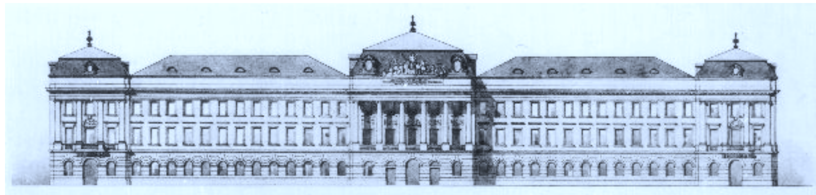
I S I S
RESEARCH
REPORT



INSTITUT FÜR SOFTWARETECHNIK
UND INTERAKTIVE SYSTEME

BUSINESS INFORMATICS GROUP

A FEATURE-BASED CLASSIFICATION OF FORMAL VERIFICATION TECHNIQUES FOR SOFTWARE MODELS

SEBASTIAN GABMEYER PETRA KAUFMANN
MARTINA SEIDL

ISIS RESEARCH REPORT BIG-TR-2014-1
MAY 2014

Institut für Softwaretechnik und
Interaktive Systeme
Business Informatics Group
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18804
Fax: +43-1-58801-18896
office@big.tuwien.ac.at
www.big.tuwien.ac.at

TU WIEN

Business
Informatics
Group

ISIS RESEARCH REPORT

ISIS RESEARCH REPORT BIG-TR-2014-1, MAY 2014

A FEATURE-BASED CLASSIFICATION OF FORMAL VERIFICATION TECHNIQUES FOR SOFTWARE MODELS

Sebastian Gabmeyer[1] Petra Kaufmann[1] Martina Seidl[2]

Abstract. Software models are the core development artifact in model-based engineering (MBE). The MBE paradigm promotes the use of software models to describe structure and behavior of the system under development and suggests to automatically generate the executable code from these models. Thus, defects in the models most likely propagate to the executable code. To detect defects already at the modeling level many approaches propose to use formal verification techniques to ensure the correctness of models. These approaches are the subject of this survey. We review the state of the art of formal verification techniques for software models and provide a feature-based classification that allows us to categorize and compare the different approaches.

[1]Institute of Software Technology and Interactive Systems, Vienna University of Technology, Favoritenstraße 9-11, A-1040 Vienna, Austria; email: {gabmeyer,kaufmann}@big.tuwien.ac.at.

[2]Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, A-4040 Linz, Austria; email: martina.seidl@jku.at

Acknowledgements: This work is supported by the Vienna Science and Technology Fund (WWTF) under grant ICT10-018.

Contents

1 Introduction

In contrast to traditional software engineering, where software models like class diagrams and state machines are merely used for design and documentation purposes, model-based engineering (MBE) projects position software models at the center of the development process [137]. A typical MBE-based software development project iteratively builds the models that describe the platform-independent implementation of the software and generates therefrom the platform-specific, executable code [146]. Consequently, defects in the software models propagate directly to the executable code and result in errors, or, more generally, malfunctions in the deployed system. Moreover, because models are built from the initial stages of the development process onward, the early identification and subsequent correction of defects is, in many cases, easier and often leads to lower development costs as compared to their detection in later development phases [20]. To this end, the correctness of the software models has to be ensured in order to obtain correct software.

In this survey, we review the state-of-the-art of methods and approaches for formally verifying the correctness of software models. We selected these approaches following an extensive literature study and classified each of these approaches based on its distinguishing features. This feature-based classification resulted in a feature model [76] that allows us to categorize each verification approach according to its pursued verification goal, the type of analyzable software models, the encoding of the software model used by the underlying verification engine, the specification language used to phrase the properties that the system should satisfy, and the employed verification technique. The feature model provides an easy-to-use instrument to obtain an overview of formal verification approaches for software models. Its benefits are two-fold. It supports, on the one hand, the identification of suitable solutions to a given verification problem that might be encountered during the development process. On the other hand, it allows a direct positioning of novel approaches in the research landscape.

Besides the formal verification of semantic properties there exist formal and informal verification techniques that are out of scope of this survey. In particular, we do not survey the vast body of formal techniques that analyze syntactic properties of software models. This category of verification techniques embraces, among others, approaches that analyze (*a*) the structural consistency with respect to, e.g., a metamodel [70, 58], (*b*) the consistency across a set of models that provide different views onto the same system [45], and (*c*) confluence and termination properties of transformations [46]. Further, informal techniques that perform a test-based, non-exhaustive analysis to raise the confidence in the software model's semantic correctness are not reviewed.

This survey is structured as follows. In the next section, we first introduce concepts and terminology used throughout this survey. We present our feature-based classification in Section 3 resulting in the feature model that we introduce in Section 4. This feature model allows us to compare the numerous verification approaches for software models discussed in Section 5. We conclude with a discussion and point out possible future research directions in Section 6.

This survey is an extended, completely revised, and in large parts rewritten version of previous work that we presented at the VOLT workshop [52].

2 Background

In computer science, the term *model* is heavily overloaded. 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 paper, we distinguish between *software models* and *logical models* and use these terms explicitly whenever ambiguities might arise.

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-based engineering. Then we shortly review the formal verification techniques relevant for this work.

2.1 Terms and Notions of Software Modeling and Model-Based Engineering

In general, modeling is the act of building abstract representations of certain observations that reflect the typical way how human cope with reality [92, 132]. Models are based on an *original* phenomenon, item, system, etc., which either already exists or is subject to be built. They are created with the pragmatic intention of using, for a special purpose, the simplified and abstracted model in place of the original. As models reduce the original to a relevant subset they allow us to communicate concepts and reason about things that are not (yet) there. This explains the attractiveness of adopting models in engineering disciplines.

Similar to construction plans in civil engineering, models in computer science are used to design multiple views onto a software system before actually implementing it in terms of executable program code [137, 136]. Due to the increasing complexity of modern software projects, new development approaches have been devised, among those, a shift from code-centric to model-centric development paradigms is proposed [15]. In model-centric software development paradigms like *model-based engineering* (MBE), software models are treated not only as informal design sketches or (outdated) documentation artifacts during the design phase, but as "first-class citizens." The idea of MBE is to automate the repetitive task of translating diagrammatic blueprints to code such that developers can concentrate on creative and non-trivial tasks that computers cannot do.

Modeling Languages and the MDA

To establish a commonly accepted set of key concepts and to preserve interoperability between domain-specific modeling languages, the Object Management Group (OMG) released the specification for *Model-Driven Architecture* (MDA) [110] that places models at the center of the entire software development process and standardizes the definitions of *models*, *metamodels*, and *meta-metamodels*. Therein, the OMG proposes a layered framework, called the *metamodeling stack*, which is organized into three layers. The topmost meta-metamodel layer M3 manifests the role of the *Meta-Object Facility* (MOF) [113] as the unique and self-defined metamodel for building metamodels, i.e., MOF is defined recursively by MOF itself. Thus, MOF is the meta-metamodel that ensures interoperability between any two metamodels conforming to it. MOF is comparable to the *Extended Backus-Naur Form* (EBNF), the metagrammar for expressing programming languages. The Eclipse Modeling Framework (EMF) provides a reference implementation for MOF, called Ecore [147], which enjoys broad acceptance in industry and academia. The metamodel layer M2 contains any metamodel defined with MOF and includes, for example, the Unified Modeling Language (UML) [115], or any other custom, domain-specific metamodel. A metamodel at this level corresponds to an EBNF definition of a programming language, examples of which are the grammars that define C# or Java. A metamodel defines the abstract syntax of the modeling language and is usually supplemented with one or more concrete syntaxes. A graphical concrete syntax, defined again as a metamodel at level M2, frames graphical elements such as shapes and edges and associates those elements with corresponding elements of the abstract syntax metamodel. The model layer M1 contains any model built with a metamodel of layer M2, e.g., a UML model. Resorting again to our previous analogy with programming languages, models correspond to specific programs written in any programming language specified with an EBNF definition. Finally, the concrete layer M0 reflects any model based representation of a real situation. This representation is an instance of a model defined at layer M1. Thus, models at layer M0 correspond to dynamic execution traces of a program of layer M1.

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) [111] 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. Nevertheless, 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 MBE projects and OMG standards.

Model Transformations

Model transformations can express arbitrary computations over models and, thus, take on a pivotal role in MBE [138]. 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 [38]. 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) [74] and the Query/View/Transformation (QVT) standard issued by the OMG [112]. 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 [112] 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.

2.2 Terms and Notions of Formal Verification Techniques

In this section, we introduce formal representations for software models and discuss verification techniques based on theorem proving and on model checking.

Graphs, Hypergraphs, and Petri Nets

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 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 [18]. Both the dedicated handbook [130] and the more recent monograph by Ehrig *et al.* [47] discuss the graph transformation theory extensively. In the following, we summarize the concepts relevant for this work. A *graph* consists of a set of vertices V, a set of edges E, and 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* $m : G \to H$ is a mapping between graphs G and H. A morphism is *injective*, $m : G \hookrightarrow H$, if no two distinct elements in G are mapped to the same element in H. A *graph transformation* (also: *rewriting rule*, *graph production*) $p : L \to R$ describes declaratively how a graph L, the left-hand side (LHS), is rewritten into a graph R, the right-hand side (RHS). We say that a graph transformation p is *applicable to* some graph G if there exists a match $m : L \hookrightarrow G$ that maps the LHS L of p into G. A graph transformation is *applied to* a graph G, thus producing graph H, by removing elements $m(L)$, the images of L under m, from G and replacing them by R.

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 are bipartite graphs that are often used to model concurrent systems and parallel processes [103]. Again, we summarize relevant concepts. A *Petri net* [118] 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-deterministically 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 $m'(s) \geq m(s)$ for every place s in the Petri net. This second property is useful to determine whether the Petri net deadlocks.

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 [107], COQ [36], or PVS [117] 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 (*a*) either accept non-terminating proof searches or (*b*) reduce the expressivity of the logic to a decidable level and thus achieve full automatism that requires no user guidance. Automatic theorem provers like VAMPIRE [84] 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 decidable first-order theories is established by SMT solvers like Z3 [39].

Model Checking

Model checking is an automatic theorem proving technique that proves a proposition valid by exploring and evaluating all relevant – usually a subset of all possible – interpretations over the product of a set of finite domains $D_1 \times \cdots \times D_n$. The proposition is referred to as the *specification* and is usually formulated in a temporal logic. Applied to the verification of hard- and software systems [33, 72], the specification expresses desired or undesired properties of the system and the interpretation corresponds to the valuation of the system's variables, whose values are drawn from domains $D_1, \cdots, D_n$. A distinct valuation of the system's variables identifies a *state* of the system and the set of all possible variable valuations that are reachable from the system's 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 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 $\rho = s_1 s_2 s_3 \ldots$ of states. We say that a state s_n is *reachable* from the initial state s_1 if there exists a finite path $\rho = s_0 \ldots s_n$.

The specification is most commonly formulated either with Computation Tree Logic (CTL) [31] or Linear Temporal Logic (LTL) [119]. 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 $\mathtt{AX}\,\varphi$, $\mathtt{AF}\,\varphi$, $\mathtt{AG}\,\varphi$, and $\mathtt{A}\,\varphi\mathtt{U}\psi$ 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.[1] Safety properties are characteristically specified by $\mathtt{AG}\,\varphi$ and describe invariants of the system [93] 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 $\mathtt{F}\,\varphi$ or $\mathtt{GF}\,\varphi$ [13]. 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 $\mathtt{EF}\,\varphi$ [13].

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 with every additional system variable. This phenomenon is known as the *state explosion problem*. Reducing the number of states and improving the

[1] Note that there exist two classification schemes, namely the *safety-liveness* [2] and the *safety-progress* classification [30].

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 [95] 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) [27]. Later, Biere *et al.* [17] 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 [37] 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) [32].

3 A feature-based Classification of Verification Approaches

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 [29] 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 software model adheres to its specification.

3.1 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.

3.2 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 [60], typed, or attributed and may support inheritance relations or compositions (also: part-of relations) [18]. Approaches that use the notation of an OMG standard may use elements or combinations of UML [115], MOF [113], QVT [112], or OCL [111].

3.3 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 [35] 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 [68] or Kodkod [152] convert models and transformations into relational declarations and predicates.

Logical verification representations can be partitioned into approaches using higher-order logic (HOL) [89], first-order logic [141], dynamic logic [61], rewriting logic [98], relational logic [67], or temporal logics, e.g., CTL [31], LTL [119], or μ-calculus [85]. 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 linear 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.

3.4 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 MBE. 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 $a \to a'$ by automaton A, for some state a', and automaton A can replicate every move $b \to b'$ by automaton B, for some state b', and the pair (a', b') is again in $\mathcal{R}$ [100]. In general, this relation is stronger than language equivalence, i.e., whether two automata accept the same language [100]. 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.

3.5 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 [37] 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.

4 The Feature Model

The classification described above is reflected in the feature model [76] depicted in the left half of Table 1. In the following presentation we use a tabular representation for our feature model, that compactly mirrors the commonly used tree-based representation (cf. [38]). 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: Linear 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 1 shows the classification presented in Section 5. 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 [78], 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 1 for all with model checking-based verification approaches that verify the consistency and behavioral correctness of UML models. In Section 5.4 we provide a more fine-grained comparison of these approaches, which are then summarized in Table 2.

5 Verification Approaches

This section surveys the different verification approaches listed in Table 1 and Table 2. It is structured as follows. In Table 1, we group the approaches by the verification technique they employ, that is, either theorem proving or model checking. As the majority of the reviewed approaches uses model checking we subdivide these approaches into (*a*) rewriting based approaches; (*b*) approaches that verify OCL specifications; and (*c*) approaches that assert the consistency and behavioral correctness of different UML models. Note that we simplified the theoretical presentation in some cases for the sake of readability, comprehension, and space.

5.1 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 [120] synthesize transformations from its specification and thus ensure the translation correctness of the transformations. The synthesis is performed in the interactive theorem prover

Table 1: The *Software Model Verification Approach* feature model.

Software Model Verification Approach	Poernomo, Terrell [120]	Poskitt, Plump [122]	Stenzel *et al.* [148]	Hülsbusch *et al.* [65]	Ehrig, Ermel [48]	Giese, Lambers [55]	Kyas *et al.* [86]	Strecker [149][a]	Schmidt, Varró [134]	Baresi *et al.* [10]	Baresi, Spoletini [11]	Kastenberg, Rensink [78][b]	Arendt *et al.* [6][c]	Narayanan, Karsai [104]	König, Kozioura [82][d]	Boronat *et al.* [22][e]	Gagnon *et al.* [53]	Troya, Vallecillo [153]	Büttner *et al.* [28]	Mullins, Oarga [102]	Al-Lail *et al.* [87]	Bill *et al.* [19][f]	UML Model Checking (see Table 2 for details)
Verification Goal (or)																							
Consistency																							✓
Translation Correctness (xor)																							
Source-Target Analysis				✓	✓	✓								✓									
Transformation Analysis	✓	✓	✓					✓											✓				
Behavioral Correctness (or)																							
Behavior by transformation		(G)						(G)	(G)	(G)	(G)	(G)	(G)		(G)	(Q)		(A)				(G)	
Behavior by operation							(S)										(S)			(O)	(O)		✓
Domain Representation (xor)																							
Graphs		✓		✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓								
OMG Standards (or)																							
UML							✓										✓		✓	✓	✓		✓
MOF	✓		✓													✓		✓				✓	
OCL	✓															✓			✓	✓	✓		
QVT																✓							
Verification Representation (xor)																							
Logic	(H)		(D)				(H)	(H)								(W)	(W)	(W)	(R)		(R)		
Transition System									($\mathcal{L}$)	($\mathcal{L}$)		($\mathcal{G}$)	($\mathcal{G}$)							($\mathcal{A}$)		($\mathcal{G}$)	($\mathcal{L}$)
Graphs		✓		✓	✓	✓							✓	✓									
Specification Language (or)																							
Logic	(H)	(F)	(D)					(H)			(R)	(C)	(μ)			(L,W)	(L)	(W)		(C)	(L)	(C)	(C,L)
Bisimulation Relation				✓	✓	✓								✓									
Graphs		✓							✓	✓		✓			✓								
OCL							✓												✓	✓	✓	✓	
Verification Technique (xor)																							
Theorem Proving (xor)																							
Automatic																							
Manual/Interactive	✓	✓	✓	✓	✓	✓	✓	✓															
Model Checking (and)																							
State space representation (xor)																							
enumerative									✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
symbolic/abstract												✓			✓								
Property type (or)																							
Reachability									✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	✓
Safety									✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
Liveness												✓	✓			✓	✓			✓		✓	✓

Legend:
Behavior: A…ATL, G…Graph Transformation, O…OCL, Q…QVT, S…State Machine
Logic: C…CTL, D…DYL, F…FOL, H…HOL, L…LTL, R…ReL, W…RwL, 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] Web interface available at `http://www.modelevolution.org/prototypes/mococl`

COQ [36]. 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 [145]. 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 [122, 123] present a Hoare calculus for graph transformations, which are specified with graph programs [94]. The calculus consists of a set of axioms and proof rules to assert the partial [122] and the total correctness [123] 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 parametrized 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 [60]. 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 $e = \exists(G \mid \gamma, e')$. An E-condition consists of a premise $G \mid \gamma$, 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 $e = \exists(G \mid \gamma, e')$ 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 [64]).

Verifying QVT Transformations

Stenzel *et al.* [148] verify properties of operational QVT (QVTO) transformations with the interactive theorem prover KIV [79]. They implement a sequent calculus based on dynamic logic [61] 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

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

where $\text{conformsTo}(m, M)$ 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.* [65] 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) [135] and the other on in-situ graph transformations and borrowed contexts [49]. 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[2] 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.

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

The bisimulation relation is established via *borrowed contexts* [63, 66]. A third technique to assert that a model transformation preserves the behavior is presented by Ehrig and Ermel [48]. 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.

Giese and Lambers [55] 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.[3]

Verification of OCL Specifications

Kyas *et al.* [86] 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 [117]. 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 [149] 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 [107]. 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 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.

[3] 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 $r : G \to G'$ transforming a graph G into G', and c' denotes the constraint in the next state.

Strecker [150] proposes two reduction techniques[4] 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.

5.2 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 [62]. 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 [22], 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 [134, 154]. It targets the behavioral verification of systems defined by 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 [125] with GROOVE [124] (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,

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

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) [125].[5] 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.* [10], whose encoding produces BIR (Bandera Intermediate Representation) code for the model checker BOGOR [129]. 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 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 [11] presented an encoding that allows to analyze graph transformations specified in AGG [131] with the ALLOY analyzer [68]. 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 [78] propose GROOVE,[6] 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 [124] 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

[5]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 [12]. However, we cannot provide up-to-date runtimes for CHECKVML as it is currently not available to the public.

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

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, 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 [126] and pattern-based abstraction techniques [127], 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 [126]. 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[7] [6] is a model transformation tool for Ecore models. Internally, it represents Ecore models as typed, attributed graphs with inheritance and containment relations [18]. HENSHIN is thus the only graph-based tool that natively supports containment relations. Similar to GROOVE [124], 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 [54] out-of-the-box, which is able to verify μ-calculus [85] 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 [104] 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 each source instance that is translated into a target instance. The verification engine uses the source instance, the cross-links, and target instance

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

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.* [9] and by König and Kozioura [81], 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.* [9] 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) [32], König and Kozioura [81] 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.[8]

Recently, König and Kozioura extended their CEGAR-based verification approach to attributed graph grammars [83]. 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

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

are computed [37]. 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 [9]. 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 [24]; (*b*) model conformance with respect to OCL-constraints [23]; and (*c*) QVT-like model transformations [22] based on membership equational logic (MEL) [97] and rewriting logic (RWL) [96]. 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 *division-by-zero* error can thus be expressed, for example, by returning the term `NaN`. Sentences in E are conditional equations of the form $t = t' \;\; \mathit{if} \; \bigwedge_{i \in I} v_i = w_i \wedge \bigwedge_{i \in I} t_i : s_i$, 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 $t \rightarrow t' \;\; \mathit{if} \;\; C$ 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[9] and a set of rewrite rules, respectively. Each term, rewritten to its unique normal form[10] 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 [35].[11] The MOMENT2 tool[12] 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 [22]. MAUDE builds the state space as a *derivation tree* for both analyses and proceeds as follows. Given an initial term that represents the

[9]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.

[10]For an introduction to term rewriting refer to [7] and [14].

[11]For an RWL theory to be executable as a system module has to be *coherent* [35].

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

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.* [53] 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 communication diagrams can be (manually) encoded into RWL theories and show how LTL specifications can be verified within MAUDE.

Troya and Vallecillo [153] 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.* [28] verify with ALLOY [68] if an exogenous ATL transformation that is defined for an OCL constrained Ecore model preserves the target model's invariants. From metamodels $\mathcal{M}_I$, $\mathcal{M}_O$, where $\mathcal{M}_I \neq \mathcal{M}_O$, and an ATL transformation $t : \mathcal{M}_I \rightarrow \mathcal{M}_O$ that transforms a source model conforming to $\mathcal{M}_I$ into a target model conforming to $\mathcal{M}_O$ Büettner *et al.* build a *transformation model* that captures $\mathcal{M}_I$, $\mathcal{M}_O$, 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 [153], 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* [69] and its ability to translate counterexamples from ALLOY back into Ecore.

5.3 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.* [40] 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 [156] 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 [128]. 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 [142] 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*) [143]. 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 [51] 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 $\mathrm{oclInState}(s : \mathit{OclState})$ 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.* [25] propose the richest extension by embedding OCL into the observational μ-calculus [26]. 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 [77] introduce a temporal extension based on Dwyer *et al.*'s patterns [43] 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 $\mathrm{isCalled}(op,\ pre,\ post)$ 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 $\mathrm{becomesTrue}(P)$ 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 $\mathtt{G}(req \Rightarrow \mathtt{F}\, ack)$ ("every request is eventually acknowledged") is expressed as `globally` $\mathrm{becomesTrue}(ack)$ `responds to` $\mathrm{isCalled}(req,\ true,\ true)$. 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.[13]

Language Extensions & Implementations

Mullins and Oarga [102] 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[14] 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 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.

Al-Lail *et al.* [87] 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 [156], 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 describes the state space of the system. 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

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

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

a source snapshot yields a transition to a target snapshot. The USE Model Validator [57, 144] 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 [152]. This Boolean formula can be solved with any off-the-shelf SAT-solver like MINISAT [44].

With MOCOCL,[15] Bill *et al.* [19] 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 [31]. 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 explains why the evaluation was successful.

5.4 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 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 synonymously.

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 2, we group the different

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

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 [114] 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 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., [56]) and orthogonal states (e.g., [75]) as well as fork and join states (e.g., [56]) are only supported by a few approaches. Zhang and Liu's [155] 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 [75], while others assume synchronous communication [99]. In case of state machines, the above mentioned incompatible semantic interpretation can be circumvented with POLYGLOT [8], 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.* [109] use classes to describe processes whose behavior is specified by state machines. Likewise, the RHAPSODY VE [133] 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.* [73] check whether for the scenarios shown in a collaboration diagram all required associations are available in the class diagram. Jussila *et al.* [75] 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 [80]. 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 [121] 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.* [91] 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 [103]. Eshuis [50] 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 2 for the details). Grumberg *et al.* [59] translate the state machines, for the purpose of verification, to C code, which is than handed to software model checker CBMC [34]. Bounded model checking is applied, but they point out how unbounded model checking might be realized. The UMC framework [151] 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 [151], a state and event based temporal logic that is tailored towards the verification of UML models. While Mozaffari and Haounabadi [101] propose to translate sequence diagrams to executable, colored Petri nets, on which they perform the verification of the given properties, Shen *et al.* [139] take advantage of verification tools available for abstract state machines [21]. 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.* [105, 106] 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.* [140] 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.* [108] 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 [121] introduces stereotypes into the UML models in order to annotate them with constraints. The specification language column (Spec. Lang.) in Table 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.

6 Conclusion

In model-based engineering (MBE), software models replace textual code as the core development artifacts and constitute the foundation for the (semi-)automatic generation of the executable system. The strength of software models stems from the abstraction they provide in form of distinct views onto the software system, which helps different stakeholders of software development projects to (*a*) cope with the complexity of modern software systems, (*b*) communicate and grasp ideas, and (*c*) respond timely and prudently to changing user requirements. The mere use of MBE techniques, however, does not automatically imply the correctness of a system w.r.t. to its specification. Progress and success of formal verification techniques in hardware design and software engineering has motivated the MBE community to adopt and apply these techniques [72] for the verification of software models. It is thus unsurprising that many verification approaches for software models investigate the peculiarities of lifting verification techniques from hard- and software to modeling.

In this survey, we provide an in-depth review of the efforts made to apply formal verification techniques in the MBE development process. We established a feature model that relates the characteristic properties of different verification approaches. In particular, for each approach we review the verification goal it offers to achieve; the representation of analyzable input models; the representation of the analyzable models used by the verification back-end; the supported specification languages; and the technique used to analyze the software models. By this means, we are able to categorize concisely and uniformly the many different approaches that assert the correctness of software models w.r.t. their specification. 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 MBE;

Table 2: Model Checking Approaches for UML

		Domain Representation					Spec. Lang.			∗	Prop.	
	Authors	Class Diagram	State Machine	Sequence Diagram	Activity Diagram	Collaboration Diagram	Graphical Language	Textual Language	Temporal Logic	Model Checker	Liveness	Safety
behavioral correctness	Balasubramanian *et al.* [8][a]		✓						L	F		✓
	ter Beek *et al.* [151][b]	✓	✓						C	O	✓	
	Del Bianco *et al.* [16]		✓						C	K		✓
	Dong *et al.* [41]		✓						L	S		✓
	Dubrovin, Junttila [42][c]		✓						B	N	✓	✓
	Eshuis [50]	✓			✓				L	N		✓
	Gnesi *et al.* [56]		✓						C	J	✓	✓
	Grumberg *et al.* [59]		✓						L	C	✓	✓
	Jussila *et al.* [75][d]	✓	✓						L	S		✓
	Lam, Padget [88]		✓						C	N		✓
	Lilius, Porres [90]; Porres [121]		✓	✓		✓	✓			S		✓
	Lima *et al.* [91]			✓				✓		S		✓
	Mikk *et al.* [99]		✓						L	S		✓
	Mozaffari, Haounabadi [101]			✓				✓		O		✓
	Niewiadomski *et al.* [105][e]		✓					✓		O		✓
	Oubelli *et al.* [116]			✓					L	S	✓	✓
	Shen *et al.* [139]	✓	✓					✓		A	✓	✓
	Siveroni *et al.* [140]	✓	✓					✓		S		
	Zhang, Liu [155][f]		✓						L	P	✓	✓
consist.	Ji *et al.* [73]	✓	✓			✓	✓			S		✓
	Knapp, Wutke [80][g]		✓	✓			✓			S		✓
	Ober *et al.* [108][h]	✓	✓				✓			I		✓
	Schinz *et al.* [133]		✓	✓			✓			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/`

- 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 and further hindered by the lack of a common benchmarks;
- the large amount of literature on formal verification techniques in MBE illustrates, however, their huge potential.

We identified three major directions of possible future work with a high potential to increase the practicality of verification techniques applicable in MBE-based development processes. These are concerned with the reduction of the state space using abstractions, comparability of different approaches, and usability.

Abstractions from models Almost all presented model checking-based approaches enumerate the state space explicitly and do not use abstractions to reduce its size. Although models, by their very nature, abstract from irrelevant implementation details, the amount of information they contain easily surpasses the capabilities of modern verification engines. Amidst this information, models contain easily extractable data that could be used to guide the reduction of the state space. In our opinion, the data most promisingly exploitable are (*a*) multiplicity constraints that provide lower and upper bounds; and (*b*) OCL constraints that impose structural and behavioral restrictions. Structural restrictions constrain the domain of possible instance models, for example, they restrict an attribute's value to a bounded interval or a set of values satisfying some characteristic property. Behavioral restrictions control the set of reachable states and, in essence, define the control flow through the system. This kind of information is usually not available explicitly to frameworks developed to verify the correctness of software; yet, approaches devised to verify the correctness of software models do not exploit it extensively. Thus, we anticipate to see more verification approaches that combine results from different analyses, e.g., multiplicities and abstract interpretation, to obtain finer-grained abstractions.

Evaluation and Benchmarks Currently, several factors prohibit an authoritative comparison of different verification approaches with respect to their performance and usability. First, only a few of the proposed approaches have been implemented, and from those approaches that describe an implementation, only a few are publicly available. Overall, from the 45 reviewed approaches only 14 provide an accessible implementation. Second, due to the ambiguities in the semantic definitions discovered in, e.g., the UML standard, many approaches propose and implement their own interpretation of these definitions. This leads to inconsistent evaluation results if compared directly. Third, a standard set of benchmarks that allows an objective evaluation of the different approaches needs to be collected. Experience from the field of hard- and software verification has shown that a competitive comparison of the available verification approaches has stirred industrial interest and subsequently led to the contribution of real-world benchmarks. A competition may not only lead to more and better benchmarks, but may also increase the number of available verifiers and spark improvements in those that already exist (cf. SAT competition [71]).

Usability Given that only a few approaches provide an implementation it is less surprising that the usability considerations are in many cases out of scope. Nonetheless, the usability certainly requires much more attention if the user base of the proposed verification approaches is intended to grow beyond the scientific/research community. For example, the way results are presented to the user often require thorough knowledge of the underlying verification back-end. Thus, a verification approach is required either to translate the result of a lower level encoding back to the domain representation the user is familiar with or to perform the analysis directly in the domain representation, which obsoletes the translation. Further, a common interchange format for problem descriptions and the returned results eases not only the definition of a common set of benchmarks, but also the combination of different verifiers. Above all, it increases the productivity of the whole research community if reading and writing of problem descriptions and result files can be delegated to a common API.

*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, 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, 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, Germany, 2010. Springer.

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

[8] 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, Germany, 2013. Springer.

[9] 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, Germany, 2001. Springer.

[10] 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.

[11] 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, Germany, 2006. Springer.

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

[13] 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, Germany, 2001.

[14] 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.

[15] Jean Bezivin. On the unification power of models. *Software and Systems Modeling*, 4:171–188, 2005.

[16] 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, USA, 2002. IEEE Computer Society.

[17] 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, Germany, 1999. Springer.

[18] 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.

[19] 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.

[20] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 1981.

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

[22] 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, Germany, 2009. Springer.

[23] 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, Germany, 2009. Springer.

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

[25] 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, Germany, 2002. Springer.

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

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

[28] 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, Germany, 2012. Springer.

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

[30] 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, Germany, 1993.

[31] 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, Germany, 1981. Springer.

[32] 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.

[33] Edmund M. Clarke, Anubhav Gupta, Himanshu Jain, and Helmut Veith. Model Checking: Back and Forth between Hardware and Software. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *LNCS*, pages 251–255, Germany, 2005. Springer.

[34] 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 *Lecture Notes in Computer Science*, pages 168–176, Germany, 2004. Springer.

[35] 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*, Germany, 2007. Springer.

[36] The Coq Proof Assistant, 2012.

[37] 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.

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

[39] 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, Germany, 2008. Springer.

[40] 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.

[41] 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, USA, 2001. IEEE Computer Society.

[42] 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, USA, 2008. IEEE.

[43] 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.

[44] 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, Germany, 2003. Springer.

[45] Alexander Egyed. Instant consistency checking for the UML. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *Proceedings of the 28th International Conference on Software Engineering*, pages 381–390, New York, NY, USA, 2006. ACM.

[46] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination Criteria for Model Transformation. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering*, volume 3442 of *LNCS*, pages 49–63, Germany, 2005. Springer.

[47] 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.

[48] Hartmut Ehrig and Claudia Ermel. Semantical Correctness and Completeness of Model Transformations Using Graph and Rule Transformation. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformation*, volume 5214 of *LNCS*, pages 194–210, Germany, 2008. Springer.

[49] 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.

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

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

[52] 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.

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

[54] 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.

[55] Holger Giese and Leen Lambers. Towards Automatic Verification of Behavior Preservation for Model Transformation via Invariant Checking. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation*, volume 7562 of *LNCS*, pages 249–263, Germany, 2012. Springer.

[56] 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, USA, 1999. IEEE Computer Society.

[57] 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.

[58] Carlos A. González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. EMFtoCSP: A tool for the lightweight verification of EMF models. In *Formal Methods in Software Engineering: Rigorous and Agile Approaches*, pages 44–50, USA, 2012. IEEE.

[59] 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, Germany, 2012. Springer.

[60] 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, Germany, 2002. Springer.

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

[62] 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, Germany, 1998. Springer.

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

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

[65] 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, Germany, 2010. Springer.

[66] 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.

[67] 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.

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

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

[70] Ethan K. Jackson, Tihamer Levendovszky, and Daniel Balasubramanian. Reasoning about Metamodeling with Formal Specifications and Automatic Proofs. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, volume 6981 of *LNCS*, pages 653–667, Germany, 2011. Springer.

[71] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The International SAT Solver Competitions. *AI Magazine*, 33(1):6, 2012.

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

[73] 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, USA, 2012. IEEE.

[74] 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, Germany, 2005. Springer.

[75] Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala Latvala, and Ivan Porres. Model Checking Dynamic and Hierarchical UML State Machines. In Thomas Kühne, editor, *Models in Software Engineering*, volume 4364 of *LNCS*, page 15, Germany, 2006. Springer.

[76] 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.

[77] 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, Germany, 2012. Springer.

[78] 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, Germany, 2006. Springer.

[79] The KIV system, October 2012.

[80] Alexander Knapp and Jochen Wuttke. Model Checking of UML 2.0 Interactions. In Thomas Kühne, editor, *Models in Software Engineering*, volume 4364 of *LNCS*, pages 42–51, Germany, 2006. Springer.

[81] 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, Germany, 2006. Springer.

[82] 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.

[83] Barbara König and Vitali Kozioura. Towards the Verification of Attributed Graph Transformation Systems. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformation*, volume 5214 of *LNCS*, pages 305–320, Germany, 2008. Springer.

[84] 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, Germany, 2013. Springer.

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

[86] 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.

[87] 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.

[88] 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, USA, 2004. IEEE Computer Society.

[89] 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.

[90] 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, USA, 1999. IEEE Computer Society.

[91] 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.

[92] Miroslaw Malek. The Art of Creating Models and Models Integration. In *Model-Based Software and Data Integration*, volume 8 of *Communications in Computer and Information Science*, pages 1–7. Springer, Germany, 2008.

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

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

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

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

[97] 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, Germany, 1997. Springer.

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

[99] 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, USA, 1998. IEEE Computer Society.

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

[101] 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, USA, 2011. IEEE.

[102] 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, Germany, 2007. Springer.

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

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

[105] 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.

[106] 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, Germany, 2009. Springer.

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

[108] 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, Germany, 2004. Springer.

[109] 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.

[110] Object Management Group OMG. Model Driven Architecture (MDA) Guide V1.0.1. `http://www.omg.org/mda/`, January 2006.

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

[112] 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.

[113] 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.

[114] 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.

[115] 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.

[116] 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.

[117] 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, Germany, 1992. Springer.

[118] Carl Adam Petri and Wolfgang Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008.

[119] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, USA, 1977. IEEE Computer Society.

[120] 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, Germany, 2010. Springer.

[121] Ivan Porres. Modeling and Analyzing Software Behavior in UML, 2001.

[122] Christopher M. Poskitt and Detlef Plump. Hoare-Style Verification of Graph Programs. *Fundam. Inform.*, 118(1-2):135–175, 2012.

[123] Christopher M. Poskitt and Detlef Plump. Verifying Total Correctness of Graph Programs. *ECEASST*, 61:20, 2013.

[124] 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, Germany, 2003. Springer.

[125] 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, Germany, 2004. Springer.

[126] Arend Rensink and Eduardo Zambon. Neighbourhood Abstraction in GROOVE. *ECEASST*, 32:13, 2010.

[127] Arend Rensink and Eduardo Zambon. Pattern-Based Graph Abstraction. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation*, volume 7562 of *LNCS*, pages 66–80, Germany, 2012. Springer.

[128] Mark Richters and Martin Gogolla. OCL: Syntax, Semantics, and Tools. In *Object Modeling with the OCL*, volume 2263 of *LNCS*, pages 42–68. Springer, Germany, 2002.

[129] 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.

[130] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[131] 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, Germany, 2011. Springer.

[132] Hermann Schichl. Models and History of Modeling. In Josef Kallrath, editor, *Modeling Languages in Mathematical Optimization*, Applied Optimization, chapter 2, pages 25–36. Springer, Germany, 2004.

[133] 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, USA, 2004. IEEE Computer Society.

[134] 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, Germany, 2003. Springer.

[135] 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, Germany, 1994. Springer.

[136] Ed Seidewitz. What Models Mean. *Software, IEEE*, 20(5):26–32, 2003.

[137] Bran Selic. The Pragmatics of Model-driven Development. *Software, IEEE*, 20(5):19–25, 2003.

[138] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, September 2003.

[139] 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, USA, 2002. IEEE Computer Society.

[140] 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, USA, 2008. IEEE Computer Society.

[141] Raymond M. Smullyan. *First-Order Logic*. Courier Dover Publications, New York, NY, USA, 1995.

[142] 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, Germany, 2009. Springer.

[143] 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.

[144] 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, USA, 2010. IEEE.

[145] 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.

[146] 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.

[147] 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.

[148] 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, Germany, 2011. Springer.

[149] Martin Strecker. Modeling and Verifying Graph Transformations in Proof Assistants. *Electr. Notes Theor. Comput. Sci.*, 203(1):135–148, 2008.

[150] Martin Strecker. Interactive and automated proofs for graph transformations. 2012.

[151] 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.

[152] 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, Germany, 2007. Springer.

[153] Javier Troya and Antonio Vallecillo. A Rewriting Logic Semantics for ATL. *Journal of Object Technology*, 10:5: 1–29, 2011.

[154] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, 2004.

[155] 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, USA, 2010. IEEE Computer Society.

[156] 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, Germany, 2003. Springer.