Volt
Second Workshop on Verification Of Model Transformations, 2013,

A Classification of Model Checking-Based Verification Approaches for Software Models

Sebastian Gabmeyer[a] Petra Brosch[a] Martina Seidl[b]

a. Vienna University of Technology, Vienna
{gabmeyer|brosch}@big.tuwien.ac.at

b. JKU Linz, Linz
martina.seidl@jku.ac.at

Abstract We present a feature-based classification of software model verification approaches. We classify a verification approach in a system-centric view according to the pursued verification goal, the representation of the input and the verification domain, the specification language of the properties, and the employed verification technique, which is one of either model checking, theorem proving, or static analysis. Our proposed feature model reflects this classification. Due to space limitations we focus on model checking-based verification techniques in this paper.

1 Introduction

We propose a classification of verification approaches used in the context of model-based engineering (MBE) to assert the correctness of software models. The landscape of verification approaches in MBE is broad and the application areas are manifold, so a structured characterization as offered by feature models allows us to compare different approaches.

The verification of model transformations is one of the application domains of formal verification techniques in MBE, which has been surveyed extensively by others [2, 6]. *Model transformations* are the enabling technique of MBE, as they allow us to employ software models beyond design and documentation purposes for various intents [1, 9], such as refinement, simulation, and code generation. As software models and their transformations are used to describe the structure and behavior of systems, from which, for example, executable code is automatically generated, errors made in the former most likely propagate to the latter. To improve the quality of software models, many works have been presented which suggest to use formal verification techniques including theorem proving, static analysis, and model checking. In this work we aim to contribute a system-centric classification of verification approaches

that takes, in addition to transformations, the domain models into account and focuses on semantic verification goals. We represent our classification as a feature model in the style of Czarnecki *et al.* [9]. Due to space limitations we restrict the discussion of our classification to model checking-based approaches and save the presentation of the full classification including other verification techniques for future work.

Model checking [7] is the process of rigorously testing if a system or implementation[1] satisfies its specification. The system is defined by the collection of all possible configurations, called *states*, that it can be in. The state of the system is defined by a valuation of the system's variables: Different valuations of the system's variables yield different states. The set of all states is called the state space of the system. The states are connected by transitions that describe admissible state changes. Model checking is exhaustive in that it explores the entire state space of the system and checks if the specification holds. The downside of this fully automatic exploration procedure is that model checking can only verify finite state systems and even then it is often the case that the set of states grows quickly with each additional system variable. This phenomenon is referred to as the *state space explosion* problem.

This paper is structured as follows. First, we review a small selection of verification approaches in Section 2. We propose a classification for verification techniques in the context of MBE and present a feature model. We finally illustrate how this feature model is used to classify the reviewed model checking approaches from Section 2 and conclude with an outlook.

2 A Selection of Verification Approaches

In this section we present a small selection of software model verification approaches to demonstrate the classification capabilities of our feature model that will be introduced in the next section.

MOMENT2.[2] Boronat *et al.* verify the behavioral correctness of MOF-based metamodels, which define the system, and QVT-like model transformations, which define the semantics of the system [3]. They translate OCL-constrained metamodels and model transformations into rewrite theories [5, 4] that are subsequently analyzed in MAUDE [8]. To check reachability and safety properties MAUDE's `search` function is used that enumerates breadth-first the state space defined by the translated metamodel and model transformations. The search term is specified as an OCL-constrained *model pattern* that describes *bad* system states. It is also possible to verify properties expressed in linear temporal logic (LTL) but these need to be defined directly in MAUDE.

GROOVE.[3] Kastenberg *et al.* propose an enumerative state model checking approach to verify the behavioral correctness of object-oriented (OO) systems [12]. An OO-system is specified as an attributed type graph with inheritance relations and the system's behavior is defined by graph transformations. Hence, states are represented by (instance) graphs conforming to the type graph. Between two states s and s' there exists a transition if a graph transformation can be applied to the graph of s such that the result is isomorphic to the graph of s'. The resulting state-transition structure

[1] We will use the terms *system* and *implementation* interchangeably. The latter is naturally used when we are concerned about software development.

[2] `http://moment.dsic.upv.es/content/blogcategory/37/76/`

[3] `http://groove.sourceforge.com`

is a graph transition system (GTS), over which safety and liveness properties can be checked that are specified as formulae in computation tree logic (CTL).

SOCL*e.* An approach that stays within the technological space of UML is presented by Mullins and Oarga [15]. They propose an extension to OCL, called EOCL, that augments OCL with CTL operators. The SOCLe tool[4] is able to assert the behavioral correctness of a system that is defined by a class diagram, a set of state charts for each class in the class diagram, and an object diagram that defines the initial state. It does so by translating class, state chart, and object diagrams into an abstract state machine and it checks on-the-fly if the system satisfies the specification given as an EOCL expression.

Hugo.[5] Hugo [13] verifies the consistency between multiple UML state charts and sequence diagrams. The sequence diagrams act as the specification and the set of state charts as its implementation. From the sequence diagrams an interaction automaton is synthesized which is used for observing message traces. From the state charts and the synthesized automaton, Promela code is generated, which is the input to the model checker Spin. The state charts are consistent w.r.t. the sequence diagram if the message sequences described by the sequence diagram correspond to paths in the state charts.

Proco.[6] Jussila *et al.* [11] present an approach where they model protocols by the means of state charts representing active classes. The initial configuration is specified by a deployment diagram. The exchanged messages are asynchronous, orthogonal states are possible under certain restrictions. The state charts are translated to Promela in order to check for deadlocks or assertion violations.

3 Feature-based Classification of Verification Approaches

In order to make different verification approaches comparable, we propose the following classification which we apply exemplarily to the model checking approaches discussed in the previous section.

3.1 The Classification

We propose a system-centric view and classify verification approaches by the pursued *verification goal*, the *input domain* of the verification approach, the *representation* employed for the verification, the *specification language* used to describe the properties, and the *verification technique* applied to achieve the verification goal.

Verification Goal. The verification goal describes the purpose or the intent of the verification. We distinguish between *consistency verification*, *translation correctness*, and *behavioral correctness* goals. In approaches targeted at verifying the consistency of models, the goal is to ensure that one or more models do not contain contradicting information. For example, in a multi-view modeling language like UML, where diagrams provide distinct views onto the system under development, it has to be ensured that different diagrams are not inconsistent.

When performing model-to-model or model-to-code transformations, we want to verify the *correctness of the translation.* This category captures all scenarios related to

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

[5] `http://www.pst.informatik.uni-muenchen.de/projekte/hugo`

[6] `http://www.tcs.hut.fi/Research/Logic/SMUML.shtml`

				[3] Moment2	[12] Groove	[15] SOCLe	[11] Procco	[13] Hugo
Software Model Verification Approach	***Verification Goal***							
	(or)	Consistency						✓
		Translation Correctness						
		(xor)	Source/Target Correctness					
			Transformation Correctness					
		Behavioral Correctness		▼	▼	▼	▼	
		(or)	Behavior by transformation	✓	✓			
			Behavior by operation			✓	✓	
	Domain Representation							
	(or)	Graphs			✓			
		OMG		▼		▼	▼	▼
		(or)	UML \ MOF	✓			✓	✓
			OCL			✓		
			QVT	✓				
		DSL						
	Verification Representation							
	(xor)	Algebraic Terms		✓				
		Transition System			(GTS)	(ASM)	(LTS)	(LTS)
		Relations						
	Specification Language							
	(or)	First-order Logic						
		Temporal Logic		(LTL)	(CTL)	(CTL)	(LTL)	(LTL)
		Rewriting Logic		✓				
		Automata						
		OCL				✓		
	Verification Technique							
	(xor)	Theorem Proving						
		Static Analysis						
		Model Checking		▼	▼	▼	▼	▼
		(and)	*State space representation*	▼	▼	▼	▼	▼
			(xor) enumerative	✓	✓	✓	✓	✓
			(xor) symbolic					
			Property type	▼	▼	▼	▼	▼
			(or) reachability	✓	✓	✓		
			(or) safety	✓	✓	✓	✓	✓
			(or) liveness	✓	✓	✓		

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

the verification of model transformations. The correctness of a model transformation that converts, for example, a UML class diagram into an ER diagram, can be asserted either (a) by analyzing the source and the target models or (b) by analyzing the transformation directly. In the second case, the verification of the transformation may include an analysis of the transformation's *termination* and *confluence* properties. Translation correctness is exhaustively discussed by Amrani *et al.* [2].

When we verify the behavioral correctness of a system we check if the system behaves according to its specification. The system behavior is defined either by a set of *transformations* or by the system's *operations and operation contracts*. An operation contract consists of a set of preconditions that define in what system states the operation can be executed and a set of postconditions that define the state of the system after the operation has terminated. In PROCO, for example, the state charts may be enhanced which assertions for which it is proved that they are never violated during execution. We want to point out that the transformation is, in contrast to the translation correctness goal described above, not subject of the verification, rather it is assumed that it correctly defines the behavior of the system. It is thus the modeler's task to ensure that the transformations correctly reproduce the intended behavior.

Domain Representation. The input or *domain representation* defines the type of the input models that the verification approach is able to analyze. *Graphs* and graph transformations are formal representation, which are possibly typed, attributed, and enhanced in expressivity by inheritance and composition relations. Further, we classify UML, MOF, OCL, and QVT based models as *OMG*-related representations. The *DSL* category captures representations like ATL or Petri Nets. Concrete examples of domain representations are state charts used in PROCO and SOCLe as well as in HUGO, which additionally considers sequence diagrams.

Verification Representation. The *verification representation* classifies the approaches by the type of representation that is used to perform the verification. We distinguish between *algebraic*, *state-transition*, and *relational* representations. As most approaches do not implement their own verification back-end, but use available verification tools out-of-the-box, this representation correlates with the input of the used verification tool. For example, approaches that employ MAUDE [8] represent models as algebraic data types such that MAUDE's search and model checking capabilities may be used to verify the system. Likewise, approaches that use ALLOY [10] as their verification back-end will use a relational representation for their models.

Specification Language. Different *specification languages* for expressing the properties to be checked are in use with varying degrees of expressivity. We distinguish between *first-order*, *temporal*, and *rewriting logic* [14]. In addition *automata* and *OCL* may be used to specify properties of a system. SOCLe, for example, defines the system behavior with simplified state charts that are translated to an ASM. The specification is usually provided by a temporal logic formula, e.g., linear temporal logic (LTL) or computation tree logic (CTL). For example, GROOVE checks if an OO-system, whose behavior is defined by a set of graph transformations, satisfies its specification given as a CTL formula. HUGO and PROCO use Promela, the input language of the model checker Spin, which performs LTL-based model checking.

Verification Technique. When classifying the approaches by the employed *verification technique* we distinguish between *static analysis*, *theorem proving*, and *model checking*. With this information, the capabilities and limitations of the different approaches become comparable. All approaches presented in the previous section, fall into the class of model checking. We further categorize model checking-based approaches by their state space exploration technique and by the type of properties that can be verified. If the state space is explored *enumeratively*, every possible combination of different variable valuations is analyzed. Contrary, *symbolic* state space

representations use abstractions to symbolically represent a set of states as a single state. We distinguish three types of properties. *Reachability* properties test if a state (or set of states) with a certain valuation is reachable. *Safety* properties ensure that nothing bad happens, while *liveness* properties ensure that something good will eventually happen. Typical safety properties are system invariants that need to hold in every state, while (non-)termination and (non-)starvation are typical liveness properties. The concrete classification of the previously discussed approaches is shown in Table 1 which we describe below.

3.2 The Feature Model

The classification described above is reflected in our feature model and depicted in the left half of Table 1. The root, named *software model verification approach*, is decomposed into five child features named verification goal, input domain, verification representation, and so on. These main features are further refined as described above. All features in the table are mandatory. Names written in *italic* denote features that are further refined by either *and*, *or*, or *xor* decompositions. An *and* (*or*, *xor*) decomposition mandates that each (at least one, no more than one) of the child features is present. For example, the *Verification Goal* feature is *or*-decomposed into the *Consistency*, the *Translation Correctness*, and the *Behavioral Correctness* feature, which is in turn *xor*-decomposed into the *Behavior by Transformation* and the *Behavior by Operation* features.

The right half of Table 1 shows the classification of the verification approaches presented in Section 2. This part of the table is read as follows. A check-mark in the table indicates that the feature is supported; a filled, downward-pointing triangle indicates that a child feature is supported, and in case a feature has different characteristic values the actual value is displayed. For example, the feature *Temporal Logic*, a child feature of *Specification Language*, has characteristic values *CTL* and *LTL*.

4 Conclusion

As software models may be employed for diverse purposes and goals, the landscape of verification approaches in MBE is broad. Existing classification schemes focus only on the transformational aspects of MBE. We extend this work to a system-centric view and present a classification that distinguishes the different verification approaches by their verification goal, their domain and target representation, their specification language, and the employed verification technique. We propose a corresponding feature model that, on the one hand, allows us to classify existing approaches and, on the other hand, acts as reference for users looking for verification support for their software models. We demonstrated the classification capabilities of our feature model with several representative verification approaches. In future work, we plan to extend our survey and include other verification approaches for software models, especially approaches in the areas of static analysis and theorem proving, which we did not describe in this paper.

References

[1] M. Amrani, J. Dingel, L. Lambers, L. Lúcio, R. Salay, G. Selim, E. Syriani, and M. Wimmer. Towards a model transformation intent catalog. In *Proceed-*

ings of the First Workshop on the Analysis of Model Transformations, AMT '12, pages 3–8, New York, NY, USA, 2012. ACM.

[2] M. Amrani, L. Lucio, G. M. K. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. L. Traon, and J. R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In G. Antoniol, A. Bertolino, and Y. Labiche, editors, *ICST*, pages 921–928. IEEE, 2012.

[3] A. Boronat, R. Heckel, and J. Meseguer. Rewriting Logic Semantics and Verification of Model Transformations. In M. Chechik and M. Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2009.

[4] A. Boronat and J. Meseguer. Algebraic Semantics of OCL-Constrained Metamodel Specifications. In M. Oriol and B. Meyer, editors, *TOOLS (47)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 96–115. Springer, 2009.

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

[6] D. Calegari and N. Szasz. Verification of model transformations. *Electron. Notes Theor. Comput. Sci.*, 292:5–25, Mar. 2013.

[7] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. The MIT press, 1999.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

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

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

[11] T. Jussila, J. Dubrovin, T. Junttila, T. L. Latvala, and I. Porres. Model Checking Dynamic and Hierarchical UML State Machines. In *Proceedings of the 3rd Workshop on Model Design and Validation (MoDeVa 2006)*, 2006.

[12] H. Kastenberg and A. Rensink. Model Checking Dynamic States in GROOVE. In A. Valmari, editor, *SPIN*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305. Springer, 2006.

[13] A. Knapp and J. Wuttke. Model Checking of UML 2.0 Interactions. In T. Kühne, editor, *MoDELS Workshops*, volume 4364 of *Lecture Notes in Computer Science*, pages 42–51. Springer, 2006.

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

[15] J. Mullins and R. Oarga. Model Checking of Extended OCL Constraints on UML Models in SOCLe. In M. M. Bonsangue and E. B. Johnsen, editors, *FMOODS*, volume 4468 of *Lecture Notes in Computer Science*, pages 59–75. Springer, 2007.