Integrating an SMT-based ModelFinder into USE

Nils Przigoda[1,5] Frank Hilken[2] Judith Peters[3] Robert Wille[4,5] Martin Gogolla[2] Rolf Drechsler[1,5]

[1]Group for Computer Architecture, University of Bremen, 28359 Bremen, Germany
[2]Database Systems Group, University of Bremen, 28359 Bremen, Germany
[3]Department of Satellite Ground Systems, OHB System AG, 28359 Bremen, Germany
[4]Institute for Integrated Circuits, Johannes Kepler University Linz, 4040 Linz, Austria
[5]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany

{przigoda,fhilken,gogolla,drechsle}@informatik.uni-bremen.de judith.peters@ohb.de robert.wille@jku.at

***Abstract*—The validation and verification of models becomes increasingly important as the complexity and overall costs of later development stages increase. Although, a variety of tools exists for this purpose, the majority are academic – used as a proof of concept for the theory behind them. Thus, implementations are mostly applicable to subsets of model verification tasks only. In order to execute all necessary verification tasks, the model under verification has to be manually prepared for each tool – usually involving several modeling languages and techniques. The manual work requires expert knowledge and is a source for errors. Simplifying this process is a desired aspect and poses new challenges to tool developers. We demonstrate how a common framework can be used to provide access to multiple model checking techniques by integrating an SMT-based ModelFinder including a high level interface to its functionality. Afterwards, the benefits are discussed comparing the new technique with the existing tool coverage in the framework.**

Keywords: model validation & verification · model finding · tool integration

I. Introduction

Verifying today's computer systems is a task of ever increasing complexity due to the ongoing development of miniaturization and, thus, more complex devices. Although the design of an actual system can be conducted based on abstract and, therefore, smaller models in the first stages of design, the verification of these models – the so-called model-finding – still is a very complex problem. In fact, verification is EXPTIME-hard for UML class diagrams and becomes undecidable with the inclusion of general OCL constraints [1]. Nevertheless, during the last decades several approaches have been published with promising results in the verification of those models. However, this resulted in plenty of different tools implementing different strategies – most of them academic and, hence, far too often poorly maintained and updated. Furthermore, most tools are only using few strategies resulting in a feasibility only for few classes of problems. Consequently, the user has to conduct several analysis tasks for using recent tools in order to verify their model.

First of all, it has to be investigated, which aspects of the model shall be verified. While most tools check for general structural contradiction-freeness, they usually cover only one or two more advanced aspects such as behavioral verification, over-specification or debugging. Therefore, if more than one of these aspects shall be covered, frequently more than one tool has to be used. This may leave the user with a very unpleasant tool-chain. The model under verification has to be transformed into all input formats for all used tools. This opens, first of all, lots of opportunities for model transformation errors, as only few of the model transformations can be conducted automatically. Additionally, most verification tools suffer from certain limitations, due to a limited focus, an out-dated underlying modeling language version or simply bugs. Consequently they, e. g., limit the commands that are supported in their input, which causes the need to further modify the model under verification, e. g., to remove certain parts for a certain verification tool. Finally, after applying the verification tools, this makes the comparability and reliability of those verification results questionable.

To overcome this problem, more generalized tools are needed which include more strategies to finally avoid the need for different tools. Relying on the tool USE (UML based Specification Environment, [2]) as one of the most wide-spread model finders, this paper shows how to include more model finding strategies – in this case the SMT-based ModelFinder – in an existing tool to set a starting point for merging existing tools into fewer, more powerful tools. Based on the existing strategies, we combine the strength of both approaches to overcome the limitations of both to finally use all of their strategies in a comparable manner. As USE's description means and graphical surface are quite sophisticated, it will serve as the base, while the SMT-based ModelFinder will be added as a USE plugin.

In the following section, the background of both the SMT-based ModelFinder and USE will be discussed. Afterwards, in Section III the general workflow of USE will be outlined, before we show how to include the SMT-based ModelFinder as a plugin for USE and explain its workflow. Subsequently, Section IV discusses the results of the combination while Section V concludes the paper.

II. Tools for Model Verification

In the last two decades a lot of different tools to verify UML/OCL models have been developed in both academia and industry. In the following, some of these tools will be introduced.[1]

[1]Please note that the following descriptions of tools are partially taken from the corresponding websites. Furthermore, the list is not complete.

In 2013, the last update for *OCLsolve – a Constraint Solver for UML/OCL*[2] was conducted and published. This tool provides a Java API to define a class model inside a Java program and, on top of this, the designer can define OCL expressions and check whether they are satisfiable or not. To finally solve the resulting satisfiability problem, *SAT4J*[3] or *MiniSat*[4] can be chosen. However, to the best of our knowledge, no model format is defined which can be parsed, thus, it is necessary to describe the whole model in terms of a Java program. Furthermore, we have not found methods to apply other checks than just structural consistency.

UML2Alloy[5], an alternative project, was started in 2005 [3]. According to the manual the last update was in May 2009. *UML2Alloy* as a tool is the product of the scientific approach to formalize UML using *Alloy*[6] [4]. It translates UML class diagrams enriched with OCL constraints to finally form an Alloy model. Afterwards, this Alloy model can be automatically analyzed using the *Alloy Analyzer*.

Consequently, *Alloy* itself translates the given model into a Boolean expression to be analyzed with an embedded SAT solver. Since version 4, *Alloy* includes *Kodkod*[7] [5]. *Kodkod* can be used as a finite model finder and it also can calculate minimal unsatisfiable cores in case the SAT problem derived from the finite model finding problem is not satisfiable.

EMFtoCSP[8] (successor of *UMLtoCSP*[9]) is a tool for the automatic verification of UML or EMF models annotated with OCL constraints which is currently maintained by the SOM Research Team. This tool is based on the *Eclipse Modeling Framework* (*EMF*) [6] and realized as an Eclipse plugin which can automatically check several correctness properties within the model such as the satisfiability of the model or the lack of contradictory constraints. As a solving engine in the background, the *ECLiPSe Constraint Programming System*[10] is used. Like *Alloy* and *Kodkod*, *EMFtoCSP* is still maintained.

Isabelle/HOL-OCL[11] is an interactive proof environment for the Object Constraint Language (OCL). It is implemented as a shallow embedding of OCL into the Higher-order Logic (HOL) instance of the interactive theorem prover Isabelle. Since it relies on *Isabelle*, *Isabelle/HOL-OCL* is not a fully automatic but – as already mentioned – interactive environment. However, the partial simplifications within a proof provided by *Isabelle* itself can still be used to speed up the verification process. Nevertheless, a designer requires expert knowledge to work with *Isabelle/HOL-OCL*.

For this work, the underlying system is the USE tool [2]. Developed and maintained since roughly 15 years, it provides an extensive graphical user interface for several UML diagram types and analysis tools for system states in UML and OCL along other features. A plugin API allows the extension of the tool's capacities, i. e., in this case to integrate an existing model finder. With the plugin API, external programs can access the model using the internal data structures of USE and represent system states in the graphical user interface, e. g., with object and sequence diagrams.

[2]Available at `http://www.mpkrieger.net/oclsolve/`

[3]Available at `http://www.sat4j.org/`

[4]Available at `http://minisat.se`

[5]Available at `http://www.cs.bham.ac.uk/~bxb/UML2Alloy`

[6]Available at `http://alloy.mit.edu/alloy/`

[7]Available at `http://alloy.mit.edu/kodkod`

[8]Available at `https://github.com/SOM-Research/EMFtoCSP`

[9]Available at `http://gres.uoc.edu/UMLtoCSP/`

[10]Available at `http://eclipseclp.org`

[11]Available at `https://www.brucker.ch/projects/hol-ocl/`

III. USE ModelValidator and SMT-based ModelFinder Workflows

So far several tools and approaches for model checking were introduced. However, this paper is focusing on USE as the basis for a generalized model checking tool. Basically, USE provides the general modeling means while the verification approaches are added in form of plugins as shown in Fig. 1. Each plugin extends the capabilities of USE by a potent instance finder based on different technologies, one is based on relational logic/SAT solvers and the other is based on SMT solvers.[12] The two following subsections will discuss the workflows for both the USE ModelValidator and the SMT-based ModelFinder plugin in detail.

A. USE ModelValidator Workflow

The USE ModelValidator plugin is based on the transformation of UML and OCL into relational logic [8]. It was implemented as a plugin for the USE tool reusing parts of the architecture USE provides (UML and OCL metamodels as well as programming logic) and uses the Kodkod library [5] which provides the metamodel for relational logic and interfaces to SAT solvers. This way, the integration into the USE tool was given from the beginning.

When the plugin was first released, it came with several features to instantiate UML/OCL models and check constraints to cover basic validation and verification tasks [9], e. g., checking model consistency by trying to generate a valid system state, generating all valid solutions within the specified model bounds, and checking invariants for independency [10]. Since then, the plugin is maintained and constantly upgraded introducing features of new OCL standards like the operations `selectByKind` and `selectByType` as well as new features such as the classifying terms [11]. Also new features are planned: a GUI for the easy specification of problem bounds and support for derived properties.

The workflow of the USE ModelValidator plugin is depicted in the bottom of Fig. 1. The USE framework provides several interfaces for plugins to attach and get access to the internal data structures containing the loaded model and system state among other information. The structural information of the model – classes, associatons, multiplicities etc. from the class diagram and OCL invariants – is transformed into the relational logic of Kodkod [5]. From here, the Kodkod library performs the remaining transformation into the SAT encoding once provided with a bound configuration to set the bounds of the model search space.

[12]A comparison of the behavior verification capabilities of the standalone tools can be found in [7].

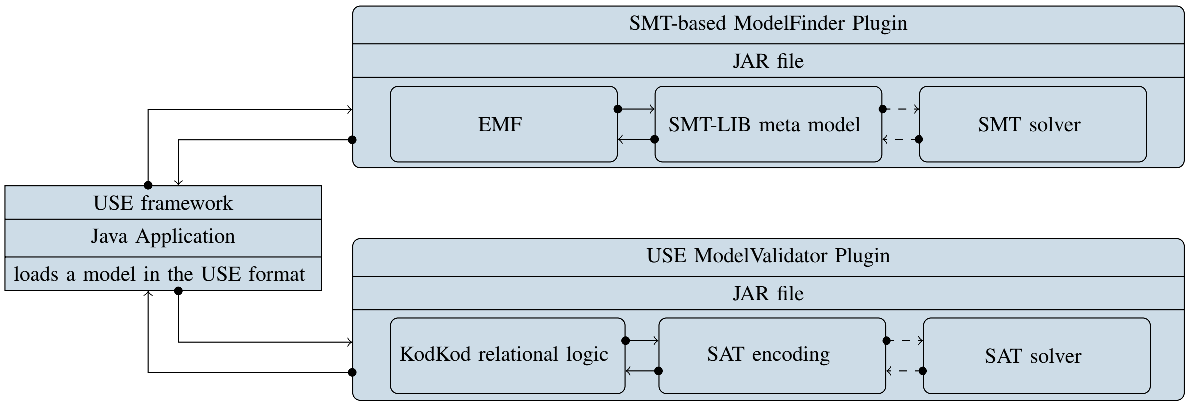

Figure 1: Interactions and workflows of the two different plugins.

The generated SAT instance is given to and solved by a SAT solver and the results are interpreted and transformed back into relational logic by the Kodkod library. The USE ModelValidator plugin takes the results and performs the transformation into a system state in the USE tool, ready for inspection and interpretation by the modeler.

The main focus of the translation into relational logic is on the structural analysis. Model behavior, specified by operations with pre- and postconditions in the USE tool, are not handled. For this reason, a transformation of behavioral models into structural models has been introduced, the so-called filmstripping [12]. With filmstripping, it is also possible to analyze behavioral aspects with the USE ModelValidator. However, a drawback of this approach is that the modeler is working on a different model, the filmstripped version of the model under verification, and has to take care of further details such as frame conditions and the transformation of the results back into the original model manually.

B. *SMT-based ModelFinder Workflow*

While the USE ModelValidator approach already covers most aspects of structural analysis, behavioral aspects are only covered with another plugin and require expert knowledge of the designer as outlined in detail in Section IV. However, these are exactly the strengths of the SMT-based ModelFinder which provides highly advanced behavioral analysis mechanisms. Thus, the SMT-based ModelFinder was chosen to extend the USE environment. As USE already provides a convenient interface for extension with plugins, this was used. The exact mechanisms of how USE and SMT-based ModelFinder were interwoven are outlined throughout this section.

So far, the SMT-based ModelFinder has been used within Eclipse and, thus, the *Eclipse Modeling Framework* (*EMF*) was used as underlying UML/OCL metamodel. However, USE is based on its own metamodel (as previously discussed in Sect. III-A), i. e., in order to use the SMT-based ModelFinder a transformation of the model under verification from the USE format to *EMF* has to be conducted. Again, as for the ModelValidator plugin, the designer has to provide some problem bounds in order to derive a decision problem with a finite search space in the next steps. In the process of model transformation the SMT-based ModelFinder creates an *EMF*-based system state in which the number of objects depends on the maximum number of instances per class as defined by the problem bounds. All attributes and references of all objects will be assigned to a default value, mostly `null`. Later on, during the solving process, these attributes are subsequently assigned with final values eventually forming a valid system state (in case one exists).

In the next step, the system state together with the model is transformed into an instance of the SMT-LIB [13] metamodel, i. e., a precise SMT problem. Now, the SMT instance can be passed to an off-the-shelf SMT solver. Due to the usage of an SMT metamodel, it is theoretically possible to use different solvers. However, the current implementation does support Z3 [14] and partially metaSMT [15] only. If the used solver finds a satisfying assignment, the respective values are transformed back to the *EMF*-based system state of the model to form the actual system state derived by the solving process. Finally, the values are further transformed backwards such that the derived actual system state is visualized in the USE framework.

If a behavioral verification task is applied instead of a single system state, a sequence of them – connected by the application of operations with all their pre- and postconditions – will be generated and solved analogously to the solving process for one system state. The SMT-based ModelFinder plugin API provides commands to access the next or a specific system state for further investigation in the post-solving manual analysis of the resulting actual system state.

So far, the usage of SMT-LIB metamodel within the SMT-based ModelFinder does use libraries for integers, reals, and strings etc. More precisely, only the pure satisfiability subset based on Booleans and bit vectors is used. This ensures that the SMT solver will never return `unknown` as a result.

A detailed analysis of the verification tasks supported by the SMT-based ModelFinder and the underlying functionalities is given by the performance analysis in the next section.

IV. Review of Performance

So far, this paper has focused on how to combine the *USE* framework with both the SMT-based ModelFinder and the USE ModelValidator plugin. Based on this, now several use cases can be applied using different functionalities from the two plugins. Some can be solved with both plugins – possibly with different additional features – while some are only supported by one plugin. However, this section aims to give a profound comparison and report on the functionalities of both plugins on the basis of those use cases. A general overview on the distribution of functionalities and use cases is provided by tables.

A. Use Case Scenarios

One of the most elemental use cases of verification is *structural consistency*, also called *model consistency* in [9]. The purpose of checking *structural consistency* is to determine whether it is possible to obtain a valid system state of the given model or not. This use case is supported by both plugins, but the USE ModelValidator uses the method from [8], while the SMT-based ModelFinder is based on [16].

Another use case called *property reachability* provides the possibility to restrict the resulting system state by further properties and to check, whether still a valid state can be found or not. Again, both plugins provide methods to load additional properties (i. e., invariants) for an existing model.

In contrast to *property reachability*, the use case *constraint implication* checks if the negation of an additional property or constraint still allows a valid system state of the model. If no valid system state can be found, this proves that the additional (non-negated) property was already implied by the original model. The USE ModelValidator offers a direct method to derive the desired information while for the SMT-based ModelFinder the given invariant has to be manually negated by the designer before using *property reachability* to finally retrieve a solution.

Determining whether specified properties of a model (i. e., invariants) are independent from or imply each other is called *constraint independence*. Both plugins provide a method to retrieve the information whether a certain invariant is dependent or not. The core idea of both plugins is based on [10]. However, the SMT-based ModelFinder goes a step further: It does not only check if an invariant is independent or not, but also analyzes which subset(s) of the other invariants imply a dependent invariant based on the work [17].

In order to give a profound overview of the possible system states of a model, *solution interval exploration* generates an enumeration of all valid system states. Furthermore, the USE ModelValidator does not only provide a method to iterate over the system states, it also categorizes them and, by this, blocks symmetric system states. On top of this, it is possible to define so-called *classifying terms* [11], such that the designer can choose constraints for equivalent classes. Compared to this, the SMT-based ModelFinder supports only the enumeration and does not support automatic blocking of symmetric states, which means, all blocking constraints have to be added manually.

Table I: Use case distribution.

Use case	SMT-based ModelFinder	USE ModelValidator
structural consistency	✓	✓
property reachability	✓	✓
constraint implication	only manually	✓
constraint independence	✓$^+$	✓
solution interval exploration	only manually	✓$^+$
partial solution completion	✓	✓
behavioral consistency	✓	filmstripping
concurrent operations	✓	✗
unsat contradiction analysis	✓$^+$	Kodkod
preconfigured system states	with GSP	bound config.

In order to retrieve certain classes of system states it is often useful to give a partially defined system state as starting point which shall automatically be completed by a solver to finally form a complete valid system state. This process is called *partial solution completion*. While SMT-based ModelFinder expects a partial initial system state as an additional input, the USE ModelValidator utilizes the system state that is present in and can be constructed with the USE tool. However, both approaches provide sufficient means to solve the problem.

Moving now from structural analysis to behavioral properties, the consistency over a sequence of system states is the matter of interest in the use case *behavioral consistency*. More precisely, it can be checked if either certain or any operations can be executed or if the system can get stuck in deadlock or livelock scenarios. While the USE ModelValidator does not directly support any behavioral checks, the designer can additionally load a USE plugin to transform the model into a so-called *filmstrip model*. After the transformation most of the behavioral checks can manually be formulated as structural consistency checks. However, deadlock checks are only possible with a lot of effort requiring expert knowledge about the filmstripping approach. The SMT-based ModelFinder innately provides methods to verify behavioral aspects of a model based on the ideas presented in [18]. The designer only has to provide more information within the problem bounds, e. g., how many system states should be analyzed. On top of this, the SMT-based ModelFinder also provides a method to analyze the impact of concurrent operation calls (cf. *concurrent operations*). This approach is based on [19].

So far, the detection of errors was covered, but afterwards, a debugging and finally correction of the errors is required. In order to find the erroneous part of the model, *unsat contradiction analysis* is used to retrieve the set of contradicting constraints preventing the instantiation of the model. Since the USE ModelValidator relies on *Kodkod*, it includes minimal unsat core extraction [20], i. e., it calculates exactly one minimal set of conflict clauses. However, the results remain at the Kodkod level and are not mapped backed to the model in USE. This leaves the designer with mapping the clauses back manually or interpreting the Kodkod result. In contrast, the SMT-based ModelFinder provides different possibilities for the contradiction analysis: One way would be to calculate candidates for the inconsistency as explained in [21], which are not necessarily complete. Another way is an extension of the former approach [22], [23], which calculates all minimal reasons for the contradiction. However, fixing the errors still must be done manually in both approaches based on the information the designer retrieved from the correspondingly used approach.

Finally, the long duration of the solving process remains a limiting factor in most cases. However, for some models the solving time can be significantly reduced using expert knowledge the designer has about the structure and behavior of his model. This last use case, *preconfigured system states*, is again provided by both plugins, but realized using two completely different ways. In contrast to a *partial solution completion*, this use case more or less restricts the number of possible values to assign. The USE ModelValidator provides a possibility to add such information at the model level within the problem bounds. This information is passed and used within the transformation such that unnecessary constraints in the SAT solver can be partially avoided. The SMT-based ModelFinder offers a concept called ground setting properties which restricts the values at the system state level, by this, much more constraints can be avoided and the scalability is better, cf. [24] for details.

B. Functionality

In addition to the differences in the set of supported use cases, also the internal functionality of both plugins differs significantly. In this subsection, the differences between the internal mechanisms and solving processes will be outlined. All of the following details are prodivded in Table II.

Starting with general UML modeling mechanisms, both plugins support *inheritance*. However, there are significant differences for associations: The SMT-based ModelFinder does only support binary associations (due to the usage of *EMF* without loading the UML metamodel) while the USE ModelValidator is capable of transforming and encoding n-ary associations. The USE ModelValidator also supports association classes while the SMT-based ModelFinder does not. However, the restriction to binary associations in the SMT-based ModelFinder does not decrease expressiveness, since it has been shown that models containing n-ary associations can be mapped into a semantically equivalent model solely composed of binary associations by adding a helping class and some invariants to the affected classes [25].

Having covered the general UML elements, now basic types are considered. Booleans, integers, and enums are supported as attribute types (and within OCL) in both plugins. Additionally, the USE ModelValidator provides a way to work on strings by defining a list of tokens. Conceptual ideas for the SMT-based ModelFinder have been proposed in [16]. Another way would be the use of an SMT string library.

Concerning the collection types, both plugins are supporting sets. The SMT-based ModelFinder also partially supports bags. However, concepts for all four collection types have been proposed in [26] for the SMT-based ModelFinder and in [27] for the USE ModelValidator. Nonetheless, these concepts are just theory and not implemented so far.

Table II: Functionality comparison.

Functionality	SMT-based ModelFinder	USE ModelValidator
Inheritance	✓	✓
Association	binary	n-ary
Association class	✗	✓
Boolean type	✓	✓
Integer type	✓	✓
Enum type	✓	✓
String type	✗	only as tokens
Set type	✓	✓
Bag type	partially	✗
OrderedSet, Sequence	concept ✓, impl. ✗	concept ✓, impl. ✗
underlying OCL logic	2-valued	3-valued
support of OCL	partially	nearly complete
selected OCL operations:		
collect	partially	✓
any	partially	one element
oclIsTypeOf	partially	✓
oclAsType	partially	✓
closure	✗	✓
frame conditions	2 automatic ways	manually

Moving now to the additional constraints formulated in OCL, the USE ModelValidator applies a 3-valued logic for the translation while the SMT-based ModelFinder currently uses only a 2-valued logic which is partially also 3-valued depending on the data type to be encoded. The support of OCL in the USE ModelValidator is nearly complete, but the SMT-based ModelFinder OCL support is restricted or partial. The latter comes from the fact that the encoding of an OCL expression depends on the exact context. For the SMT-based ModelFinder some OCL operations were selected in order to make the differences a bit clearer. Note that a 4-valued logic of OCL, i. e., full OCL support is planned as an option for the SMT-based ModelFinder to only be used if needed and, thus, keep the encoding as small as possible.

To clarify the OCL coverage in both plugins, some operations are investigated further. Concerning collections, the operation `any` is supported by both, but due the relational logic, the USE ModelValidator implementation only works if the condition limits the number of elements to zero or one, i. e., the result must be deterministic. In contrast, the SMT-based ModelFinder passes the freedom to actually chose "any" element to the precise encoding, but it can not be used in every context. The `oclIsTypeOf` and `oclAsType` are fully supported in the USE ModelValidator, while the general applicability in the SMT-based ModelFinder again depends on the context. The `closure` operation is only supported in the USE ModelValidator.

Finally, in behavioral verification, additionally frame conditions may have to be considered. Corresponding solutions for both, USE and the SMT-based ModelFinder have been proposed and are described detailedly in [7]. [28] gives the details about frame conditions in the SMT-based ModelFinder.

V. Conclusion and Future Work

Returning to the problem from the beginning, in this paper we highlighted the importance of a unified toolchain without the necessity of manual model transformations. Throughout the last sections, a tool was presented that combines the powers of several solving approaches and toolchains. As it is based on a general tool which was extended using plugins, we managed to get rid of all manual model transformations and, within this process, also the danger of errors due to these manual transformations. Still, depending on the use case and choice of verification plugin, some information has to be provided manually. However, this can be done in a much more convenient fashion and does usually not require expert knowledge about internal methods of the underlying model finders.

Essentially, the combination of USE with the SMT-based ModelFinder and the USE ModelValidator plugin offers a broad variety of sophisticated model verification approaches which can be easily combined and utilized by a designer even without a deeper knowledge about verification methods.

Regarding future research, adding more model verification tools to the USE environment might be a fruitful area. First of all, we would suggest to add *EMFtoCSP* since the current implementation already incorporates a transition to *EMF*. The addition would be easy to manage and open interesting possibilities for verification using CSP as the third great solving concept besides the already incorporated relational logic (Kodkod/Alloy) and SMT.

VI. Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SPECifIC under grant no. 01IW13001 and the project SELFIE under grant no. 01IW16001, the German Research Foundation (DFG) within the Reinhart Koselleck project under grant no. DR 287/23-1, the Graduate School SyDe funded by the German Excellence Initiative within the University of Bremen's institutional strategy, and the Siemens AG.

References

[1] D. Berardi, D. Calvanese, and G. De Giacomo, "Reasoning on UML class diagrams," *Artif. Intell.*, vol. 168, no. 1-2, pp. 70–118, 2005.

[2] M. Gogolla, F. Büttner, and M. Richters, "USE: A uml-based specification environment for validating UML and OCL," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 27–34, 2007.

[3] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A Challenging Model Transformation," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, 2007, pp. 436–450.

[4] D. Jackson, "Alloy: A new technology for software modelling," in *Tools and Algorithms for Construction and Analysis of Systems*, 2002, p. 20.

[5] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for Construction and Analysis of Systems*, 2007, pp. 632–647.

[6] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[7] F. Hilken, P. Niemann, M. Gogolla, and R. Wille, "Filmstripping and Unrolling: A Comparison of Verification Approaches for UML and OCL Behavioral Models," in *Tests and Proofs*, 2014, pp. 99–116.

[8] M. Kuhlmann and M. Gogolla, "From UML and OCL to relational logic and back," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, 2012, pp. 415–431.

[9] M. Gogolla and F. Hilken, "Model validation and verification options in a contemporary UML and OCL analysis tool," in *Modellierung 2016*, 2016, pp. 205–220.

[10] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, Independence and Consequences in UML and OCL Models," in *Tests and Proof*, 2009, pp. 90–104.

[11] M. Gogolla, A. Vallecillo, L. Burgueño, and F. Hilken, "Employing classifying terms for testing model transformations," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, 2015, pp. 312–321.

[12] F. Hilken, L. Hamann, and M. Gogolla, "Transformation of UML and OCL models into filmstrip models," in *Theory and Practice of Model Transformations, ICMT 2014*, 2014, pp. 170–185.

[13] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," 2016. [Online]. Available: https://www.SMT-LIB.org

[14] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for Construction and Analysis of Systems*, 2008, pp. 337–340.

[15] H. Riener, F. Haedicke, S. Frehse, M. Soeken, D. Große, R. Drechsler, and G. Fey, "metaSMT: focus on your application and not on solver integration," *International Journal on Software Tools for Technology Transfer*, pp. 1–17, 2016.

[16] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using Boolean satisfiability," in *Design, Automation and Test in Europe*, 2010, pp. 1341–1344.

[17] N. Przigoda, R. Wille, and R. Drechsler, "Leveraging the Analysis for Invariant Independence in Formal System Models," in *Euromicro Conference on Digital System Design*, 2015, pp. 359–366.

[18] M. Soeken, R. Wille, and R. Drechsler, "Verifying Dynamic Aspects of UML models," in *Design, Automation and Test in Europe*, 2011, pp. 1077–1082.

[19] N. Przigoda, C. Hilken, R. Wille, J. Peleska, and R. Drechsler, "Checking concurrent behavior in UML/OCL models," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, 2015, pp. 176–185.

[20] E. Torlak, F. S.-H. Chang, and D. Jackson, "Finding Minimal Unsatisfiable Cores of Declarative Specifications," 2008, pp. 326–341.

[21] R. Wille, M. Soeken, and R. Drechsler, "Debugging of inconsistent UML/OCL models," in *DATE*, 2012, pp. 1078–1083.

[22] N. Przigoda, R. Wille, and R. Drechsler, "Contradiction analysis for inconsistent formal models," in *18th IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS 2015, Belgrade, Serbia, April 22-24, 2015*. IEEE Computer Society, 2015, pp. 171–176.

[23] ——, "Analyzing inconsistencies in UML/OCL models," *Journal of Circuits, Systems, and Computers*, vol. 25, no. 3, 2016.

[24] ——, "Ground Setting Properties for an Efficient Translation of OCL in SMT-based Model Finding," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, 2016, pp. 261–271.

[25] M. Gogolla and M. Richters, "Expressing UML Class Diagrams Properties with OCL," in *Object Modeling with the OCL*, 2002, pp. 85–114.

[26] M. Soeken, R. Wille, and R. Drechsler, "Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models," in *Tests and Proof*, 2011, pp. 152–170.

[27] M. Kuhlmann and M. Gogolla, "Strengthening SAT-Based Validation of UML/OCL Models by Representing Collections as Relations," in *Modelling Foundations and Applications - 8th European Conference, ECMFA*, 2012, pp. 32–48.

[28] N. Przigoda, J. Gomes Filho, P. Niemann, R. Wille, and R. Drechsler, "Frame Conditions in Symbolic Representations of UML/OCL Models," in *Int'l Conf. Formal Methods and Models for System Design*.