Verifying UML/OCL Models Using Boolean Satisfiability

Mathias Soeken[1] Robert Wille[1] Mirco Kuhlmann[2] Martin Gogolla[2] Rolf Drechsler[1]
[1]Group for Computer Architecture, University of Bremen, 28359 Bremen, Germany
[2]Database Systems Group, University of Bremen, 28359 Bremen, Germany
{msoeken,rwille,mk,gogolla,drechsle}@informatik.uni-bremen.de

***Abstract*—Nowadays, modeling languages like UML are essential in the design of complex software systems and also start to enter the domain of hardware and hardware/software co-design. Due to shortening time-to-market demands, "first time right" requirements have thereby to be satisfied. In this paper, we propose an approach that makes use of Boolean satisfiability for verifying UML/OCL models. We describe how the respective components of a verification problem, namely system states of a UML model, OCL constraints, and the actual verification task, can be encoded and afterwards automatically solved using an off-the-shelf SAT solver. Experiments show that our approach can solve verification tasks significantly faster than previous methods while still supporting a large variety of UML/OCL constructs.**

I. Introduction

Modeling languages like the *Unified Modeling Language* (UML) have been established to specify the requirements and the design of software systems [13]. Moreover, due to the increasing complexity of nowadays hardware systems, researchers also started to investigate the application of UML in the design of integrated circuits [11]. In the context of hardware/software co-design, a whole system is specified first. Then, the respective partitioning into software and hardware parts is done after first performance checks on a prototypical implementation have been carried out. For all these purposes, UML models provide an appropriate level of abstraction hiding concrete implementation details but being expressive enough to specify a complex system. Additionally, the *Object Constraint Language* (OCL) is typically used to extend UML models by textual constraints. OCL constraints enable the specification of complex system requirements by defining properties and relations between the respective parts of a model.

Detecting design flaws in early stages of development is very important, since adapting an abstract UML model is cheaper than making changes within a final product. In particular, due to shortening time-to-market demands, this requires computer-aided approaches for validating or even verifying a system specification in context of UML and OCL.

As one step in this direction, the *UML-based Specification Environment* (USE) provides well-established methods that can be applied e.g. to automatically generate test cases for the respective models [7]. However, most of these methods are based on enumeration, i.e. in the worst case USE requires a traversal of the complete search space. As a result, for large models the approach runs into complexity problems.

To face quality assurance problems, in the last years researchers began to exploit formal methods for the verification of UML models. Approaches based on theorem provers like PVS [9] and HOL-OCL/Isabelle [3] have been applied. However, these approaches usually need manual interaction. Besides that, they often require a strong formal background of the designer.

As a consequence, researchers started to investigate the application of fully automatic proof engines. A method based on *Constraint Satisfaction Problems* (CSP) has been introduced in [4]. Other CSP methods [10] as well as description logic [2], [14] have been considered, but cannot be applied to the verification of UML models since they do not support OCL constraints. In [12], the CQC method is used for verifying UML/OCL models, but the method requires a manual translation. Another approach is based on Alloy [8], a modeling language based on relational logic. In [1] it has been shown how UML verification tasks can be transformed into this language. Afterwards, the Alloy analyzer can be applied to solve the problem. However, as already discussed in [1], UML and Alloy are using completely different design philosophies so that many UML/OCL constructs cannot be supported.

In this paper, we propose an alternative automatic approach for verifying UML/OCL models that supports a large variety of UML/OCL constructs (as e.g. in USE) while still exploiting formal methods to accelerate the verification process (as e.g. in Alloy). Techniques of *Boolean satisfiability* (SAT) are thereby applied. In the past, SAT already has been successfully applied e.g. in the domain of electronic design automation [6]. In this work, we transfer these achievements to the verification of UML/OCL models. We show how UML system states, OCL constraints, and the respective verification tasks can be encoded as a SAT instance. Well-developed off-the-shelf SAT solvers (e.g. [5]) are applied to efficiently solve the resulting SAT instance. Thus, UML models can be checked significantly faster than by enumerative methods, while still a large variety of OCL constraints is supported. This is also experimentally confirmed by applying different verification tasks to UML/OCL models. More precisely, consistency and independence can be checked magnitudes faster in comparison to USE. In comparison to Alloy, more complex verification tasks can be tackled.

II. Background

A. UML/OCL, Class/Object Diagrams, and OCL

Classes and *associations* represent the main constructs in a class diagram. Classes describe what information can be handled within the modeled system and how the information is structured. *Attributes* define the single data elements.

Furthermore, a class can contain *operations* as well as OCL constraints for describing its behavior. In our approach we focus on OCL *invariants*, i.e. constraints restricting the set of valid system states by enforcing specific system properties.

A *model* specified in terms of a class diagram can be seen as a template for creating a concrete *system state* complying to the specification. System states are illustrated by object diagrams. Each element in an object diagram has a corresponding counterpart in the class diagram. In other words: an *object* is an instantiation of a specific class holding values for each class attribute (at a particular point in time). A *link* connecting objects is an instantiation of an association. A system state can comprise any number of objects and links.

In the remainder of this paper, the set of classes specified in a UML model is denoted by $CLASS$. Let $c \in CLASS$ be

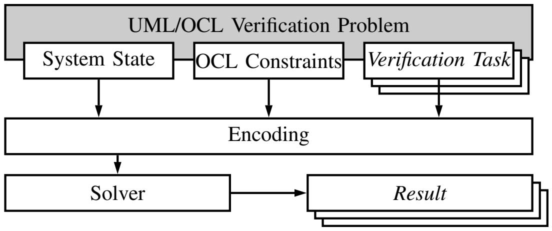


Fig. 1. General flow for formal verification of UML/OCL models

a class. Then ATT_c is the set of attributes of that class and $\mathrm{oid}(c)$ is the set of objects in a corresponding system state.

A model is *consistent*, if there exists a non-empty system state which satisfies all defined OCL invariants, i.e. no defined system property contradicts the invariants. An invariant i is said to be *independent* when no other invariants in the model imply i. That is, the invariant i adds essential information to the model. Checking consistency and independence are typical verification tasks that are formally defined in [7].

B. Boolean Satisfiability

The *Boolean satisfiability problem* (SAT problem) is to determine a satisfying assignment for a given Boolean function f or to prove that no such assignment exists. In general, the Boolean function f is given in *Conjunctive Normal Form* (CNF), i.e. a product-of-sum representation.

In the past, several backtracking algorithms solving the SAT problem have been proposed [5] (so called *SAT solvers*). A SAT solver traverses the search space until a satisfying assignment has been found or it can be deduced that no such assignment is possible.

For sake of a better readability, in the remainder of the paper SAT variables and clauses are formulated in terms of bit-vectors and bit-vector expressions. Converting them to a SAT instance consisting of Boolean variables and clauses is straight-forward [15].

III. SAT-based Verification of UML/OCL Models

In this section, the main idea of SAT-based verification of UML/OCL models is presented. Usually, the verification problem consists of three components: system states of a UML model, OCL constraints (defining e.g. properties of the model), and the respective verification task (e.g. checking for consistency, independence). In order to solve a verification problem using existing solve engines, the flow depicted in Fig. 1 is applied. First the respective components are encoded. Afterwards, the resulting instance is passed to a solve engine which is used as a black box. As a result, either unsatisfiable or satisfiable is returned. In case of satisfiable, additionally an assignment to all variables of the encoding is given from which e.g. a system state can be obtained.

As can be seen in Fig. 1, the respective components of a UML/OCL problem are encoded independently of each other. That is, if several verification tasks are applied, only the encoding for the respective task has to be replaced. Using this flow, several problems occurring in the domain of UML system modeling can be tackled. We thereby apply a solver of Boolean satisfiability to solve the respective instances. In the past, these kind of solve engines already have been shown to be efficient in several domains like e.g. electronic design automation [6].

It should thereby be noted that OCL constraints in general may be undecidable. However, decidability in OCL is achieved by defining a finite solution space, i.e. considering finite bounds for the number of objects as well as for the domains of the attributes. These restrictions are reasonable since at least for the concrete implementation finite bounds are applied.

IV. SAT Encoding

In this section, the technical contribution of this work is introduced. That is, the concrete SAT encoding needed to implement SAT-based verification of UML/OCL models is described. We distinguish between the three components already introduced in Fig. 1, namely the encoding of system states, OCL constraints, and the respective verification tasks, respectively. The UML/OCL model given in Fig. 2(a) is used as a running example throughout the whole section.

A. Encoding System States of the UML Model

Verification of UML models includes – among other aspects – the proof of consistency and independence, i.e. system states can consistently be generated and no OCL constraint is redundant. In all these cases, a system state must be created. To formulate system states of the UML model, an encoding of objects (in particular of their respective attribute assignments) as well as of the links between these objects is required.

Attribute assignments are thereby represented as follows: Let $a \in ATT_c$ be an attribute of a class $c \in CLASS$. Then, the assignment to this attribute for the object $o \in \mathrm{oid}(c)$ is encoded by the bit-vector $\vec{\alpha}_a^o \in \mathbb{B}^k$ with $k = \lceil \mathrm{ld}(n+1) \rceil$. Here, n denotes the cardinal number of the domain of a. The extra value is needed to allow an undefined value (i.e. $\perp$).

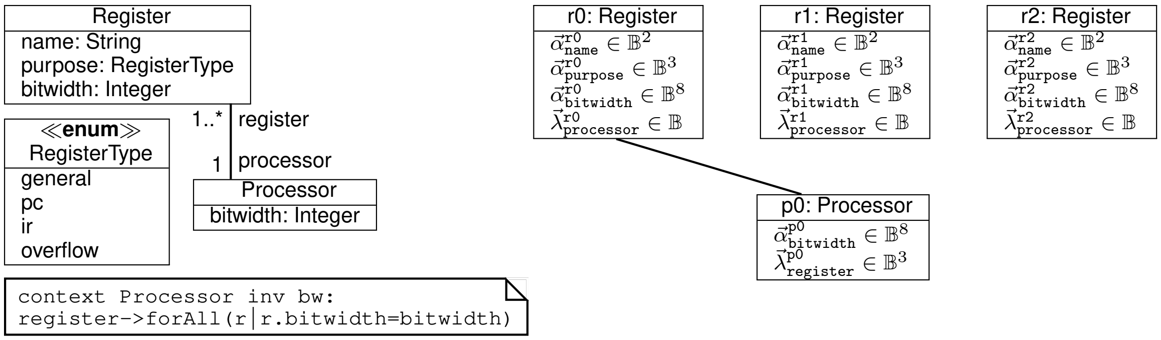
For each object to be created and for each attribute, SAT variables for the respective bits of $\vec{\alpha}$ are introduced to encode an object diagram. The value of n depends on the type of a. For Boolean attributes n is equal to 2. If a is an enumerated type over four possible values (e.g. as the *RegisterType* shown in Fig. 2(a)), then obviously $n = 4$. In case of integers, n is set to $2^l - 1$ which leads to a finite domain of l bit integers as already discussed in Section III. Another special case is the encoding of strings, which is done by abstraction. Here, n is set to the number of possible string values that can exist in the final system state[1].

For example, consider the UML class diagram given in Fig. 2(a). An object diagram representing a system state including three registers and one processor has to be encoded. Thus, variables as illustrated in Fig. 2(b) are introduced. In this case, integers are encoded by 8 bits.

Links between objects are encoded in a similar manner: Let there be an association between the classes $c_1, c_2 \in CLASS$ where e_1, e_2 define the respective roles. Then, for each object $o_1 \in \mathrm{oid}(c_1)$ a bit-vector $\vec{\lambda}_{e_2}^{o_1} \in \mathbb{B}^{k_{o_1}}$ is introduced which represents links to objects $o_2 \in \mathrm{oid}(c_2)$. The bit-width k_{o_1} depends on the number of objects of class c_2, i.e. $k_{o_1} = |\mathrm{oid}(c_2)|$. Assigning the i^{th} bit of $\vec{\lambda}_{e_2}^{o_1}$ to 1 states that o_1 is linked to the i^{th} object $o'_2 \in \mathrm{oid}(c_2)$, while assigning this bit to 0 states that both o_1 and o'_2 are unconnected. Analogously, for each object $o_2 \in \mathrm{oid}(c_2)$ a bit-vector $\vec{\lambda}_{e_1}^{o_2} \in \mathbb{B}^{k_{o_2}}$ with $k_{o_2} = |\mathrm{oid}(c_1)|$ is introduced. Additional constraints ensure the consistency of the semantics, e.g. they restrict the number of possible links to given multiplicities in the UML class diagram.

For example, consider the $\vec{\lambda}$-variables as shown in Fig. 2(b). Let $\vec{\lambda}_{\texttt{register}}^{\texttt{p0}}$ be assigned to 100. Then, $\vec{\lambda}_{\texttt{processor}}^{\texttt{r0}}$ must be assigned to 1 while $\vec{\lambda}_{\texttt{processor}}^{\texttt{r1}}$ and $\vec{\lambda}_{\texttt{processor}}^{\texttt{r2}}$ must be assigned to 0. These assignments state, that `p0` is linked to the register `r0` only.

[1] Note that the respective SAT variables $\vec{\alpha}$ allow the representation of 2^k values in total. Thus, it may be possible that values greater than the cardinality n of the domain of the attribute can be represented. This is prevented by explicitly excluding invalid assignments in the solution space.



(a) UML class diagram with one OCL constraint (b) SAT encoding of a system state

Fig. 2. An example of how to encode a given class diagram with OCL constraints as a system state using SAT

Having the encodings introduced so far, UML object diagrams can be created. The resulting SAT instance has to be given to a SAT solver, that returns with a satisfying assignment to all SAT variables if it exists. From this, the respective values of the object attributes can be obtained. Due to the added SAT constraints, only valid assignments are returned.

B. Encoding OCL Constraints

OCL consists of many “standard” operations like logical or arithmetical expressions for which many SAT encodings already exist (see e.g. [15]). However, since the undefined value $\perp$ may occur, we slightly extended these encodings to support OCL. Besides that, OCL offers the possibility to handle collections (sets, bags, sequences, ordered sets) and corresponding operations like `forAll` and `includes`. For these OCL constraints, respective SAT encodings have been developed. However, due to page limitations, it is not feasible to introduce all of them in detail. Thus, in the following we briefly describe the encoding of the OCL constraint used in the example.

The OCL constraint in Fig. 2(a) states that the bit-width of all `Register` objects connected to an object of the class `Processor` must be equal to the bit-width of that processor. The translation of the `forAll` statement is basically a conjunction of the inner expression over all connected registers. Whether a register is connected or not can be extracted from the variable $\vec{\lambda}^{\mathtt{p0}}_{\mathtt{register}}$. For object `p0` the following SAT constraint is generated for the invariant `bw`:

$$\bigwedge_{i=0}^{|\mathrm{oid}(\mathtt{Register})|-1} \left[\vec{\lambda}^{\mathtt{p0}}_{\mathtt{register}}[i] \Rightarrow \left(\vec{\alpha}^{\mathtt{r}i}_{\mathtt{bitwidth}} = \vec{\alpha}^{\mathtt{p0}}_{\mathtt{bitwidth}}\right)\right]$$

If $\vec{\lambda}^{\mathtt{p0}}_{\mathtt{register}}[i] = 1$, which means that there is a link, the inner expression must hold to satisfy the whole SAT constraint. Otherwise, the inner expression has no effect. The inner expression is thereby copied for each register.

C. Encoding the Verification Tasks

Using the encodings for system states as well as for UML constraints, verification tasks can be encoded. Together, they can be passed to the SAT solver which solves the respective problem. In the following, we exemplarily describe how to encode consistency. Nevertheless, further verification tasks, e.g. checking for independence, can be constrained similarly.

Each model M consists of a set $I = \{i_1, \dots, i_n\}$ of invariants. An invariant i is evaluated to be true or false in the context of a specific system state σ, denoted by $\sigma(i)$. Let $\sigma(M)$ be the set of all possible valid and invalid system states. Then the consistency problem can be formulated as [7]:

$$\exists \sigma \in \sigma(M) : \bigwedge_{i \in I} \sigma(i)$$

Encoding consistency in SAT, in fact, does not require any further SAT variables or SAT constraints, respectively. Using the encoding from above is sufficient. If the resulting SAT instance is satisfiable, then a valid system state (showing the consistency of the model) can be obtained from the satisfying assignment. More precisely, attribute values can be obtained from the assignments to $\vec{\alpha}$-variables and the links can be obtained from the assignments to $\vec{\lambda}$-variables, respectively. If the SAT instance is unsatisfiable, it has been proven that no consistent system state for the given bounds (i.e. maximal number of objects, restricted domains) exists.

TABLE I
SELECTION OF MODELS

Model	#Classes	#Attributes	#Associations
demo	3	7	3
simple-cpu	4	6	3
arbiter	5	7	6
person	1	4	1
train	2	0	2
percom	3	3	3
ex	3	7	3
carrental	8	14	10

V. EXPERIMENTAL EVALUATION

Using the proposed encodings, a verification approach getting a UML class diagram, OCL constraints, as well as the verification task as input has been implemented in C++. As underlying SAT solver, MiniSAT [5] has been used. In this section, we experimentally evaluate SAT-based UML/OCL model verification and compare it to previous approaches, namely USE [7] that is based on enumeration and Alloy [1] requiring transformation of UML/OCL using UML2Alloy.

As benchmarks, we use UML models given together with the USE environment (see [7]). Furthermore, a UML model representing an arbiter has been taken from [16]. Another model (namely *simple-cpu*) represents a simple CPU. The specifics of the UML models, i.e. the number of classes, attributes, and associations, are given in Table I. For all UML classes, appropriate OCL constraints have been written. Consistent (independent) as well as non-consistent (non-independent) models are thereby created, respectively.

In the following, we present the results obtained by performing consistency checks and independence checks. All experiments have been carried out on an Intel Core 2 Duo 2.2 GHz machine with 3 GB of main memory. The time-out was set to 500 CPU seconds.

TABLE II
RESULTS OF CONSISTENCY CHECKS

Model	#OCL	#Obj.	USE	Alloy	SAT
Consistent models					
demo	4	8	19.52	3.00	0.05
simple-cpu	7	7	>500.00	5.00	0.01
arbiter	2	8	14.24	2.00	0.01
person	4	3	17.78	7.00	0.02
train	7	6	>500.00	*N.A.*	0.02
percom	6	8	2.62	*N.A.*	0.00
ex	7	8	20.39	5.00	0.05
carrental	8	8	>500.00	*N.A.*	0.01
Inconsistent models					
demo	4	9	>500.00	2.00	0.02
simple-cpu	7	5	>500.00	3.00	0.01
arbiter	3	8	>500.00	4.00	0.01
person	5	3	>500.00	4.00	0.01
train	7	6	>500.00	*N.A.*	0.06
percom	7	8	10.72	*N.A.*	0.01
ex	8	8	>500.00	6.00	0.04
carrental	9	8	>500.00	*N.A.*	0.04

A. Consistency

In a first evaluation, the performance of the respective approaches on consistency checks are evaluated in detail. For this purpose, the respective models described above have been applied to our SAT-based approach, USE, and Alloy. The results are shown in Table II. The first column thereby denotes the name of the model, followed by columns giving the number of OCL constraints (#OCL) and objects (#Obj.), respectively. Finally, the last three columns give the run-times (in CPU seconds) for all three approaches.

As can be clearly seen, the SAT-based approach can handle all models in less than one CPU second. In contrast, USE either needs a significant amount of run-time or cannot check the model within the given time-out. In particular for the inconsistent cases, this can be explained by the enumerative behavior of USE. To prove the inconsistency of a model (considering the discussed restrictions), all possible system states have to be checked. Since USE does that step by step (without any pruning techniques), it runs into complexity problems.

In comparison to Alloy, it can be seen that the SAT-based approach is still faster; albeit not that significant. But even more important, using Alloy the models *train*, *percom*, and *carrental* cannot be applied (denoted by *N.A.*). This is, because UML2Alloy does not support transitive closure (used in *train* and *percom*) and n-ary associations (used in *carrental*), respectively (see also [1]).

B. Independence

In a second evaluation, independence checks are considered. Using Alloy, independence cannot be automatically checked. Thus, we omit Alloy in this evaluation. In contrast, USE has mechanisms for inverting invariants and so independence can be checked automatically using respective scripts. Using SAT, the independence verification task can be encoded by modifying the consistency encoding.

The resulting run-times (in CPU seconds) are given in Table III. Also here it can be seen, that the enumerative behavior of USE leads to large run-times (in fact, most of the instances cannot be solved within the time-out) while the SAT-based approach can solve these problems very fast.

TABLE III
RESULTS OF INDEPENDENCE EVALUATION

Model	#OCL	#Obj.	USE	SAT
Models with independent invariants				
demo	2	9	>500.00	0.04
simple-cpu	7	7	>500.00	0.05
arbiter	2	8	25.26	0.05
person	4	3	>500.00	0.05
train	3	6	>500.00	0.13
percom	3	12	>500.00	0.13
ex	2	8	161.65	0.03
carrental	4	8	>500.00	0.02
Models with non-independent invariants				
demo	4	9	>500.00	0.15
simple-cpu	7	7	>500.00	0.09
arbiter	3	8	>500.00	0.03
person	5	3	>500.00	0.07
train	7	6	>500.00	0.24
percom	6	12	>500.00	0.56
ex	7	8	>500.00	0.15
carrental	8	8	>500.00	0.01

VI. Conclusions

In this work, an approach for solving UML/OCL verification problems based on Boolean satisfiability has been presented. We described the encoding of the respective problem components and experimentally evaluated the application of our approach by means of consistency and independence checks. In comparison to previous work, verification tasks can be solved faster while still a significant variety of UML/OCL constructs is supported.

The proposed encodings can be applied to further verification problems without changing the formulation for the UML components, i.e. system states and OCL constraints.

References

[1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *Model Driven Engineering Languages and Systems*, pages 436–450. 2007.

[2] D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on UML class diagrams. *Artif. Intell.*, 168(1):70–118, 2005.

[3] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[4] J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 73–80, Washington, DC, USA, 2008. IEEE Computer Society.

[5] N. Eén and N. Sörensson. An extensible SAT solver. In *SAT 2003*, pages 502–518, 2003.

[6] M. Ganai and A. Gupta. *SAT-Based Scalable Formal Verification Solutions*. Springer, 2007.

[7] M. Gogolla, M. Kuhlmann, and L. Hamann. Consistency, independence and consequences in UML and OCL models. In *TAP 2009*, pages 90–104, 2009.

[8] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[9] M. Kyas, H. Fecher, F. S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons, and H. Kugler. Formalizing UML models and OCL constraints in PVS. *Electronic Notes in Theoretical Computer Science*, 115:39 – 47, 2005.

[10] T. Mancini. Finite satisfiability of UML class diagrams by constraint programming. In *Description Logics*, 2004.

[11] G. Martin and W. Müller. *UML for SOC Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[12] A. Queralt and E. Teniente. Reasoning on UML class diagrams with OCL constraints. In *ER*, pages 497–512, 2006.

[13] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley Professional, Boston, Massachusetts, second edition, 2004.

[14] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML models. In *UML*, pages 326–340, 2003.

[15] G. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pages 115–125, 1968.

[16] Y. Vanderperren, W. Mueller, and W. Dehaene. UML for electronic systems design: a comprehensive overview. *Design Automation for Embedded Systems*, 12(4):261–292, 2008.