Determining Relevant Model Elements for the Verification of UML/OCL Specifications

Julia Seiter[1] Robert Wille[1] Mathias Soeken[1,2] Rolf Drechsler[1,2]

[1]Group for Computer Architecture, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, D-28359 Bremen, Germany
{jseiter,msoeken,rwille,drechsle}@informatik.uni-bremen.de

***Abstract*—Modeling languages such as UML or SysML received significant attention over the last years. They allow for an abstract description of systems already in the absence of a precise implementation or a hardware/software partitioning. Additionally considering textual constraints, for example provided by means of OCL, enables to automatically check the specified systems e.g. for consistency of the structure or reachability of certain system states. However, for the majority of verification tasks, not the entire model has to be considered.**

In this work, we propose an approach that automatically determines reduced system models, i.e. system descriptions that only include model elements which are relevant for the considered verification task. Considering reduced models eases the access by the designer and supports incremental design and verification schemes. But most important, they improve the efficiency of the applied formal verification engine. Experiments demonstrate that already small reductions in the model lead to significant accelerations in the run-time of the verification engine.

I. Introduction

In the past, various approaches for verification of UML/OCL (*Unified Modeling Language* [1], *Object Constraint Language* [2]) models have been developed providing the possibility to ensure correctness of a system in an early design stage [3]–[5]. However, proving correctness of a UML/OCL model is a difficult task [6] and employing formal methods to a UML/OCL model often requires a significant amount of run-time. Even small models can be difficult to verify when the consideration of dynamic behavior results in a large number of possible system states to be checked.

However, for the majority of models and verification tasks, it is not necessary to consider the complete model – in particular when models are composed modularly and verification tasks consider local aspects only. In these cases, it is sufficient to perform the verification on a reduced model containing only those elements which are relevant to the task at hand. Especially in large models, each attribute is often only affected by a subset of invariants and operations as many components are just locally relevant.

In [7], a first approach to model reduction for the purpose of verification, based on the idea of *slicing* presented in [8], has been discussed. The authors propose to partition the model into several sub models depending on the influence of the invariants. The single slices are verified individually and the results are merged such that correctness of all slices induces correctness of the original (complete) model. Although this approach has proven to be effective in order to reduce the time required for verification, it is not applicable for the verification of dynamic behavior specified by UML or SysML models since only invariants have been considered. As a result, only verification tasks such as consistency can be conducted while no support for tasks such as the executibility of a method or the reachability of a good/bad state is provided.

In this paper, we propose a generic approach to model reduction which is applicable to both, static and dynamic aspects, as well as its corresponding verification tasks. Starting from an empty model and a set of model elements derived from the verification task, elements originating from the complete model and, in fact, being relevant to the considered verification task are iteratively added to the empty model. This eventually results in a reduced model which still is sufficient to perform the respective checks.

As a result, the obtained model is smaller and, due to the reduced number of model elements to be inspected, easier to access by the designer. Furthermore, the proposed approach supports an incremental design and verification scheme as the designer is enabled to verify only those parts of the model which recently have been changed or added. Finally, also the applied verification engines profit from the reduced model: Since fewer model elements have to be considered, the run-time of the corresponding engines are improved. This also has been experimentally evaluated. Our results show that the consideration of a reduced model sometimes even enables to solve verification tasks which could not be handled before due to complexity reasons.

978-3-9815370-0-0/DATE13/©2013 EDAA

II. Background and Notation

UML/OCL enables to formally specify the structure and the behavior of systems. UML *class diagrams* are usually the medium of choice to represent the structure of a system. The main component of a class diagram is a *class* that describes an entity of the model. A class itself consists of *attributes* and *operations*, where attributes describe the information which is stored by the class and operations define possible actions that can be executed in order to change the attributes' values. Classes can be set into relation via *associations*. The type of a relation is expressed by *multiplicities* that are added to each *association-end*.

Example 1. *Throughout the paper we are making use of a running example as outlined by means of Fig. 1. It represents a fragment of a smartphone on which apps can be executed. Both, software components (i.e. the apps) and hardware components (i.e. the microphone and the speaker) are specified. Three apps on the phone allow to make a telephone call, to write a text message, and to play music.*

The phone, the apps, and the hardware are represented by individual classes which have attributes and operations depending on their purpose. The phone is connected to all other classes by means of 1*-to-*1 *associations.*

In order to express further properties or restrictions as well as the behavior of a model, textual constraints provided by OCL [2] can be added to a model. OCL conditions may appear as both, *invariants* or *pre-* and *post-conditions* of operations. Invariants are global constraints that must be satisfied by all system states. Pre- and post-conditions are considered only locally in the context of an operation call. More precisely, an operation can only be invoked if its corresponding pre-condition is satisfied. Afterwards, the following system state needs to match the operation's post-condition.

Example 2. *According to the system description of the smartphone device in Fig. 1, the operation* CallingApp::placeCall *can only be invoked if the user is not already on a call (denoted by* not inCall*) and still has more than 10 units of credits left (denoted by* phone.credit >= 10*). After the execution of this operation, a system state must be reached where* inCall *is set to* true*.*

In the remainder of this paper, UML/OCL models are formally denoted as follows: A UML/OCL *model* is a tuple $m = (C, R)$ composed of a set of classes C and a set of associations R.

A *class* $c = (A, O, I) \in C$ of a model m is a tuple composed of attributes $A \subseteq V$, operations O, and invariants $I \subseteq \Phi$. The set V denotes all *variables* of the model, whereas the set Φ refers to all *OCL conditions* of the model. An OCL condition $\varphi \in \Phi$ is defined as a function $\varphi : V' \to \mathbb{B}$ with its *domain* $\mathrm{dom}(\varphi) = V' \subseteq V$ being a

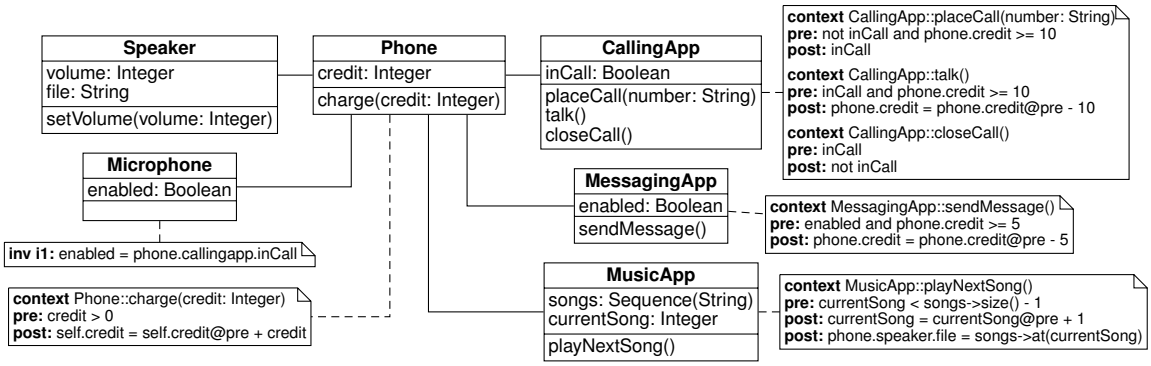

Fig. 1. UML/OCL model of a smartphone specification

subset of the model's variables. Furthermore, an *operation* $o \in O$ is defined as a tuple $o = (P, \lhd, \rhd)$ with a set of parameters $P \subseteq V$ and pre- and post-conditions $\lhd \subseteq \Phi$ and $\rhd \subseteq \Phi$, respectively.

An n-ary *association* $r \in R$ of a model m is a tuple $r = (r_{\text{ends}}, r_{\text{mult}})$ with *association ends* $r_{\text{ends}} \in C^n$ for a given set of classes C and *multiplicities* $r_{\text{mult}} \in (\mathbb{N}_0 \times \mathbb{N})^n$ that is defined as a range with a *lower bound* and an *upper bound*. Associations ends are also variables of the model, i.e. $r_{\text{ends}} \subseteq V$.

III. Consideration of Relevant Model Elements

UML/OCL provides a *formal* description of a system to be realized. As a result, the structure, the behavior, and properties of the described system can be checked for correctness already in the absence of a precise implementation. *Consistency checks* [9] are a prominent example. Various independently added constraints (e.g. invariants) may lead to contradictions in the system which prohibits any valid system state to be instantiated. Consistency checkers such as proposed in [10], [11] can automatically determine whether a model is consistent or not. Besides that, also *verification tasks* that consider the dynamic aspects of a model [3], [4] are being considered. Here, it is checked whether certain states (e.g. bad states, good states, deadlock states, etc.) are reachable according to the formal description of the system behavior through pre- and post-conditions of operations.

However, existing approaches usually consider the entire model when addressing the respective verification tasks. This is crucial since the automatic formal verification of UML/OCL models is a very complex and time consuming task. In fact, the search space and, thus, the run-time of the respective solve engines often grows exponentially with respect to the model size. In contrast, for many verification tasks not the entire model has to be considered.

Example 3. *Assume that, for the formal system specification from Fig. 1, it should be verified whether the operation* MusicApp::playNextSong *can ever be executed. For this, it is sufficient to show that from a given initial state a system state meeting the operation's pre-conditions can be reached, i.e. a typical reachability problem is addressed. However, not the entire model from Fig. 1 needs to be considered. Obviously, the attributes used in the pre-condition, i.e.* currentSong *and* songs*, are relevant for this case. But since they are neither affected by any other operation nor an invariant, all remaining model elements can be discarded. Hence, a consideration of these two attributes as well as its corresponding class* MusicApp *only is sufficient to solve this verification task.*

Considering only relevant parts of a UML/OCL model during the verification has some obvious benefits:

- A smaller model size accelerates the applied solving engines since instead of the entire model only the relevant sub model is considered.
- Manual interactions of the design team with the model are improved. If e.g. a design flaw has been detected, the team can focus on the affected parts of the model and are not being misled by the remaining parts. This can also be exploited for automatic debugging methods such as [12].
- Incremental design and verification schemes are improved. New elements can subsequently be added to the model. In each iteration, only the newly added model elements as well as elements affected by them have to be checked for correctness.

In order to ensure completeness, *all* relevant model elements need to be considered. Depending on the size of the model as well as on the considered verification task, determining the relevant elements is a non-trivial task.

Example 4. *Assume now that it should be verified whether the operation* CallingApp::placeCall *can ever be executed. To meet the corresponding pre-condition, the values of the attribute* inCall *of the same class needs to be set to* false*. Furthermore, also the attribute* credit *of the connected phone must have a certain value. Consequently, the classes* CallingApp *and* Phone *with their attributes as well as all invariants and operations whose post-conditions include these attributes are relevant for the verification task. Moreover, also the classes* Microphone *and* MessagingApp *and their attribute* enabled *need to be considered as they affect the value of* inCall *and* credit *and, thus, are also relevant to the verification task. In contrast, e.g. the classes* MusicApp *and* Speaker *can entirely be ignored in this case.*

Already in this simple example, all dependencies are not obvious at a first glance. Hence, a structured approach which completely determines all relevant model elements for a given model and verification task is introduced in the next section.

IV. Determining Relevant Model Elements

In this section, we describe the proposed reduction method by means of Algorithm 1 with its inputs being the original model $m = (C, R)$ with classes C and associations R as well as the considered verification task τ. From this model a reduced model $m' = (C', R')$ is determined by following a bottom-up scheme: While starting with an empty model at the beginning, the algorithm subsequently enriches it with UML/OCL model elements that need to be considered to solve the considered verification task.

The procedure includes the execution of the following steps:

1) *Initialize (Lines 1–6):* Initializes the empty model $m' = (C', R')$ to be filled as well as the required auxiliary data-structures and determines the initial UML/OCL model elements relevant to the verification task.
2) *Add classes (Lines 7–19):* Adds UML/OCL classes to the reduced model that have been identified as relevant to the verification task and have not yet been considered. For this, all classes C of the original model are traversed (see Line 9). In each iteration, it is checked whether
   - attributes (Lines 10–11),
   - invariants (Lines 12–14), or
   - operations (Lines 15–18)

   have to be added. If so, the respective model element is added to the reduced model. At the same time, it is checked whether these additions imply the consideration of further UML/OCL model elements.

Input: model $m = (C, R)$, verification task τ
Output: reduced model $m' = (C', R')$

1 $C' \leftarrow \emptyset$, $R' \leftarrow \emptyset$ *Initialize*
2 **foreach** $i \in \{0, \ldots, |C| - 1\}$ **do**
3 $c' \leftarrow (\emptyset, \emptyset, \emptyset)$
4 $C' \leftarrow C' \cup \{c'\}$
5 $D \leftarrow$ deriveModelElements(τ)
6 $D_{\text{visited}} \leftarrow \emptyset$
7 **foreach** $v \in D \setminus D_{\text{visited}}$ **do** *Classes*
8 $D_{\text{visited}} \leftarrow D_{\text{visited}} \cup \{v\}$
9 **foreach** $c_i = (A_i, O_i, I_i) \in C$ **do**
10 **if** $v \in A_i$ **then** *Attributes*
11 $A'_i \leftarrow A'_i \cup \{v\}$
12 **foreach** $\varphi \in I_i$ with $v \in \text{dom}(\varphi)$ **do** *Invariants*
13 $I'_i \leftarrow I'_i \cup \{\varphi\}$
14 $D \leftarrow D \cup \text{dom}(\varphi)$
15 **foreach** $o = (P, \lhd, \rhd) \in O_i$ **do** *Operations*
16 **foreach** $\varphi \in \rhd$ with $v \in \text{dom}(\varphi)$ **do**
17 $O'_i \leftarrow O'_i \cup \{o\}$
18 $D \leftarrow D \cup \text{dom}(\lhd) \cup \text{dom}(\varphi)$
19 $C' \leftarrow \{c' \in C' \mid c' \neq (\emptyset, \emptyset, \emptyset)\}$
20 **foreach** $r = (r_{\text{ends}} = (c_1, \ldots, c_n), r_{\text{mult}}) \in R$ **do** *Associations*
21 **if** $(c_1 \in D) \vee \cdots \vee (c_n \in D)$ **then**
22 $R' \leftarrow R' \cup \{r\}$
23 **foreach** $c_i \in \{c_1, \ldots, c_n\}$ with $c'_i \notin C'$ **do**
24 $c' \leftarrow (\emptyset, \emptyset, \emptyset)$, $C' \leftarrow C' \cup \{c'\}$
25 **if** $\{c_1, \ldots, c_n\} \subseteq C'$ **then**
26 $R' \leftarrow R' \cup \{r\}$

Algorithm 1: Proposed algorithm

3) *Add associations (Lines 20–26):* Checks, depending on the classes and the invariants that eventually remained in the reduced model, which associations of the original model have to be considered to solve the verification task. These associations are then added to the reduced model.

In the following, the respective steps are outlined in detail.

A. Initialize

Following the bottom-up scheme, an empty model $m' = (C', R')$ is initialized first (see Line 1). The classes are initialized as "empty shells", i.e. each class is added but being initialized with no attributes, invariants, or operations whatsoever (see Line 2-4).

Besides that, auxiliary data structures are introduced which are required later:

- The set D (for *to deduce*; initialized in Line 5) stores model elements which have been identified as relevant for the considered verification task and, thus, have to be added to the reduced model. Furthermore, these model elements might imply the addition of further elements to be considered. This needs to be deduced. Initially, the respective elements are derived from the given verification task.
- The set D_{visited} (initialized in Line 6) stores UML/OCL model elements from which possible implications already have been deduced. By this, it is ensured that elements which have been added to D more than once are deduced just one single time.

Example 5. *In order to generate a reduced model for the verification task as described in Example 4 (i.e. the executability of* callingApp::placeCall*), the reduced model $m' = (C', R')$ is initialized as described above and D_{visited} is initialized empty. Finally, the set D of model elements from which further implications may be deduced is determined. For the considered verification task, all elements of the pre-condition of* callingApp::placeCall*, i.e. the attributes* CallingApp::inCall*,* Phone::credit*, and the association end* CallingApp::phone *are identified as relevant. Hence, D is set to* {CallingApp::inCall, Phone::credit, CallingApp::phone}.

B. Add Classes

In the next step, class elements identified as relevant to the verification task are added to the reduced model and further implications are deduced. For this purpose, each model element v in the set D is considered (except the already visited ones stored in D_{visited}, see Line 7). Each model element is flagged visited in Line 8. Then, all classes $c_i = (A_i, O_i, I_i) \in C$ of the original model are traversed (Line 9). In each iteration, it is checked whether attributes, invariants, or operations from the original class c_i also have to be considered in the corresponding class c'_i of the reduced model.

1) Add Attributes: In case of attributes, the respective check is simple. If the currently considered model element v (which already has been identified as relevant to the verification task) is an attribute of the original class c_i (i.e. if $v \in A_i$), then v is also added to the class c'_i of the reduced model (i.e. v is added to A'_i, see Line 11).

Example 6. *In the example, the attributes* CallingApp::inCall *and* Phone::credit *are considered and all classes of the original model are traversed. Since these attributes belong to the classes* CallingApp *and* Phone *of the original model, these attributes are added to the corresponding classes* CallingApp′ *and* Phone′ *in the reduced model, respectively.*

2) Add Invariants: Invariants are added to the reduced model in a similar fashion. All invariants $\varphi \in I_i$ of the currently considered class c_i whose domain $\text{dom}(\varphi)$ contains the currently considered model element v are traversed (see Line 12). These invariants are added to the corresponding class c'_i of the reduced model (i.e. φ is added to I'_i, see Line 13). Besides that, the invariant φ may imply the addition of further UML/OCL model elements to the reduced model. In fact, all model elements within the domain of φ may affect the value of the currently considered element v. As a consequence, all these elements are relevant for the considered verification task and, hence, need to be added to the reduced model. This is ensured by adding all elements $\text{dom}(\varphi)$ to the set D (see Line 14).

Example 7. *To exemplarily illustrate this step, just consider the attribute* CallingApp::inCall*. Since it is restricted by the invariant* Microphone::i1*, this invariant is also added to the set of invariants for class* Microphone′ *in the reduced model. The currently deduced model elements D are enriched by all model elements in the domain of that invariant, i.e. $D \leftarrow D \cup$ {*Microphone::enabled*}.*

3) Add Operations: Finally, each operation $o = (P, \lhd, \rhd)$ of each class c_i is considered (Line 15). Here, particularly the post-conditions are of interest. An operation has to be added to the reduced model, only if it affects a UML/OCL model element (v in this case) that already has been identified as relevant for the verification task. This is the case when v is part of the post-condition $\rhd$. Hence, each constraint φ in the post-condition $\rhd$ of the currently considered operation is checked (see Line 16). If v is in the domain of such a constraint (i.e. if $v \in \text{dom}(\varphi)$), the operation is added to the respective class c'_i (see Line 17). Besides that, also here implications requiring the addition of further model elements need to be checked. In fact, in this case, the value of v directly depends on the domain $\text{dom}(\varphi)$ of the constraint $\varphi \in \rhd$. Since additionally the operation o is only invoked if the pre-condition $\lhd$ holds, also the domain of $\text{dom}(\lhd)$ of the pre-condition $\lhd$ has to be further considered. Hence, all these model elements are added to D (see Line 18).

Example 8. *Consider for example the attribute* credit *of the class* Phone*. Since this attribute is restricted by the post-condition of the operation* MessagingApp::sendMessage*, this operation is also added to the set of operations for class* MessagingApp′*. A new element is deduced as the domain of the operation's pre-condition contains* MessagingApp::enabled*.*

All classes for which no component has been added remained empty shells and can be dropped (Line 19). It remains the consideration of the required associations R' between these classes.

TABLE I
EXPERIMENTAL EVALUATION

Benchmark	Verification task	Without reduction						With reduction					
		Attr.	Inv.	Op.	Cond.	Assoc.	Run-time	Attr.	Inv.	Op.	Cond.	Assoc.	Run-time
Traffic light	Exec.	4	3	3	15	1	>5000	3	2	3	15	0	0.21
	Exec.	4	3	3	15	1	15.88	3	2	3	15	0	11.19
	Exec.	4	3	3	15	1	0.54	3	2	3	15	0	0.31
	Reach.	4	3	3	15	1	>5000	1	1	0	0	0	0.00
	Reach.	4	3	3	15	1	>5000	3	2	3	15	1	0.14
	Reach.	4	3	3	15	1	>5000	3	2	3	15	1	0.04
Memory	Exec.	6	2	5	26	2	>5000	5	1	5	26	2	0.28
	Exec.	6	2	5	26	2	>5000	5	1	5	26	2	0.29
	Exec.	6	2	5	26	2	>5000	5	1	5	26	2	0.26
	Exec.	6	2	5	26	2	>5000	5	1	5	26	2	0.26
	Exec.	6	2	5	26	2	>5000	5	1	5	26	2	0.27
	Reach.	6	2	5	26	2	>5000	5	1	5	26	2	0.21
	Reach.	6	2	5	26	2	>5000	1	1	0	0	0	0.00
	Reach.	6	2	5	26	2	>5000	5	1	5	26	2	0.29
CPU	Exec.	10	4	5	40	6	19.65	8	4	5	34	6	18.68
	Exec.	10	4	5	40	6	5.44	8	4	5	34	6	5.41
	Exec.	10	4	5	40	6	5.59	8	4	5	34	6	5.49
	Exec.	10	4	5	40	6	5.60	8	4	5	34	6	5.41
	Exec.	10	4	5	40	6	5.49	8	4	5	34	6	5.41
	Reach.	10	4	5	40	6	5.71	8	4	5	34	6	5.58
	Reach.	10	4	5	40	6	5.72	8	4	5	34	6	5.41
	Reach.	10	4	5	40	6	5.90	8	4	5	34	6	5.35

C. Add Associations

To determine R' two steps need to be performed. The first step involves the addition of association due to references by association ends that have been deduced and are stored in D. That is, whenever an association end is in D, then the respective association is added to R' (Line 22). In addition all classes that are referred to by this association but are not present in the reduced model, i.e. they are not included in C', need to be added back as empty shell (Line 23–24). Afterwards, all associations whose association ends are contained completely in C' are also added to R' (Line 25–26).

Example 9. *The resulting reduced model contains the classes* Phone*,* CallingApp*,* MessagingApp *and* Microphone *with all their attributes, invariants, operations and pre- and post-conditions as well as the associations between all of these classes as depicted in the original model.*

By applying this algorithm, a smaller and reduced system model results that only includes those elements that definitely have to be considered in order to address the respective verification task. As a result, the benefits discussed in Section III can be exploited. In particular, the run-time of the underlying solving engines can significantly be improved as shown by the experimental evaluation in the next section.

V. EXPERIMENTAL EVALUATION

The algorithm described in Section IV has been implemented in Java and evaluated by applying it to several UML/OCL models. As test cases, system specifications describing a traffic light preemption (denoted by *Traffic Light*), a memory controller (denoted by *Memory*), and a central processing unit (denoted by *CPU*) have been considered. The executability of each operation in the respective models and the reachability of three different system states have been considered as verification tasks. As verification engine, the approach presented in [3] has been applied. The experiments have been conducted on a Linux 3.4 machine with a 2.4 GHz Intel Core i7 and 8 GB main memory. The time-out has been set to 5000 seconds.

Table I shows the results of our evaluation. The first two columns denote the name of the system model as well as the considered verification task. Afterwards, the number of attributes (denoted by *Attr.*), invariants (denoted by *Inv.*), operations (denoted by *Op.*), pre- and post-conditions (denoted by *Cond.*), and associations (denoted by *Assoc.*) are provided for both, the original model and the reduced model obtained with the proposed algorithm. The resulting run-time (in CPU seconds) needed by the verification engine is respectively provided in the columns denoted by *Run-time*.

In all considered cases, it was possible to reduce the model. Furthermore, the run-time has been improved significantly for several tests. Without reduction, verification was limited by the applied time-out in numerous cases. In contrast, after the obtained reduction, the verification task could be solved easily within just a few seconds or less. The run-time improvement depends not only the number of removed model elements, but also on the particular element which has been removed. This leads to the effect that the removal of a few elements can already result in a significant reduction of the run-time. Apparently, the removal of model elements changes the search space in such a way that the respective verification tasks can be performed much faster. As a result, a small model reduction often is sufficient to improve the run-time of the verification significantly.

VI. CONCLUSION

In this work, we presented an algorithm for determining relevant model elements for the verification of UML/SysML specifications. Considering a reduced model leads to several benefits. Due to the reduced number of model elements to be inspected, the resulting model is smaller and, thus, easier to access by the designer. Incremental design and verification schemes are supported as only those parts of the model have to be verified which have recently been changed or added. Moreover, the verification engines itself profit from the consideration of a reduced model. As shown by an experimental evaluation, even small reductions in the model may lead to significant improvements in the run-time of the verification engine.

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.

REFERENCES

[1] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Essex, UK: Addison-Wesley Longman, Jan. 1999.

[2] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*. Boston, MA, USA: Addison-Wesley Longman, Mar. 1999.

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

[4] J. Cabot, R. Clarisó, and D. Riera, "Verifying UML/OCL Operation Contracts," in *Integrated Formal Methods*, ser. Lecture Notes in Computer Science, M. Leuschel and H. Wehrheim, Eds., vol. 5423. Springer, Feb. 2009, pp. 40–55.

[5] 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*. Springer, Oct. 2007, pp. 436–450.

[6] D. Berardi, D. Calvanese, and G. D. Giacomo, "Reasoning on UML Class Diagrams," *Artif. Intell.*, vol. 168, no. 1–2, pp. 70–118, Oct. 2005.

[7] A. Shaikh, R. Clarisó, U. K. Wiil, and N. Memon, "Verification-driven Slicing of UML/OCL Models," in *IEEE/ACM Int'l. Conf. on Automated Software Engineering*, Sep. 2010, pp. 185–194.

[8] M. Weiser, "Program Slicing," in *Int'l. Conf. on Software Engineering*, Mar. 1981, pp. 439–449.

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

[10] 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*, Mar. 2010, pp. 1341–1344.

[11] J. Cabot, R. Clarisó, and D. Riera, "Verification of UML/OCL Class Diagrams Using Constraint Programming," in *IEEE Int'l. Conf. on Software Testing Verification and Validation Workshop*, Apr. 2008, pp. 73–80.

[12] R. Wille, M. Soeken, and R. Drechsler, "Debugging of Inconsistent UML/OCL Models," in *Design, Automation and Test in Europe*, Mar. 2012, pp. 1078–1083.