosopher right holds holds holds left Fork left Philosopher right Fork

Table 6.4: Invariant properties formulated for use in MocOCL, GRYPHON, and GROOVE

The non-injective implementation of the above described transformations in GROOVE is shown in Figure 6.8. In Figure 6.8a, we observe that the implementation of the *GetHungry* transformation in GROOVE visualizes updates to attributes destructively, i.e., the old attribute value "thinking" is deleted before the new attribute value "hungry" is created. The implementation of the *GetLeftFork* transformation (Fig. 6.8b) establishes a `holds` edge between the philosopher and her right fork and sets the `hasLeft` flag. In contrast to the implementation in HENSHIN, the transformation in GROOVE requires only one negative application condition as the forbidden pattern of a philosopher holding the to-be-acquired fork may match any philosopher including the philosopher matched by the LHS that tries to "acquire" the fork but might already hold it. In a similar fashion, we exploit the non-injective matching of GROOVE in the *GetRightForkAndEat* transformation (Fig. 6.8c), which also sets the `hasRight` flag on the philosopher. The *ReleaseForks* transformation (Fig. 6.8d) clears the `hasLeft` and `hasRight` flags from the philosopher but is otherwise equivalent to the implementation in HENSHIN (see Fig. 6.7b).

Specification

Similar to the Pacman game we identify the following invariants based on the rules that define the Dining Philosophers Problem:

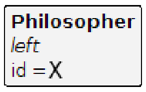
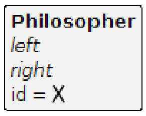
- The number of philosophers and the number of forks remain constant throughout the game. Moreover, the number of forks placed on the tables is equal to number of philosophers.
- No two philosophers hold the same fork.
- If a philosopher hold a fork, its either her left or her right fork.
- The philosophers do not deadlock.

In Table 6.4, we list the formulation of the above properties for MocOCL, GRYPHON, and GROOVE. Again, we provide for each property an identifying keyword and the natural language formulation followed by the implementation of the property either as cOCL expression for MocOCL, as an invariant or bad property for GRYPHON, or a combination of CTL operators and graph constraints in case of GROOVE, which we compare in the following.

While property `SAME_FORK` expressed in cOCL requires us to ensure that neither of two philosophers holds a fork the other one has already acquired and vice versa, we simply describe the undesired state as a graph constraint for GRYPHON and GROOVE. Note that due to GROOVE's non-injective matching algorithm we need to ensure that both philosophers in the graph constraint are mapped to two different philosophers in the host graph. This is achieved by an *inequality* edge labeled with `!=` between the two philosophers. Similarly, for GROOVE's implementation of property `LEFT_RIGHT`, inequality edges are used to ensure that all three fork nodes are mapped to different forks in the host graph. Finally, cOCL allows to formulate the `DEADLOCK` property succinctly

`ACQUIRE_RIGHT`:	*If a philosopher acquired a left fork, she will eventually acquire the right fork, too.*

cOCL	`Always Globally philosophers->`
	`  forAll(p|(p.holds->includes(p.left) implies`
	`    Always Eventually p.holds=Set{p.left,p.right}))`

GROOVE	`AG (!p1HasLeft | AF p1HasLeftRight) & AG (!p2HasLeft | AF p2HasLeftRight) & ... & AG (!pNHasLeft | AF pNHasLeftRight)` where N is the number of philosophers on the table and *pXHasLeft* and *pXHasLeftRight* with $X = \{1, \ldots, N\}$ are defined as  and .
`NO_STARVATION`:	*Every philosopher will eventually eat.*

cOCL	`Always Globally philosophers->`
	`  forAll(p|Always Eventually p.status="eating")`

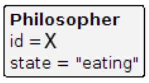
GROOVE	`AG AF p1Eating & AG AF p2Eating & ... & AG AF pNEating` where N is the number of philosophers on the table and *pXEating* with $X = \{1, \ldots, N\}$ is defined as .

Table 6.5: Safety and liveness properties formulated for use in MocOCL and Groove

as it demands that the set of forks held by at least one philosopher is different from the singleton set that contains only the left fork. By contrast, the implementation with graph constraints hardly scales because the lack of a quantification operator requires to model the deadlock state explicitly for five, seven, and nine philosophers.

In addition, we define the following properties that help us assess that every philosopher will have a chance to eat:

- If a philosopher acquired a left fork, she will always acquire the right fork eventually, too.
- Every philosopher will eat regularly.

The implementation of these properties in cOCL and Groove is given in Table 6.5. Although both formulas check essentially for the same desired result state, i.e., either whether a philosopher is eating or whether a philosopher has acquired her left and her right fork, which entitles a philosopher to eat, the formulas are not equivalent. This is because the first checks that all those philosophers, who acquired a left fork, will eventually eat, i.e., hold their right fork, while it omits all those philosophers from the starvation check that never even acquire a left fork. This is not the case with the second formula. We further observe that the formulation in cOCL is more succinct than the

		Groove		MocOCL		Gryphon	
		Total	σ	Total	σ	Total	σ
SAME_FORK	5 Philosophers St: 573 Tr: 2365	3847	1224.65	1956	108.62	257	19.33
	7 Philosophers St: 7269 Tr: 42007	14474	5186.04	41919	1232.53	526	35.00
	9 Philosophers St: 92205 Tr: 685089	49832	15793.60	(TO)	(TO)	1004	75.98
WRONG_FORK	5 Philosophers St: 573 Tr: 2365	3868	1233.49	934	6.51	242	7.68
	7 Philosophers St: 7269 Tr: 42007	16233	5267.51	16747	254.55	467	9.37
	9 Philosophers St: 92205 Tr: 685089	50577	16011.30	(TO)	(TO)	921	77.21
DEADLOCK	5 Philosophers St: 573 Tr: 2365	3816	1216.41	754	24.33	766	61.20
	7 Philosophers St: 7269 Tr: 42007	15133	5129.01	13679	229.71	2621	231.79
	9 Philosophers St: 92205 Tr: 685089	49672	15758.90	(TO)	(TO)	7401	2960.36
ACQUIRE_RIGHT	5 Philosophers St: 573 Tr: 2365	3741	1193.87	353	2.98	—	—
	7 Philosophers St: 7269 Tr: 42007	14636	5463.19	12642	223.91	—	—
	9 Philosophers St: 92205 Tr: 1112401	49619	15775.10	64962	105.06	—	—
NO_STARVATION	5 Philosophers St: 573 Tr: 2365	3840.2	1216.72	1366	54.61	—	—
	7 Philosophers St: 7269 Tr: 42007	14227	5463.19	1983	141.96	—	—
	9 Philosophers St: 92205 Tr: 685089	49997	15853.70	35714	3346.73	—	—

Table 6.6: Dining philosopher benchmarks

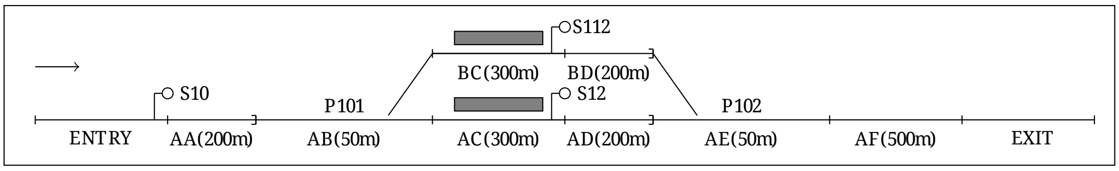
formulation with graph constraints, which require us to model each of desired conditions of the formula explicitly.

Benchmark Results

For the benchmarks we use initial models with five, seven and nine *thinking* philosophers, none of which holds a fork. Table 6.6 lists the results of the benchmarks. Cells that are marked — indicate that the specific verification condition could not be executed by the tool, while a timed out benchmark run is indicated by (TO).

6.3 Verification of Interlocking Railway Systems

The final benchmark targets the verification of interlocking railway systems and is inspired by the examples from [98]. A railway system is described by a *scheme plan* that consists of a *track plan*, a *control table*, and a set of *release tables* as depicted in Figure 6.9. The topology of the railway network is captured by the *track plan*, which displays tracks and their lengths, e.g., *AA* (200m), entry and exit tracks, signals, e.g.,



Control Table

Route	Normal	Reverse	Clear
R10A	P101		*AA*, *AB*, *AC*, *AD*
R10B		P101	*AA*, *AB*, *BC*, *BD*
R12	P102		*AD*, *AE*, *AF*
R112		P102	*BD*, *AE*, *AF*

Release tables

P101	Occupied
R10A	*AC*
R10B	*BC*

P102	Occupied
R12	*AF*
R112	*AF*

Figure 6.9: Station scheme plan [98]

signal S12 on track *AC*, and points, e.g., *P101*. A *route* consists of a set of tracks, the first of which may be entered if the guarding signal shows proceed. For example, route R10A consists of tracks *AA*, *AB*, *AC*, and *AD*; it may be entered if signal S10 on the entry track shows green. The conditions that change the *aspect* of a signal from red to green, thus allowing a train to pass and continue along the route, are provided by the *control table* (see Fig. 6.9). Each row in the control table is associated with a route and specifies the tracks that need to be *cleared* in order for a train to pass the signal guarding the route. If the route passes a point, the control table specifies the required *position* of the point. A point is locked either in *normal* position, leading the train straight ahead, or in *reverse* position, in which case the train is routed to another line. For example, route *R10A* is guarded by signal *S10*. In order for a train to travel along route *R10A*, point *P101* needs to be locked in normal position and tracks *AA*, *AB*, *AC*, and *AD* need to be cleared. On the other hand, if a train intends to travel along route *R10B*, also guarded by signal *S10*, point *P101* needs to be locked in reverse position and tracks *AA*, *AB*, *BC*, and *BD* need to be cleared.

A train is required to obtain a *lock* on a point prior to passing it and is required to release it after traversing the point. The *release table* that is associated with a point specifies the track, where a train must release the acquired lock. For example, a train traveling along route *R10A* is required to release the lock obtained for point *P101* upon reaching track *AC*. Note that a release table for a point always contains two release track entries, one for the route traveling over the point in normal position and one for the route traveling over the point in reverse position.

Following this setup, the authors of [98] model a railway system consisting of at least four components, the *controller*, the *interlocking*, the *track equipment*, and the *trains*. The controller is tasked with requesting and releasing routes, the interlocking monitors the track equipment and mediates between the controller and the trains, whose behavior is defined by the actions of the driver [98]. The verification of the railway system then centers around three safety properties:

- *collision freedom* prohibits two trains occupying the same track;
- *no-derailment* demands that a point does not change position while being occupied by a train;
- *run-through* requires a point to be set in position as specified by the control table for the specific route when a train is about to enter the point.

Modeling an Interlocking Railway System

In addition to the assumption that the train equipment operates correctly and reacts instantly [98], we introduce the following simplifications:[2]

S1 Train lengths are assumed to be shorter than the track lengths.

S2 Consecutive routes are joined together on their overlapping tracks thus producing routes that run from an entry to an exit track. For example, routes R10A and R12 are joined together to produce a single route *AA*, *AB*, *AC*, *AD*, *AE*, *AF*.

S3 A route is statically assigned to each train. This assignment does not change over the lifetime of the train.

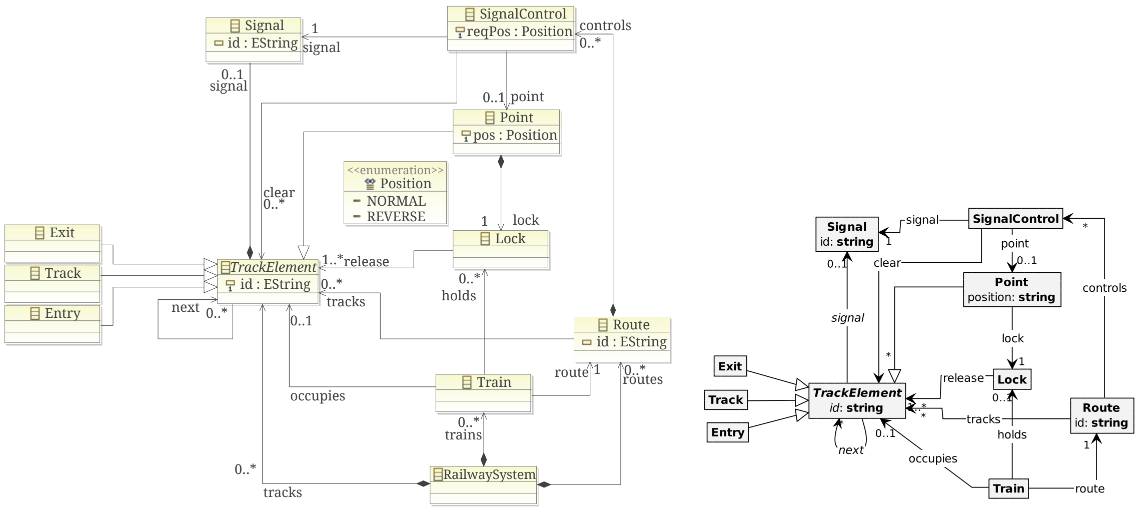
S4 Trains enter the railway network through entry tracks and leave through exit tracks. Multiple trains are allowed on both entry and exit tracks.

S5 A train, or its driver, may decide to either pass or halt in front of a green signal deliberately. A driver, however, may not overrun a stop signal onto the overlap track.

S6 All trains move at the same speed and track lengths are omitted from the implementation and analysis.

Due to simplifications S2 and S3 we essentially omit the controller from our representation of the railway system. The structures of our EMF and GROOVE implementations are shown in Figure 6.10. A `RailwaySystem` consists of set of routes, a set of trains, and a set of track elements. A train may occupy a track element and is assigned a fixed route. In case it must pass a point, a train can obtain a lock for that point. A `TrackElement` is either an ordinary `Track`, an `Entry` or an `Exit` track, or a `Point`. A track element may contain a signal and can have zero or more follow up tracks, given by the `next`

[2]Note that this simplifications are not due to limitations of our modeling approach and will be gradually eliminated to obtain a more faithful representation of the real-world scenario in the future.



(a) EMF (b) Groove

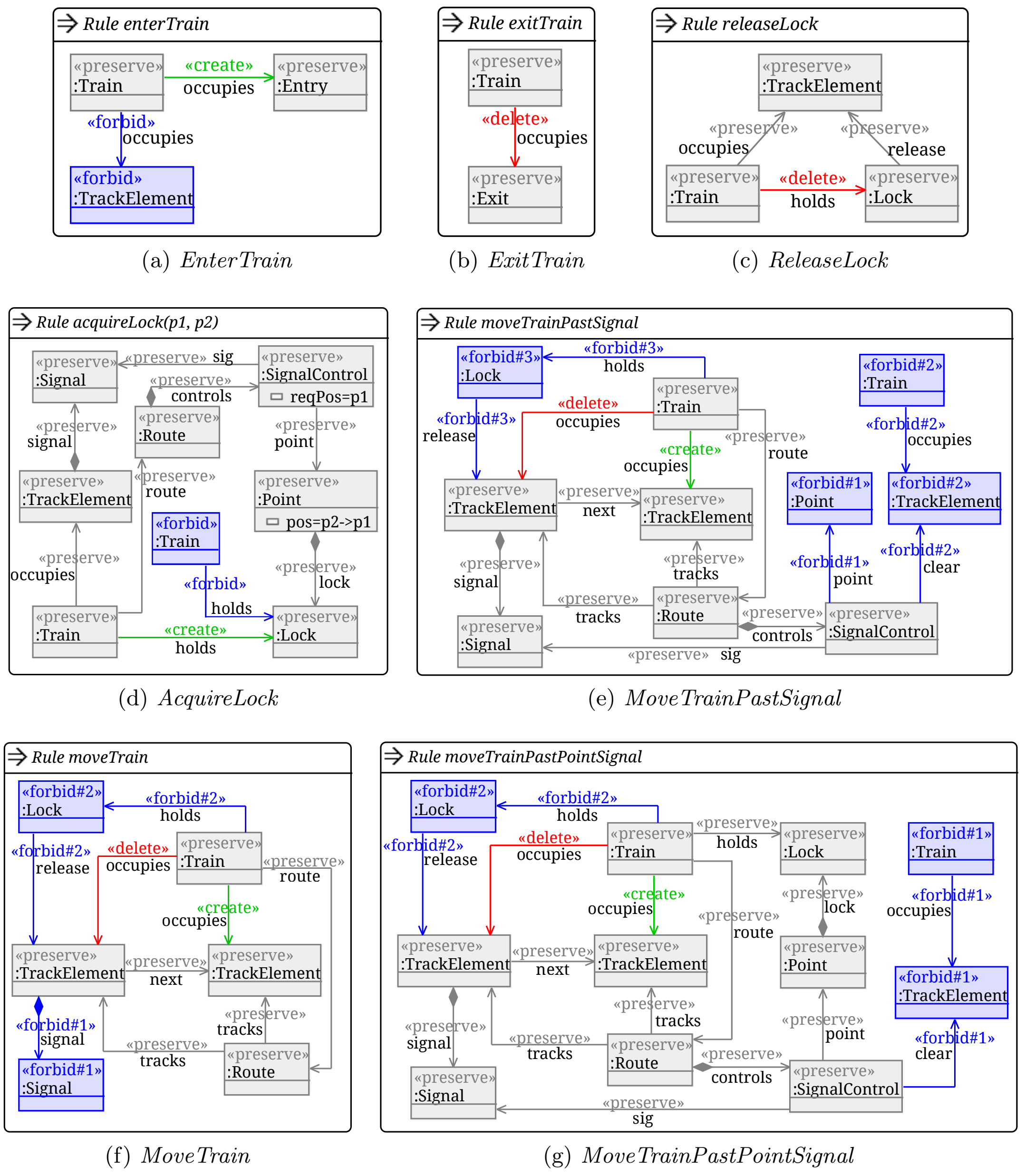
Figure 6.10: Railway System

reference. More specifically, an `Exit` track has no follow-up tracks, and all other track elements have exactly one follow-up track, except for points, which have two. Note that these restrictions are neither enforced by the model nor by OCL constraints, but are implemented as safety properties and verified with the model checker. As will become apparent from the definition of the graph transformations that model the behavior of the interlocking and the trains the validity of these constraints can be deduced statically provided that the initial model was correct. A `Route` consists of a set of tracks and for each signal along a route it contains a signal control. A `SignalControl` is thus associated with a signal and, if the signal guards a point, the signal links to this point and stores the position required for a train to pass the point to continue its route. Note that the implementation in Groove differs only in the type of the position (`pos`) attribute of the `Point` class, which is string valued in Groove and enumeration based in EMF.

A `RailwaySystem` is deemed consistent if it satisfies the following *consistency constraints*:

- Every route starts at an entry track and leads to an exit track.
- A route has exactly one signal control for each signal that is positioned at one of the route's tracks.
- An exit track has no `next` track set, an entry track and a (normal) track have exactly one `next` track set, and a point has exactly two `next` tracks set.

The behavior of the interlocking and the trains is implemented by the graph transformations depicted in Figure 6.11 and in Figure 6.12. The *EnterTrain* transformation places a train on an entry track if the train does not occupy a track element already.



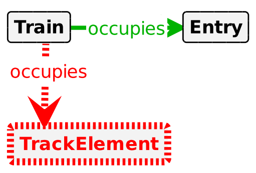
(a) *EnterTrain* (b) *ExitTrain* (c) *ReleaseLock*

(d) *AcquireLock* (e) *MoveTrainPastSignal*

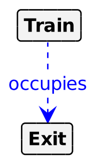
(f) *MoveTrain* (g) *MoveTrainPastPointSignal*

Figure 6.11: Behavior of the interlocking and train components modeled with Henshin

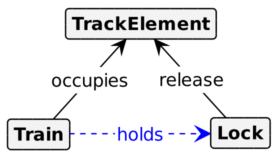
Similarly, the *ExitTrain* transformation removes a train from an exit track. If a train occupies a track (element) that contains a signal guarding a point, the train is required to acquire a lock on the point. In this case the *AcquireLock* transformations is applicable, and, if not taken by any other train, assigns the lock to the requesting train. Moreover, it changes the position of the point, for which the lock is to be acquired, according to



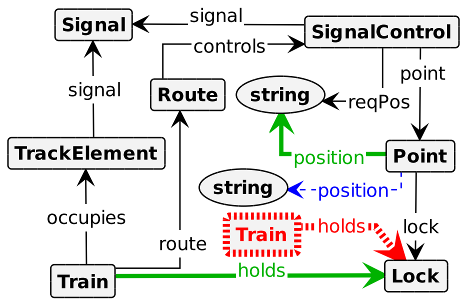
(a) *EnterTrain*



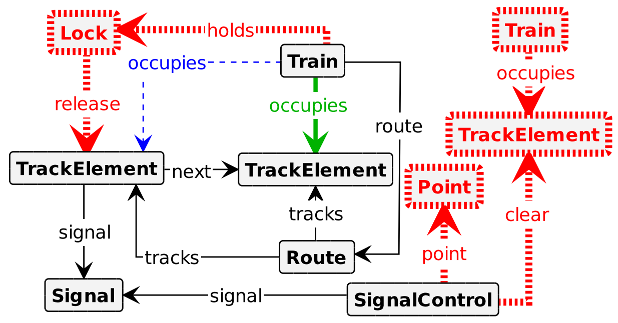
(b) *ExitTrain*



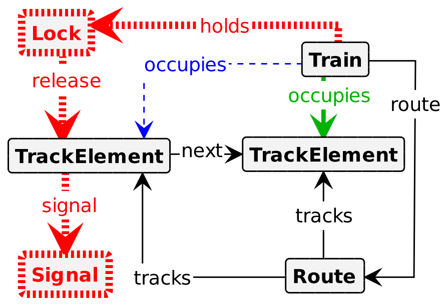
(c) *ReleaseLock*



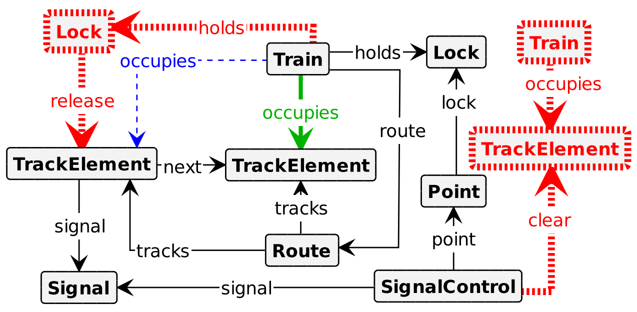
(d) *AcquireLock*



(e) *MoveTrainPastSignal*



(f) *MoveTrain*



(g) *MoveTrainPastPointSignal*

Figure 6.12: Behavior of the interlocking and train components modeled with GROOVE

the position requirement of the route (as defined by the signal control). Note that the HENSHIN implementation of the *AcquireLock* transformation (Fig. 6.11d) declares two parameters, `p1` and `p2`, which are bound to the values of the `reqPos` (required position) attribute of the matched `SignalControl` object and the `pos` (position) attribute of the matched `Point` object, respectively. The expression `p1->p2` rewrites the `pos` attribute to the value held by `p2`. Similarly, the GROOVE implementation of the *AcquireLock* transformation (Fig. 6.12d) deletes the current value of the `pos` attribute and rewrites it to the value held by the `reqPos` attribute by creating a reference to that value. Once the train has acquired the lock, it may pass the point and, upon reaching the `release` track, the lock is freed through the *ReleaseLock* transformation. The movement of trains through the railway network is implemented by three transformations, *MoveTrain*, *MoveTrain-PastSignal*, and *MoveTrainPastPointSignal*. A train may only move if it is not on a `release` track of a lock that the train holds. In case, no signal guards the tracks ahead of the train the train may move to the next track along the route. In all other cases, the conditions of the transformations *MoveTrainPastSignal* and *MoveTrainPastPointSignal* need to be satisfied. A train may move past a signal only if all guarded tracks are clear of other trains. If, in addition, the signal guards a point the train may only move past the signal if the required lock has been acquired. Note that all but the *EnterTrain* transformation (Fig. 6.11a and Fig. 6.12a) and the *AcquireLock* transformation (Fig. 6.11d and Fig. 6.12d) are matched injectively.

Finally, we define a safe *initial state* where every train is assigned a fixed route but does not occupy any track, all signals are set to red, each point is in normal position, and no locks are allocated [98]. The initial state as implemented in Groove is depicted in Figure 6.13. For the sake of readability we have grayed out all `tracks` edges leading from a route to its track elements.

Specification

Based on the three safety properties, *collision freedom*, *no derailment*, and *run through*, introduced at the beginning of this section we formulate the following specification.

- No two trains occupy the same point or track at the same time.
- Whenever a train occupies a point, the point is set in position for the train to continue its route accordingly.
- Whenever a point lies directly ahead of a train, the point is set in position for the train to continue its route accordingly.

The implementation of this specification is provided in Tables 6.7 and 6.8. The implementation of the first property, `COLLISION_FREE`, checks whether two different trains are located on the same track or point. While the cOCL expression uses type checks and casts to test for the type of the track element the train currently occupies, the graph constraints use two different patterns that test for the presence of the undesired state, i.e., a collision on a track or a point. For the `NO_DERAILMENT` property, the implementation in cOCL is again formulated positively, that is, it checks, provided that the train occupies a point, whether the position of the point coincides with the position required by the route as given by the signal control. The graph constraints, on the other hand, match the undesired state as their pattern searches for an instance of the railway system, where a train occupies an oppositely positioned point than required by the route. Finally, the `RUN_THROUGH` property checks whether a point is positioned such that the approaching train that occupies the track element just ahead of the point can pass, i.e., run through, and continue its route. Thus, the property's implementations are similar to those of the `NO_DERAILMENT` property with the only difference that the check of whether the point is positioned correctly is performed just before the train enters the point.

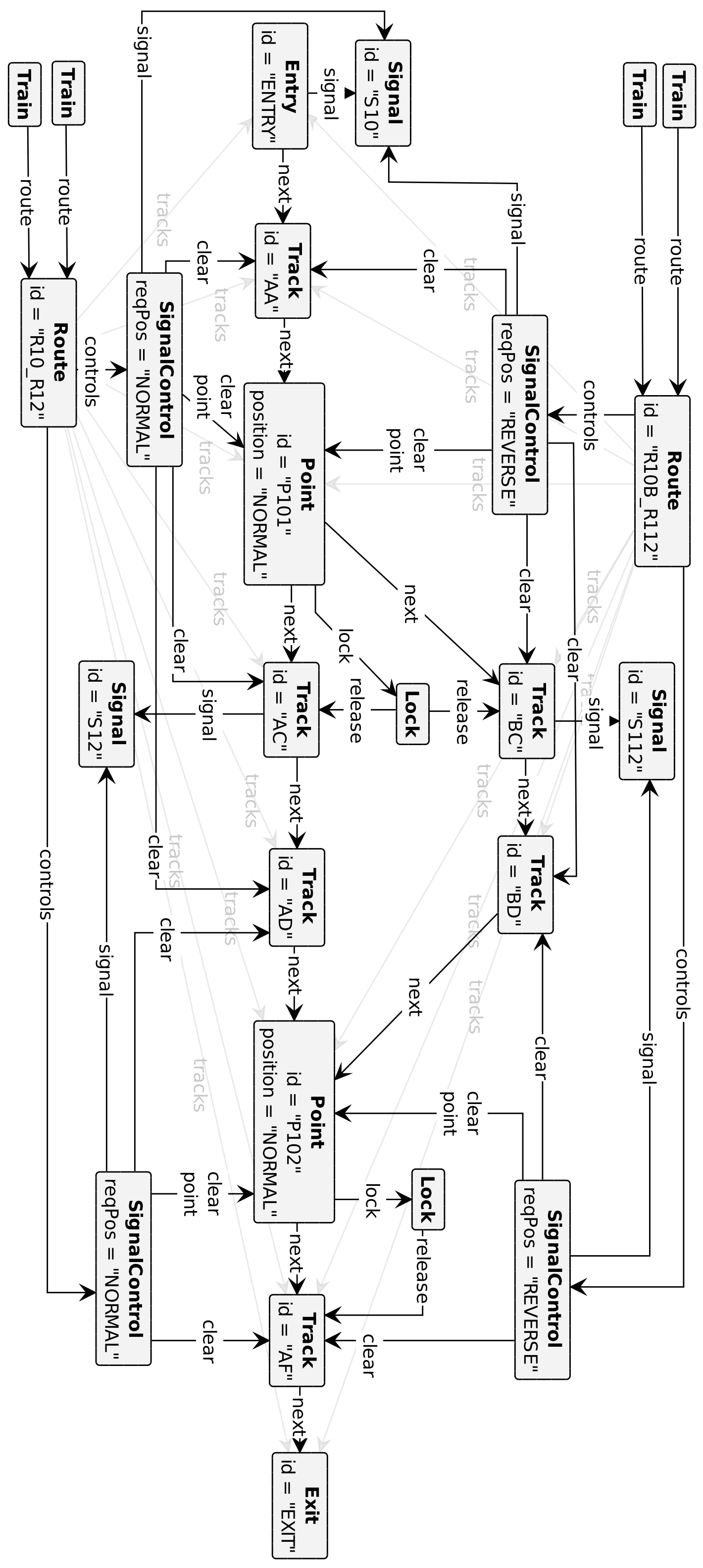
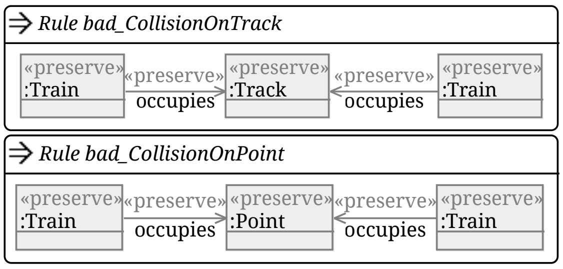
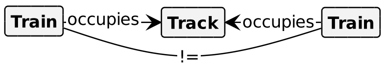
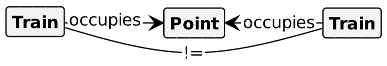


Figure 6.13: Initial state of the station scheme plan with four trains modeled with GROOVE

`COLLISION_FREE`: *No two trains occupy the same point or track at the same time.*	

cOCL	`Always Globally trains->forAll(t1,t2| t1<>t2 implies`
	`  ((t1.occupies.oclIsTypeOf(Track) or`
	`    t1.occupies.oclIsTypeOf(Point)) implies`
	`      t1.occupies <> t2.occupies))`

GRYPHON	 Rule bad_CollisionOnTrack «preserve» :Train «preserve» occupies «preserve» :Track «preserve» occupies «preserve» :Train Rule bad_CollisionOnPoint «preserve» :Train «preserve» occupies «preserve» :Point «preserve» occupies «preserve» :Train
GROOVE	`AG !(collisionOnTrack | collisionOnPoint)` where graph constraint `collisionOnTrack` and `collisionOnPoint` are defined as  and .
`NO_DERAILMENT`: *Whenever a train occupies a point, the point is set in position for the train to continue its route accordingly.*	

cOCL	`Always Globally trains->forAll(t|`
	` t.occupies.oclIsTypeOf(Point) implies`
	`  (let p=t.occupies.oclAsType(Point),`
	`       c=t.route.controls->select(c|c.point=p)->first() in`
	`    p.pos = c.reqPos))`

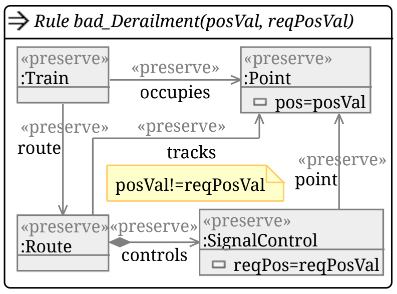
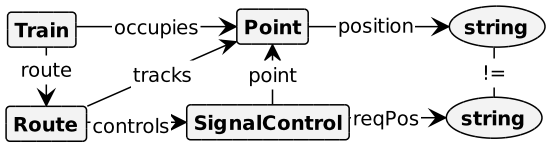
GRYPHON	 Rule bad_Derailment(posVal, reqPosVal) «preserve» :Train «preserve» occupies «preserve» :Point pos=posVal «preserve» route «preserve» tracks posVal!=reqPosVal «preserve» point «preserve» :Route «preserve» controls «preserve» :SignalControl reqPos=reqPosVal
GROOVE	`AG !derailment` where graph constraint `derailment` is defined as  Train occupies Point position string route tracks point != Route controls SignalControl reqPos string .

Table 6.7: The implementations of the `COLLISION_FREE` and the `NO_DERAILMENT` properties

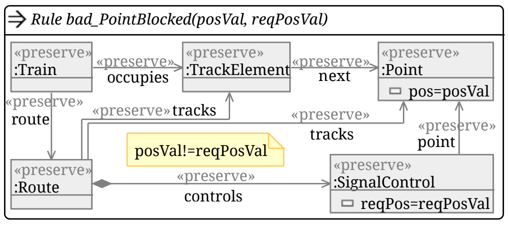
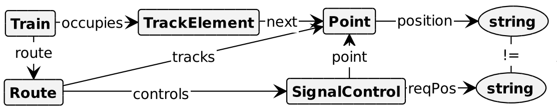
`RUN_THROUGH:` *Whenever a point lies directly ahead of a train, the point is set in position for the train to continue its route accordingly.*	
cOCL	`Always Globally trains->forAll(t|t.occupies<>null and t.occupies.next->first().oclIsTypeOf(Point) implies (let p=t.occupies.next->first().oclAsType(Point), c=t.route.controls->select(c|c.point=p)->first() in p.pos = c.reqPos))`
GRYPHON	 Rule bad_PointBlocked(posVal, reqPosVal) «preserve» :Train «preserve» occupies «preserve» :TrackElement «preserve» next «preserve» :Point pos=posVal «preserve» route «preserve» tracks «preserve» tracks «preserve» point posVal!=reqPosVal «preserve» :Route «preserve» controls «preserve» :SignalControl reqPos=reqPosVal
GROOVE	`AG !pointBlocked` where graph constraint `pointBlocked` is defined as  Train occupies TrackElement next Point position string route tracks point != Route controls SignalControl reqPos string .

Table 6.8: The implementation of the `RUN_THROUGH` property

Benchmark Results

For the performance evaluation we use the implementation of the station scheme plan that is depicted in Figure 6.9. We define four different initial states that instantiate the implementation with four, six, eight, and ten trains, and each route is assigned two, three, four, or five trains, respectively. The track plan of the station with four trains is depicted in Figure 6.13. Note that the benchmark did not complete correctly with MocOCL which, at the moment, cannot handle OCL's type checks (`oclIsTypeOf(...)`) and type casts (`oclAsType(...)`).

6.4 Summary

In this chapter we presented three benchmarks of varying complexity and compared the runtimes of our model checkers MocOCL and GRYPHON against GROOVE [106], a state-of-the-art modeling and verification tool for graph transformations. The intended contribution of this chapter is thus twofold. First, we provide an initial yet extensible set of benchmark problems that progress from structurally and behaviorally simple models to more complex, real-world inspired setups. We thus hope that others will find this initial set of benchmarks a useful starting point for their own comparisons and testing purposes. Second, we implement and discuss the implementation of each of the three benchmarks and compare the average runtimes of GROOVE, MocOCL, and GRYPHON.

In summary, we point out that, although all of the compared tools target the verifi-

		Groove		MocOCL		Gryphon	
		Total	σ	Total	σ	Total	σ
NO_COLLISION	4 Trains St: 2314 Tr: 7836	7079	3084.97	—	—	17280	1518.07
	6 Trains St: 9642 Tr: 46770	14282	4572.39	—	—	44894	2638.09
	8 Trains St: 27580 Tr: 174674	39358	12581.90	—	—	94725	7088.43
	10 Trains St: 63352 Tr: 495854	103639	32781.70	—	—	186877	16257.54
NO_DERAILMENT	4 Trains St: 2314 Tr: 7836	7427	3559.91	—	—	9778	507.26
	6 Trains St: 9642 Tr: 46770	14451	4623.55	—	—	19116	2091.32
	8 Trains St: 27580 Tr: 174674	39710	12578.50	—	—	33089	2959.80
	10 Trains St: 63352 Tr: 495854	103601	32788.70	—	—	57593	5798.69
RUN_THROUGH	4 Trains St: 2314 Tr: 7836	6254	2463.17	—	—	11472	780.74
	6 Trains St: 9642 Tr: 46770	14323	4636.48	—	—	21937	2562.94
	8 Trains St: 27580 Tr: 174674	39659	12667.40	—	—	37871	3043.84
	10 Trains St: 63352 Tr: 495854	104260	33008.30	—	—	68068	7207.89

Table 6.9: Railway system benchmarks

cation of graph transformation systems, each tool provides distinct features that might lead a developer to choose one of the tools over the others despite its running times. Clearly, GROOVE offers the most stable platform of the three tools and provides consistent verification and runtime results. Although it explores the entire state space it does not run into memory outages as MOCOCL does. On the other hand, MOCOCL was not intended to perform efficiently and optimizations have yet to be integrated. Its focus, however, lies on the verification of an expressive specification language. In contrast, GRYPHON's focus is scalability and speed at the expense of a reduced set of specifications that it may verify.

CHAPTER 7

Conclusion

In this dissertation I present techniques to assert the behavioral correctness of model-driven software development artifacts and introduce two novel model checkers, MocOCL and Gryphon. As such the dissertation comprises three contributions:

(i) an in-depth survey and a classification of existing, state-of-the-art verification approaches that verify the behavior of a set of model-driven software development artifacts against its specification,

(ii) an explicit state model checker, called MocOCL, that verifies expression written in cOCL, a novel CTL based extension of OCL, against an implementation whose static structure is described by type graphs and whose behavior is defined by graph transformations, and, third,

(iii) a new symbolic model checker, called Gryphon, that encodes a model based software implementation consisting of a type graph and a set of graph transformations into bounded, first-order relational logic to assert the unreachability of a bad state.

Our survey encompasses more than forty different verification approaches that we classify along five dimensions: *verification goal*, *domain representation*, *verification representation*, *specification language*, and *verification technique*. With this classification we hope to assist practitioner to select an adequate verification approach for their verification problem and, on the other hand, identify viable inputs and directions for future research by identifying white spots in the research landscape.

With MocOCL we presented a novel approach to verify rich, temporal specifications directly at the level of models. For this purpose we present formal syntax and semantics of cOCL, a CTL-extension for OCL, that allows to formulate temporal OCL specifications. These cOCL specifications can be verified with MocOCL against a system whose static structure is defined attributed, type graphs with inheritance and containment relations and whose behavior is captured by a set of graph transformations. It performs an iterative exploration of the state space, that it stores explicitly, starting from the user-provided initial state and verifies the cOCL specification on-the-fly; once the state space has been explored up to the point that the specification is either found to hold or not MocOCL reports the cause of the result on demand.

Finally, in an effort to raise the scalability of our verification technique work on a symbolic model checker, called Gryphon, was initiated with the aim to efficiently verify properties of the form $\mathsf{AG}\,\neg\phi$ where ϕ denotes a bad state of the system. Similar to MocOCL, our symbolic model checker receives as input an attributed, type graph with inheritance and containment relations and a set of graph transformations that describe structure and behavior of the system, respectively. The verification requires, in addition, an initial state and a set of upper bounds on the maximal number of objects per class to ensure that the resulting state space is finite. The specification, i.e. the bad state of the system, is defined in terms of graph constraint, i.e., a graph transformation with identical left-hand and right-hand side. The inputs are first encoded into bounded, first-order relational logic and subsequently in a series of three steps converted into the AIGER format, which is the standardized input format of the Hardware Model Checking Competition. Thus, Gryphon is able to directly exploit advances in industrial grade model checkers.

Future Work

In the following I list exemplarily two possible future extensions for the above summarized contributions.

Classification and Future Trends. The world of verification techniques is neither black nor white and it is sometimes not readily apparent whether a verification approach that incorporates an automatic verification engine like an SMT solver qualifies as model checking based approach or a theorem proving based approach or both. Currently, the classification presented in Chapter 3.2 cannot adequately categorize such verification approaches. In the future we expect, however, that verification approaches for software, and model-driven software development artifacts in particular, will integrate different verification techniques, that were formerly developed in isolation of each other. In order to reflect this change in the classification, we intend to adopt it to distinguish in the future between *automatic*, *semi-automatic*, and *interactive* verification techniques.

Incremental Evaluation of OCL Expressions. Currently, MocOCL checks subexpression of a cOCL expression in every state of the system regardless whether parts relevant to the checked expression have changed or not. Given that the information on what has changed is readily derivable from the source graph and the applied graph transformation the re-evaluation of a cOCL expression could be optimized in this respect. Work on the incremental evaluation of OCL expression has been previously presented in [16, 39, 76].

Bibliography

[1] Lukman Ab. Rahim and Jon Whittle. A survey of approaches for verifying model transformations. *Software and System Modeling*, pages 1–26, 2013.

[2] Bowen Alpern and Fred B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2(3):117–126, 1987.

[3] Moussa Amrani, Jürgen Dingel, Leen Lambers, Levi Lúcio, Rick Salay, Gehan Selim, Eugene Syriani, and Manuel Wimmer. Towards a model transformation intent catalog. In *Proceedings of the First Workshop on the Analysis of Model Transformations*, AMT '12, pages 3–8, New York, NY, USA, 2012. ACM.

[4] Moussa Amrani, Levi Lúcio, Gehan M. K. Selim, Benoît Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *Proceedings of the Fifth International Conference on Software Testing, Verification, and Validation*, pages 921–928, Washington, DC, USA, 2012. IEEE Computer Society.

[5] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A Challenging Model Transformation. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 436–450, Heidelberg, Germany, 2007. Springer.

[6] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *LNCS*, pages 121–135, Heidelberg, Germany, 2010. Springer.

[7] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, Cambridge, MA, USA, 2008.

[9] Daniel Balasubramanian, Corina Pasareanu, Gabor Karsai, and Michael Lowry. Polyglot: Systematic Analysis for Multiple Statechart Formalisms. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *LNCS*, pages 523–529, Heidelberg, Germany, 2013. Springer.

[10] Paolo Baldan, Andrea Corradini, and Barbara König. A Static Analysis Technique for Graph Transformation Systems. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR 2001 — Concurrency Theory*, volume 2154 of *LNCS*, pages 381–395, Heidelberg, Germany, 2001. Springer.

[11] Luciano Baresi, Vahid Rafe, Adel Torkaman Rahmani, and Paola Spoletini. An efficient solution for model checking graph transformation systems. *Electr. Notes Theor. Comput. Sci.*, 213(1):3–21, 2008.

[12] Luciano Baresi and Paola Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 4178 of *LNCS*, pages 306–320, Heidelberg, Germany, 2006. Springer.

[13] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *LNCS*, pages 171–177, Heidelberg, Germany, 2011. Springer.

[14] Mordechai Ben-Ari. *Principles of the Spin Model Checker.* Springer, London, UK, 2008.

[15] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification: model-checking techniques and tools.* Springer, Heidelberg, Germany, 2001.

[16] Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Integrating Efficient Model Queries in State-of-the-Art EMF Tools. In Carlo A. Furia and Sebastian Nanz, editors, *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*, volume 7304 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 2012.

[17] Marc Bezem, Jan Willem Klop, and Roel de Vrijer, editors. *Term rewriting systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, New York, NY, USA, 2003.

[18] Vieri Del Bianco, Luigi Lavazza, and Marco Mauri. Model Checking UML Specifications of Real Time Software. In *Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, pages 203–212, Washington, DC, USA, 2002. IEEE Computer Society.

[19] Armin Biere. The AIGER And-Inverter Graph (AIG) Format Version 20071012. `http://fmv.jku.at/aiger/FORMAT`, 2007.

[20] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207, Heidelberg, Germany, 1999. Springer.

[21] Armin Biere and Keijo Heljanko. Hardware Model Checking Competition 2014 CAV Edition. `http://fmv.jku.at/hwmcc14cav/hwmcc14olympics.pdf`, 2014.

[22] Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and Beyond. `http://fmv.jku.at/hwmcc11/beyond1.pdf`, July 2011.

[23] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Lifting Parallel Graph Transformation Concepts to Model Transformation based on the Eclipse Modeling Framework. *ECEASST*, 26, 2010.

[24] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software and System Modeling*, 11(2):227–250, 2012.

[25] Robert Bill. Towards Software Model Checking in the Context of Model-Driven Engineering. Master's thesis, E188 Institut für Softwaretechnik und Interaktive Systeme, 2014. Supervisor: G. Kappel, P. Kaufmann, S. Gabmeyer.

[26] Robert Bill, Sebastian Gabmeyer, Petra Brosch, and Martina Seidl. MocOCL: A Model Checker for CTL-Extended OCL Specifications. In *Proceedings of the STAF Workshop on Verification of Model Transformations (VOLT 2014)*, 2014. Available at: `http://volt2014.big.tuwien.ac.at/papers/volt2014_paper_4.pdf`.

[27] Robert Bill, Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. OCL meets CTL: Towards CTL-Extended OCL Model Checking. In Jordi Cabot, Martin Gogolla, István Ráth, and Edward D. Willink, editors, *Proceedings of the MODELS 2013 OCL Workshop*, volume 1092 of *CEUR Workshop Proceedings*, pages 13–22, Aachen, Germany, 2013. CEUR-WS.org.

[28] Robert Bill, Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. Model checking of ctl-extended OCL specifications. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *LNCS*, pages 221–240, Heidelberg, Germany, 2014. Springer.

[29] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, Heidelberg, Germany, 2003.

[30] Artur Boronat, Reiko Heckel, and José Meseguer. Rewriting Logic Semantics and Verification of Model Transformations. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering*, volume 5503 of *LNCS*, pages 18–33, Heidelberg, Germany, 2009. Springer.

[31] Artur Boronat and José Meseguer. Algebraic Semantics of OCL-Constrained Metamodel Specifications. In Manuel Oriol and Bertrand Meyer, editors, *Objects, Components, Models and Patterns*, volume 33 of *Lecture Notes in Business Information Processing*, pages 96–115, Heidelberg, Germany, 2009. Springer.

[32] Artur Boronat and José Meseguer. An algebraic semantics for MOF. *Formal Asp. Comput.*, 22(3-4):269–296, 2010.

[33] Julian C. Bradfield, Juliana Küster Filipe, and Perdita Stevens. Enriching OCL Using Observational Mu-Calculus. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 203–217, Heidelberg, Germany, 2002. Springer.

[34] Julian C. Bradfield and Perdita Stevens. *Observational mu-calculus.* BRICS Report Series. BRICS, Department of Computer Science, Univ. of Aarhus, Aarhus, Denmark, 1999.

[35] Achim D. Brucker, Frédéric Tuong, and Burkhart Wolff. Featherweight OCL: A Proposal for a Machine-Checked Formal Semantics for OCL 2.5. *Archive of Formal Proofs*, 2014, 2014.

[36] Achim D. Brucker and Burkhart Wolff. Featherweight OCL: a study for the consistent semantics of OCL 2.3 in HOL. In Mira Balaban, Jordi Cabot, Martin Gogolla, and Claas Wilke, editors, *Proceedings of the 12th Workshop on OCL and Textual Modelling, Innsbruck, Austria, September 30, 2012*, pages 19–24. ACM, 2012.

[37] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[38] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. Verification of ATL Transformations Using Transformation Models and Model Finders. In Toshiaki Aoki and Kenji Taguchi, editors, *Formal Methods and Software Engineering*, volume 7635 of *LNCS*, pages 198–213, Heidelberg, Germany, 2012. Springer.

[39] Jordi Cabot and Ernest Teniente. Incremental integrity checking of UML/OCL conceptual schemas. *Journal of Systems and Software*, 82(9):1459–1478, 2009.

[40] Daniel Calegari and Nora Szasz. Verification of Model Transformations. *Electr. Notes Theor. Comput. Sci.*, 292:5–25, March 2013.

[41] Edward Chang, Zohar Manna, and Amir Pnueli. The Safety-Progress Classification. In FriedrichL. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors,

Logic and Algebra of Specification, volume 94 of *NATO ASI Series*, pages 143–202. Springer, Heidelberg, Germany, 1993.

[42] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *LNCS*, pages 52–71, Heidelberg, Germany, 1981. Springer.

[43] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[44] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 168–176, Heidelberg, Germany, 2004. Springer.

[45] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*, Heidelberg, Germany, 2007. Springer.

[46] The Coq Proof Assistant. `http://coq.inria.fr/`, 2012. Accessed 2015-05-20.

[47] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach. In Rozenberg [173], pages 163–246.

[48] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[49] Cve.mitre.org. CVE-2014-0160. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`, April 2014.

[50] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[51] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340, Heidelberg, Germany, 2008. Springer.

[52] Edsger W. Dijkstra. Cooperating sequential processes, ewd 123. `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html`.

[53] Edsger W. Dijkstra. The Humble Programmer. *Commun. ACM*, 15(10):859–866, 1972.

[54] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, August 1975.

[55] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. On a Temporal Logic for Object-Based Systems. In Scott F. Smith and Carolyn L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV*, volume 49 of *IFIP Advances in Information and Communication Technology*, pages 305–325, New York, NY, USA, 2000. Springer US.

[56] Wei Dong, Ji Wang, Xuan Qi, and Zhichang Qi. Model Checking UML Statecharts. In *Proceedings of the Eighth Asia-Pacific Software Engineering Conference*, pages 363–370, Washington, DC, USA, 2001. IEEE Computer Society.

[57] Jori Dubrovin and Tommi A. Junttila. Symbolic model checking of hierarchical uml state machines. In Jonathan Billington, Zhenhua Duan, and Maciej Koutny, editors, *Proceedings of the Eighth International Conference on Application of Concurrency to System Design*, pages 108–117, Washington, DC, USA, 2008. IEEE.

[58] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.

[59] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 21st International Conference on Software Engineering*, pages 411–420, New York, NY, USA, 1999. ACM.

[60] Eclipse.org. Package org.eclipse.emf.ecore JavaDoc. `http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/index.html?org/eclipse/emf/ecore/package-summary.html`. Retrieved 2015-05-20.

[61] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, pages 502–518, Heidelberg, Germany, 2003. Springer.

[62] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[63] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Graph Transformations - 6th International Conference, ICGT 2012, Bremen,*

Germany, September 24-29, 2012. Proceedings, volume 7562 of *LNCS*, Heidelberg, Germany, 2012. Springer.

[64] Hartmut Ehrig and Claudia Ermel. Semantical Correctness and Completeness of Model Transformations Using Graph and Rule Transformation. In Ehrig et al. [65], pages 194–210.

[65] Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors. *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, volume 5214 of *LNCS*, Heidelberg, Germany, 2008. Springer.

[66] Hartmut Ehrig and Barbara König. Deriving bisimulation congruences in the dpo approach to graph rewriting with borrowed contexts. *Mathematical Structures in Computer Science*, 16(6):1133–1163, 2006.

[67] E. Allen Emerson and Edmund M. Clarke. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *LNCS*, pages 169–181, Heidelberg, Germany, 1980. Springer.

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

[69] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 new unclarities in the semantics of UML 2.0 state machines. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2005.

[70] Stephan Flake and Wolfgang Müller. Formal semantics of static and temporal state-oriented OCL constraints. *Software and System Modeling*, 2(3):164–186, 2003.

[71] Sebastian Gabmeyer. Quality assurance in MBE back and forth. In Seidl and Tillmann [179], pages 78–81.

[72] Sebastian Gabmeyer, Petra Brosch, and Martina Seidl. A Classification of Model Checking-Based Verification Approaches for Software Models. In *Proceedings of the STAF Workshop on Verification of Model Transformations (VOLT 2013)*, pages 1–7, 2013.

[73] Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. A feature-based classification of formal verification techniques for software models. Technical Report BIG-TR-2014-1, Institut für Softwaretechnik und Interaktive Systeme; Technische Universität Wien, 2014.

[74] Patrice Gagnon, Farid Mokhati, and Mourad Badri. Applying Model Checking to Concurrent UML Models. *Journal of Object Technology*, 7(1):59–84, 2008.

[75] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: a toolbox for the construction and analysis of distributed processes. *STTT*, 15(2):89–107, 2013.

[76] Miguel García and Ralf Möller. Incremental Evaluation of OCL Invariants in the Essential MOF Object Model. In Thomas Kühne, Wolfgang Reisig, and Friedrich Steimann, editors, *Modellierung 2008, 12.-14. März 2008, Berlin*, volume 127 of *LNI*, pages 11–26. GI, 2008.

[77] Holger Giese and Leen Lambers. Towards Automatic Verification of Behavior Preservation for Model Transformation via Invariant Checking. In Ehrig et al. [63], pages 249–263.

[78] Stefania Gnesi, Diego Latella, and Mieke Massink. Model Checking UML Statechart Diagrams Using JACK. In *Proceeding of the Fourth IEEE International Symposium on High-Assurance Systems Engineering*, pages 46–55, Washington, DC, USA, 1999. IEEE Computer Society.

[79] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69(1-3):27–34, 2007.

[80] Martin Gogolla, Lars Hamann, and Frank Hilken. Checking Transformation Model Properties with a UML and OCL Model Validator. In Moussa Amrani, Eugene Syriani, and Manuel Wimmer, editors, *Proceedings of the Third International Workshop on Verification of Model Transformations co-located with Software Technologies: Applications and Foundations, VOLT@STAF 2014, York, UK, July 21, 2014.*, volume 1325 of *CEUR Workshop Proceedings*, pages 16–25. CEUR-WS.org, 2014.

[81] Martin Gogolla, Lars Hamann, Frank Hilken, Mirco Kuhlmann, and Robert B. France. From application models to filmstrip models: An approach to automatic validation of model dynamics. In Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer, editors, *Modellierung 2014, 19.-21. März 2014, Wien, Österreich*, volume 225 of *LNI*, pages 273–288. GI, 2014.

[82] Orna Grumberg, Yael Meller, and Karen Yorav. Applying Software Model Checking Techniques for Behavioral UML Models. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, volume 7436 of *LNCS*, pages 277–292, Heidelberg, Germany, 2012. Springer.

[83] Annegret Habel and Detlef Plump. Relabelling in Graph Transformation. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg,

editors, *Graph Transformation*, volume 2505 of *LNCS*, pages 135–147, Heidelberg, Germany, 2002. Springer.

[84] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic.* MIT Press, Cambridge, MA, USA, 2000.

[85] David Harel, Amir Pnueli, Jeanette P. Schmidt, and Rivi Sherman. On the Formal Semantics of Statecharts (Extended Abstract). In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*, pages 54–64. IEEE Computer Society, 1987.

[86] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16, 2012.

[87] Reiko Heckel. Compositional Verification of Reactive Systems Specified by Graph Transformation. In Egidio Astesiano, editor, *Fundamental Approaches to Software Engineering*, volume 1382 of *LNCS*, pages 138–153, Heidelberg, Germany, 1998. Springer.

[88] Frank Hermann, Mathias Hülsbusch, and Barbara König. Specification and Verification of Model Transformations. *ECEASST*, 30:20, 2010.

[89] Frank Hilken, Lars Hamann, and Martin Gogolla. Transformation of UML and OCL Models into Filmstrip Models. In Davide Di Ruscio and Dániel Varró, editors, *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*, volume 8568 of *Lecture Notes in Computer Science*, pages 170–185. Springer, 2014.

[90] Frank Hilken, Philipp Niemann, Martin Gogolla, and Robert Wille. Filmstripping and Unrolling: A comparison of Verification Approaches for UML and OCL Behavioral Models. In Seidl and Tillmann [179], pages 99–116.

[91] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[92] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Upper Saddle River, NJ, USA, 1985.

[93] Mathias Hülsbusch, Barbara König, Arend Rensink, Maria Semenyak, Christian Soltenborn, and Heike Wehrheim. Showing Full Semantics Preservation in Model Transformation - A Comparison of Techniques. In Dominique Méry and Stephan Merz, editors, *Integrated Formal Methods*, volume 6396 of *LNCS*, pages 183–198, Heidelberg, Germany, 2010. Springer.

[94] Mathias Hülsbusch, Barbara König, Arend Rensink, Maria Semenyak, Christian Soltenborn, and Heike Wehrheim. Showing Full Semantics Preservation in Model

Transformation - A Comparison of Techniques. Technical Report TR-CTIT-10-09, Centre for Telematics and Information Technology, University of Twente, 2012.

[95] Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 130–139, New York, NY, USA, 2000. ACM.

[96] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

[97] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, Cambridge, MA, USA, Rev. edition, 2012.

[98] Phillip James, Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve A. Schneider, and Helen Treharne. On modelling and verifying railway interlockings: Tracking train lengths. *Sci. Comput. Program.*, 96:315–336, 2014.

[99] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[100] Lixia Ji, Jianhong Ma, and Zhuowei Shan. Research on Model Checking Technology of UML. In *2012 International Conference on Computer Science Service System (CSSS)*, pages 2337–2340, Washington, DC, USA, 2012. IEEE.

[101] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 128–138, Heidelberg, Germany, 2005. Springer.

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

[103] Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala Latvala, and Ivan Porres. Model Checking Dynamic and Hierarchical UML State Machines. In Kühne [115], page 15.

[104] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, November 1990.

[105] Bilal Kanso and Safouan Taha. Temporal Constraint Support for OCL. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering*, volume 7745 of *LNCS*, pages 83–103, Heidelberg, Germany, 2012. Springer.

[106] Harmen Kastenberg and Arend Rensink. Model Checking Dynamic States in GROOVE. In Antti Valmari, editor, *Model Checking Software*, volume 3925 of *LNCS*, pages 299–305, Heidelberg, Germany, 2006. Springer.

[107] The KIV system, October 2012.

[108] Anneke Kleppe. *Software Language Engineering*. Addison-Wesley Professional, 1 edition, December 2008.

[109] Alexander Knapp and Jochen Wuttke. Model Checking of UML 2.0 Interactions. In Kühne [115], pages 42–51.

[110] Barbara König and Vitali Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 197–211, Heidelberg, Germany, 2006. Springer.

[111] Barbara König and Vitali Kozioura. Augur 2 - A New Version of a Tool for the Analysis of Graph Transformation Systems. *Electr. Notes Theor. Comput. Sci.*, 211:201–210, 2008.

[112] Barbara König and Vitali Kozioura. Towards the Verification of Attributed Graph Transformation Systems. In Ehrig et al. [65], pages 305–320.

[113] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *LNCS*, pages 1–35, Heidelberg, Germany, 2013. Springer.

[114] Dexter Kozen. Results on the Propositional mu-Calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[115] Thomas Kühne, editor. *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, volume 4364 of *LNCS*, Heidelberg, Germany, 2007. Springer.

[116] Marcel Kyas, Harald Fecher, Frank S. de Boer, Joost Jacob, Jozef Hooman, Mark van der Zwaag, Tamarah Arons, and Hillel Kugler. Formalizing UML Models and OCL Constraints in PVS. *Electr. Notes Theor. Comput. Sci.*, 115:39–47, 2005.

[117] Mustafa Al Lail, Ramadan Abdunabi, Robert France, and Indrakshi Ray. An Approach to Analyzing Temporal Properties in UML Class Models. In Frédéric Boulanger, Michalis Famelis, and Daniel Ratiu, editors, *Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa 2013)*, volume 1069 of *CEUR Workshop Proceedings*, pages 77–86, Aachen, Germany, 2013. CEUR-WS.org.

[118] Vitus S. W. Lam and Julian A. Padget. Symbolic Model Checking of UML Statechart Diagrams with an Integrated Approach. In *Proceedings of the 11th IEEE International Conference on the Engineering of Computer-Based Systems*, pages 337–347, Washington, DC, USA, 2004. IEEE Computer Society.

[119] Daniel Leivant. Higher order logic. In Dov M. Gabbay, Christopher J. Hogger, J. A. Robinson, and Jörg H. Siekmann, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming (2)*, pages 229–322. Oxford University Press, Oxford, UK, 1994.

[120] Nancy Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[121] Johan Lilius and Iván Porres Paltor. vUML: A Tool for Verifying UML Models. In *14th IEEE International Conference on Automated Software Engineering*, pages 255–258, Washington, DC, USA, 1999. IEEE Computer Society.

[122] 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. *Electr. Notes Theor. Comput. Sci.*, 254:143–160, 2009.

[123] Jaques-Louis Lion, Lennart Lübeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedagg, Leonardo Mazzini, Didier Merle, and Colin O'Hallorn. Ariane 5 Flight 501 Failure, Report by the Inquiry Board. `http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`, July 1996.

[124] Delphine Longuet, Frédéric Tuong, and Burkhart Wolff. Towards a Tool for Featherweight OCL: A Case Study On Semantic Reflection. In Achim D. Brucker, Carolina Dania, Geri Georg, and Martin Gogolla, editors, *Proceedings of the 14th International Workshop on OCL and Textual Modelling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 30, 2014.*, volume 1285 of *CEUR Workshop Proceedings*, pages 43–52. CEUR-WS.org, 2014.

[125] Tim A. Majchrzak. *Improving Software Testing, Technical and Organizational Developments.* SpringerBriefs in Information Systems. Springer-Verlag Berlin Heidelberg, Heidelberg, Germany, 1 edition, 2012.

[126] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification.* Springer, Heidelberg, Germany, 1992.

[127] Greg Manning and Detlef Plump. The GP Programming System. *ECEASST*, 10:13, 2008.

[128] D. L. McBurney and M. Ronan Sleep. Graph Rewriting as a Computational Model. In Akinori Yonezawa and Takayasu Ito, editors, *Concurrency: Theory, Language, And Architecture, UK/Japan Workshop, Oxford, UK, September 25-27, 1989, Proceedings*, volume 491 of *Lecture Notes in Computer Science*, pages 235–256. Springer, 1989.

[129] John McCarthy and Peter J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In Matthew L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 26–45. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[130] Kenneth L. McMillan. *Symbolic model checking.* Kluwer Academic Publishers, Dordrecht, the Netherlands, 1993.

[131] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[132] José Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *LNCS*, pages 18–61, Heidelberg, Germany, 1997. Springer.

[133] José Meseguer. Twenty years of rewriting logic. *Formal Asp. Comput.*, 81(7–8):721–781, 2012.

[134] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. Implementing Statecharts in PROMELA/SPIN. In *Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, pages 90–101, Washington, DC, USA, 1998. IEEE Computer Society.

[135] Robin Milner. *Communication and Concurrency.* PHI Series in computer science. Prentice Hall, Upper Saddle River, NJ, USA, 1989.

[136] Maryam Mozaffari and Ali Harounabadi. Verification and validation of UML 2.0 sequence diagrams using colored Petri nets. In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pages 117–121, Washington, DC, USA, 2011. IEEE.

[137] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science.* Springer, 2002.

[138] John Mullins and Raveca Oarga. Model Checking of Extended OCL Constraints on UML Models in SOCLe. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *LNCS*, pages 59–75, Heidelberg, Germany, 2007. Springer.

[139] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[140] Anantha Narayanan and Gabor Karsai. Towards Verifying Model Transformations. *Electr. Notes Theor. Comput. Sci.*, 211:191–200, 2008.

[141] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO.* NATO, 1969.

[142] Thomas R. Nicely. Pentium FDIV flaw. `http://www.trnicely.net/pentbug/pentbug.html`, August 2011.

[143] Artur Niewiadomski, Wojciech Penczek, and Maciej Szreter. A New Approach to Model Checking of UML State Machines. *Fundam. Inform.*, 93(1-3):289–303, 2009.

[144] Artur Niewiadomski, Wojciech Penczek, and Maciej Szreter. Towards Checking Parametric Reachability for UML State Machines. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 319–330, Heidelberg, Germany, 2009. Springer.

[145] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, Heidelberg, Germany, 2002.

[146] Iulian Ober, Susanne Graf, and Ileana Ober. Validation of UML Models via a Mapping to Communicating Extended Timed Automata. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software*, volume 2989 of *LNCS*, pages 127–145, Heidelberg, Germany, 2004. Springer.

[147] Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *International Journal on Software Tools for Technology Transfer*, 8(2):128–145, 2006.

[148] Object Management Group OMG. Object Constraint Language (OCL) V2.2. `http://www.omg.org/spec/OCL/2.2/`, February 2010.

[149] Object Management Group OMG. OMG Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification V1.1. `http://www.omg.org/spec/QVT/1.1/`, January 2011.

[150] Object Management Group OMG. OMG Meta Object Facility (MOF) Core Specification V2.4.1. `http://www.omg.org/spec/MOF/2.4.1/`, August 2011.

[151] Object Management Group OMG. OMG MOF 2 XMI Mapping Specification V2.4.1. `http://www.omg.org/spec/XMI/2.4.1`, August 2011.

[152] Object Management Group OMG. OMG Unified Modeling Language (OMG UML), Infrastructure V2.4.1. `http://www.omg.org/spec/UML/2.4.1/`, August 2011.

[153] Object Management Group OMG. OMG Unified Modeling Language (OMG UML), Superstructure V2.4.1. `http://www.omg.org/spec/UML/2.4.1/`, August 2011.

[154] Object Management Group OMG. Model Driven Architecture (MDA) Guide rev. 2.0. `http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf`, June 2014.

[155] Mouna Ait Oubelli, Nadia Younsi, Abdelkrim Amirat, and Ahcene Menasria. From UML 2.0 Sequence Diagrams to PROMELA code by Graph Transformation using AToM3. In Abdelmalek Amine, Otmane Aït Mohamed, Boualem Benatallah, and Zakaria Elberrichi, editors, *Proceedings of the Third International Conference on Computer Science and its Applications*, volume 825 of *CEUR Workshop Proceedings*, Aachen, Germany, 2011. CEUR-WS.org.

[156] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *LNCS*, pages 748–752, Heidelberg, Germany, 1992. Springer.

[157] Carl Adam Petri and Wolfgang Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008.

[158] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[159] Iman Poernomo and Jeffrey Terrell. Correct-by-Construction Model Transformations from Partially Ordered Specifications in Coq. In Jin Song Dong and Huibiao Zhu, editors, *Formal Methods and Software Engineering*, volume 6447 of *LNCS*, pages 56–73, Heidelberg, Germany, 2010. Springer.

[160] Ivan Porres. Modeling and Analyzing Software Behavior in UML, 2001.

[161] Christopher M. Poskitt and Detlef Plump. Hoare-Style Verification of Graph Programs. *Fundam. Inform.*, 118(1-2):135–175, 2012.

[162] Christopher M. Poskitt and Detlef Plump. Verifying Total Correctness of Graph Programs. *ECEASST*, 61:20, 2013.

[163] Kevin Poulsen. Software Bug Contributed to Blackout. `http://www.securityfocus.com/news/8016`, February 2004.

[164] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.

[165] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *LNCS*, pages 479–485, Heidelberg, Germany, 2003. Springer.

[166] Arend Rensink, Iovka Boneva, Harmen Kastenberg, and Tom Staijen. *User Manual for the GROOVE Tool Set.* Department of Computer Science, University of Twente, P.O.Box 217, 7500 AE Enschede, The Netherlands, Nov 2012.

[167] Arend Rensink, Ákos Schmidt, and Dániel Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 3256 of *LNCS*, pages 226–241, Heidelberg, Germany, 2004. Springer.

[168] Arend Rensink and Eduardo Zambon. Neighbourhood Abstraction in GROOVE. *ECEASST*, 32:13, 2010.

[169] Arend Rensink and Eduardo Zambon. Pattern-Based Graph Abstraction. In Ehrig et al. [63], pages 66–80.

[170] Mark Richters and Martin Gogolla. OCL: Syntax, Semantics, and Tools. In *Object Modeling with the OCL*, volume 2263 of *LNCS*, pages 42–68. Springer, Heidelberg, Germany, 2002.

[171] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 267–276, New York, NY, USA, 2003. ACM.

[172] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes).* Elsevier and MIT Press, 2001.

[173] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.

[174] Olga Runge, Claudia Ermel, and Gabriele Taentzer. AGG 2.0 - New Features for Specifying and Analyzing Algebraic Graph Transformations. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 7233 of *LNCS*, pages 81–88, Heidelberg, Germany, 2011. Springer.

[175] Ingo Schinz, Tobe Toben, Christian Mrugalla, and Bernd Westphal. The Rhapsody UML Verification Environment. In *Proceedings of the Second Software Engineering and Formal Methods, Second International Conference*, pages 174–183, Washington, DC, USA, 2004. IEEE Computer Society.

[176] Ákos Schmidt and Dániel Varró. CheckVML: A Tool for Model Checking Visual Modeling Languages. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language and Applications*, volume 2863 of *LNCS*, pages 92–95, Heidelberg, Germany, 2003. Springer.

[177] Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *WG*, volume 903 of *LNCS*, pages 151–163, Heidelberg, Germany, 1994. Springer.

[178] Martina Seidl, Marion Scholz, Christian Huemer, and Gerti Kappel. *UML @ Classroom - An Introduction to Object-Oriented Modeling.* Undergraduate Topics in Computer Science. Springer, 2015.

[179] Martina Seidl and Nikolai Tillmann, editors. *Tests and Proofs - 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24-25, 2014. Proceedings*, volume 8570 of *LNCS*, Heidelberg, Germany, 2014. Springer.

[180] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, September 2003.

[181] Wuwei Shen, Kevin J. Compton, and James Huggins. A Toolset for Supporting UML Static and Dynamic Model Checking. In *Proceedings of the 26th International Computer Software and Applications Conference*, pages 147–152, Washington, DC, USA, 2002. IEEE Computer Society.

[182] Anthony J.H. Simons and Ian Graham. 30 Things that Go Wrong in Object Modelling with UML 1.3. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, volume 523 of *The Springer International Series in Engineering and Computer Science*, pages 237–257. Springer US, 1999.

[183] Igor Siveroni, Andrea Zisman, and George Spanoudakis. Property Specification and Static Verification of UML Models. In *Proceedings of the Third International Conference on Availability, Reliability and Security*, pages 96–103, Washington, DC, USA, 2008. IEEE Computer Society.

[184] Raymond M. Smullyan. *First-Order Logic.* Courier Dover Publications, New York, NY, USA, 1995.

[185] Michael Soden and Hajo Eichler. Temporal Extensions of OCL Revisited. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *LNCS*, pages 190–205, Heidelberg, Germany, 2009. Springer.

[186] Michael Soden and Hajo Eichler. Towards a model execution framework for Eclipse. In *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, pages 4:1–4:7, New York, NY, USA, 2009. ACM.

[187] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL models using Boolean satisfiability. In *Design, Automation and Test in Europe*, pages 1341–1344, Washington, DC, USA, 2010. IEEE.

[188] Morten Heine Sørensen and Paweł Urzyczyin, editors. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, Amsterdam, the Netherlands, 2006.

[189] Thomas Stahl, Markus Völter, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven Software Development - Technology, Engineering, Management*. John Wiley & Sons, Ltd., Chichester, UK, 2006.

[190] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. the eclipse series. Pearson Eduction, Inc., Boston, MA, USA, 2 edition, 2008.

[191] Kurt Stenzel, Nina Moebius, and Wolfgang Reif. Formal Verification of QVT Transformations for Code Generation. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, volume 6981 of *LNCS*, pages 533–547, Heidelberg, Germany, 2011. Springer.

[192] Martin Strecker. Modeling and Verifying Graph Transformations in Proof Assistants. *Electr. Notes Theor. Comput. Sci.*, 203(1):135–148, 2008.

[193] Martin Strecker. Interactive and automated proofs for graph transformations. Available at: `http://www.irit.fr/~Martin.Strecker/Publications/proofs_graph_transformations.html`, 2012.

[194] Alfred Tarski. On the calculus of relations. *J. Symb. Log.*, 6(3):73–89, 1941.

[195] 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. *Sci. Comput. Program.*, 76(2):119–135, February 2011.

[196] Emina Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2009. AAI0821754.

[197] Emina Torlak and Greg Dennis. Kodkod for Alloy Users. In *First Alloy Workship, Portland, Oregon, November 6, 2006*, 2006.

[198] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for Construction and Analysis of Systems*, volume 4424 of *LNCS*, pages 632–647, Heidelberg, Germany, 2007. Springer.

[199] Javier Troya and Antonio Vallecillo. A Rewriting Logic Semantics for ATL. *Journal of Object Technology*, 10:5: 1–29, 2011.

[200] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, 2004.

[201] Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.

[202] Shao Jie Zhang and Yang Liu. An Automatic Approach to Model Checking UML State Machines. In *Proceedings of the Fourth International Conference on Secure Software Integration and Reliability Improvement*, pages 1–6, Washington, DC, USA, 2010. IEEE Computer Society.

[203] Paul Ziemann and Martin Gogolla. OCL Extended with Temporal Logic. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of Systems Informatics*, volume 2890 of *LNCS*, pages 351–357, Heidelberg, Germany, 2003. Springer.