IET Cyber-Physical Systems: Theory & Applications

Review Article

IET Journals
The Institution of Engineering and Technology

Verifying the structure and behavior in UML/OCL models using satisfiability solvers

ISSN 2398-3396
Received on 20th October 2016
Revised on 31st October 2016
Accepted on 4th November 2016
doi: 10.1049/iet-cps.2016.0022
www.ietdl.org

Nils Przigoda[1,2] ✉, Mathias Soeken[3], Robert Wille[2,4], Rolf Drechsler[1,2]

[1]*Group for Computer Architecture, University of Bremen, 28359 Bremen, Germany*
[2]*Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany*
[3]*Integrated Systems Laboratory, EPFL, Lausanne, Switzerland*
[4]*Institute for Integrated Circuits, Johannes Kepler University Linz, 4040 Linz, Austria*
✉ *E-mail: przigoda@informatik.uni-bremen.de*

Abstract: Due to the ever increasing complexity of embedded and cyber-physical systems, corresponding design solutions relying on modelling languages such as Unified Modelling Language (UML)/Object Constraint Language (OCL) find increasing attention. Due to the recent success of formal verification techniques, UML/OCL models also allow to verify and/or check certain properties of a given model in early stages of the design phase. To this end, different approaches for verification and validation have been proposed. In this work, the authors motivate, define, and describe different verification tasks for structural, as well as behavioural UML/OCL models that can be solved using solvers for Boolean satisfiability. They describe how these verification tasks can be translated into a symbolic formulation which is passed to off-the-shelf solvers afterwards. The obtained results enable designers to draw conclusions about the correctness of the considered model.

1 Introduction

The design of today's computing devices (including embedded and cyber-physical systems) is one of the most complex problems *Electronic Design Automation* (EDA) is currently facing. To handle the ever increasing complexity, designers strive for higher levels of abstraction in the corresponding design flows. While the design evolved from the *Register Transfer Level* to the *Electronic System Level* in the past, new trends include the exploitation of modelling languages such as the *Unified Modelling Language* (UML [1]) and the *Object Constraint Language* (OCL [2]). By this, the promising concepts of model-driven engineering [3, 4] are transferred from software engineering to the design of embedded and cyber-physical systems.

At the same time, ensuring the correctness of a system is going to become a crucial bottleneck in today's design flows. Since modelling languages such as UML/OCL allow for formal descriptions, they additionally enable designers to verify whether the model of a system indeed is correct or not. For example, in its most simple form, it can automatically be checked whether the structural description of given model indeed is free from contradictions (usually known as *consistency checking*). Moreover, if models contain operations specified in terms of contracts (i.e., pre- and post-conditions), even behavioural aspects of models can be verified.

A promising approach to conduct corresponding verifications is to translate the models and the corresponding verification tasks into an instance of a language that can be processed by a formal verification tool. This has been done for several target languages, which are chosen w.r.t. applicability and expressiveness of the modelling language and in particular to scalability, which is of highest priority. It has been observed that a translation into a low level target language yields better results, however, the translation is more complicated compared with targeting a more abstract language. As an example, automatic analysis using the modelling language Alloy [5] translated to SAT (Boolean satisfiability) is often efficient.

Translating UML/OCL [1, 2] models to Alloy first and then to SAT in order to utilise formal verification is easier than directly translating it to SAT. However, it is not surprising, that the overhead generated by the additional transformation yields worse results, i.e. worse performance, compared with the direct translation [6, 7]. Furthermore, a target language at a low abstraction level can provide more expressive freedom due to a few simple atomic expressions compared to more complex but less flexible constructs in a language at a higher abstraction level. However, translations into a low level language are difficult and require expert knowledge.

In this paper, we motivate, define, and describe different verification tasks for structural aspects of UML/OCL models such as consistency checking as well as behavioural aspects (discussed in Section 3). Based on this, we propose a general methodology which solves both kinds of verification task (cf. Section 4). To this end, we propose a symbolic formulation which represents all possible system states (cf. Section 5) as well as all sequences of operation calls (cf. Section 6). In addition, precise formal definitions for a selection of verification tasks are given and explained in detail (cf. Section 7). Some of these verification tasks are applied and evaluated by means of examples from the literature (cf. Section 8). Finally, this work is summarised with a conclusion in Section 9.

2 Background

To keep this paper self-contained, the basics on models and system states (as they are described in modelling languages such as UML or SysML) as well as on the OCL are briefly reviewed.

2.1 Models and system states

In the following, we make use of the convention, which denotes sets by upper case letters and single elements by lower case letters (with respect to some exceptions). Notations that refer to the model are denoted using letters from the Latin alphabet, while Greek letters refer to concepts in system states. All sets are finite sets.

A *model* $m = (\mathcal{C}, \mathcal{R})$ is a tuple of classes $\mathcal{C}$ and associations $\mathcal{R}$ (also known as *relations*). A *class* $c \in \mathcal{C}$ with $c = (A, O, \mathcal{I})$ is a

three-tuple composed of attributes A, operations O, and invariants $\mathcal{I}$ [For now, it is not important to understand that the attributes and operations actually rely on a precise underlying type system. Hence, we omit this fact in the definition. We will informally refer to the type system when we need it later in this paper.]. An *operation* $o \in O$ is a five-tuple $o = (P, r, \triangleleft, \triangleright, \mathcal{F})$ composed of a set of parameters P, a return value r, preconditions $\triangleleft$, postconditions $\triangleright$, and frame conditions $\mathcal{F}$. The invariants $\mathcal{I}$ from all classes as well as pre- and post-conditions ($\triangleleft$ and $\triangleright$) and the frame conditions of all operations are sets of OCL constraint expressions and are introduced in the following subsection.

Besides classes, a model $m = (\mathcal{C}, \mathcal{R})$ also consists of associations $r \in \mathcal{R}$. For the sake of simplicity, we restrict ourselves to binary associations. This restriction 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 [8]. Furthermore, modelling languages such as EMF [9] do not support n-ary associations at all. An *association* $r = (c_1, c_2, (l_1, u_1), (l_2, u_2))$ consists of classes $c_1, c_2 \in \mathcal{C}$ that should be related. The other elements in the tuple describe the multiplicities at the association ends in terms of a lower bound $l_i \in \mathbb{N}$ and an upper bound $u_i \in (\mathbb{N}\backslash\{0\}) \cup \{\infty\}$. That is, any object instance of class c_1 needs to be connected to at least l_2 as well as to at most u_2 object instances of class c_2 and vice versa. Of course, the relation must be also satisfied for all instances of all subclasses of c_1 and the opposite instance of the relation can also be a subclass of c_2. Note that restricting the multiplicities to intervals yields a simpler formalisation without decreasing the expressiveness as other multiplicities can be expressed in terms of additional OCL constraints.

Example 1: The notation is illustrated by means of the formal specification of a simple traffic light preemption, which also serves as a running example throughout the remainder of this paper. The corresponding model is depicted in Fig. 1. The main class of the example is the **`Controller`**, which is connected to exactly one traffic light signal for the cars and exactly one traffic light signal for the pedestrians. Two buttons are connected to the controller that can be pushed in order to send a request to the controller. This request is supposed to indicate whether pedestrians want to pass the street.

The model consists of two attributes, i.e. a **`request`** flag for the **`Controller`** class storing whether a pedestrian has requested to cross and a **`light`** attributes for the **`Signal`** class storing the current state [Note that in this work we assume that attributes are always defined.]. The invariant **`safety`** ensures a general safety property for traffic lights, i.e. both the signal for the pedestrians and the signal for the cars must not be 'green' at the same time. Finally, the invariant **`oneRole`** ensures that an instance of the class **`Signal`** can either serve as a pedestrian light or as a car light, but not as both.

Given a model $m = (\mathcal{C}, \mathcal{R})$, a *system state* $\sigma = (\Upsilon, \Lambda)$ is a tuple composed of object instances Υ derived from the classes $\mathcal{C}$ and a set of links Λ derived from the associations $\mathcal{R}$. An *object instance* or simply *instance* $\upsilon \in \Upsilon$ is a precise assignment of values to the attributes of the respective class c respecting the domain of each attribute. A *link* $\lambda \in \Lambda$ is a precise instance of an association, i.e. a connection of two instances $\upsilon_1, \upsilon_2 \in \Upsilon$ derived from $c_1, c_2 \in \mathcal{C}$ and with c_1, c_2 being connected by an association $r \in \mathcal{R}$.

Example 2: An example system state for the traffic light model from Fig. 1 is illustrated in Fig. 2. It consists of two controllers, four buttons, and four signals.

In the following, we are assuming a restricted state space in the considered models. A *problem bound* is defined for a given system state, i.e. the number of instances for each class and the values for infinitely large attribute domains (such as integers) are restricted. This restriction leads to a finite search space which makes the verification problem decidable. In turn, this hardly restricts the applicability of the approaches as, eventually, the implemented system will be composed of a finite set of instances and values anyway.

Further notations are introduced for convenience as follows: All instances of a class c (and also its subclasses) in a given system state are referred to by **`oid`**(c). The set of all valid system states for a given model m with respect to some predetermined problem bounds is denoted by $\mathcal{V}(m)$, i.e. $\mathcal{V}(m)$ contains all those variable assignments such that the evaluation of the invariants yield true. Note that $\mathcal{V}(m)$ is finite.

2.2 Object constraint language

The OCL is a declarative language which allows for the formulation of *constraint expressions*. Constraint expressions are used together with the model in order to add further restrictions that cannot be expressed by the given model notation itself. The OCL mainly consists of

- navigation expressions to access model elements,
- logic expressions (i.e., conjunction, disjunction, negation etc.),
- arithmetic expressions (i.e., addition, subtraction, multiplication, division etc.), and
- collection expressions (i.e., intersection, union, element containment etc.).

A comprehensive overview on all OCL expressions as well as keywords is given in [2]. A precise semantic definition can be also obtained from [2]. For a model m, a system state σ, and an arbitrary OCL expression e, we define $[\![e]\!]_m^\sigma$ as the *evaluation* of e in system state σ derived from the model m. When it is clear from

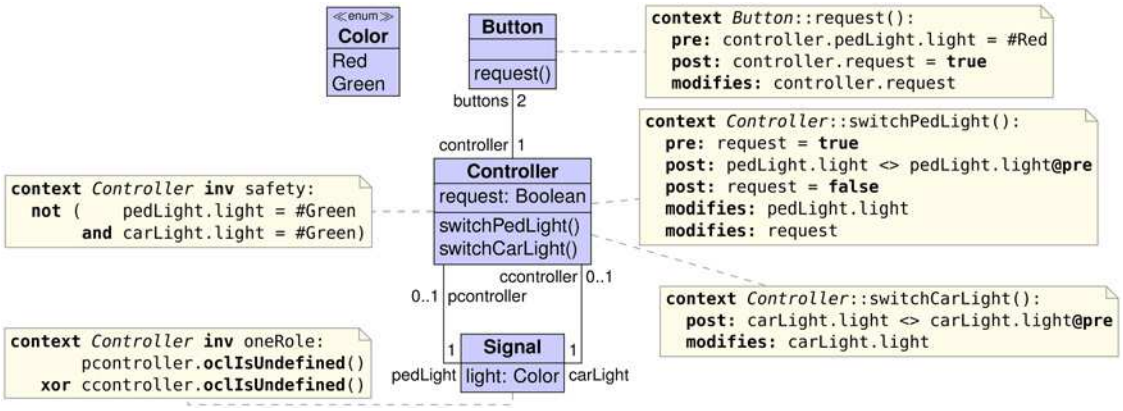


Fig. 1 *Traffic light example*

IET Cyber-Phys. Syst., Theory Appl., 2016, Vol. 1, Iss. 1, pp. 49–59

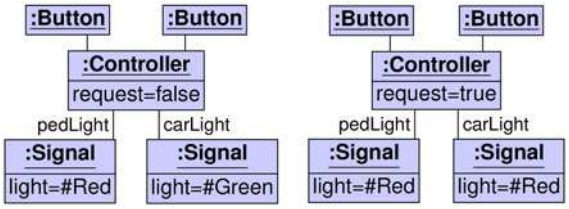


Fig. 2 *System state for the traffic light model*

the context, we drop the system state σ and/or the model m and write $[\![e]\!]$ for the sake of convenience.

Example 3: Consider again the model m from Fig. 1 and one of its system states σ as shown in Fig. 2. Let i be the invariant **`safety`**. This invariant is satisfied for the given system state σ, i.e. $[\![i]\!]_m^\sigma$ evaluates to **`true`**. Since the invariant and **`oneRole`** is also satisfied, σ is a valid instance of the model m, i.e. $\sigma \in \mathcal{V}(m)$.

Besides that and as already mentioned above, OCL expressions can also be applied in order to specify the pre-conditions $\triangleleft_o$ and post-conditions $\triangleright_o$ of an operation $o \in O$. The pre-conditions define in which systems states σ an operation o can be invoked (only in those with $[\![\triangleleft_o]\!]_m^\sigma = \textbf{true}$), while the post-conditions restrict the resulting successor state σ' (which must satisfy $[\![\triangleright_o]\!]_m^{\sigma,\sigma'} = \textbf{true}$). Those specifications are also known as *contracts*. Note that, in the case of a post-condition, the *invoking* state as well as the succeeding state may be needed in order to evaluate the navigation expression **`@pre`**. Again, for the sake of convenience, we simply drop the system states in the notation when the context is clear.

A frame condition $f \in \mathcal{F}$, also expressed in terms of OCL, evaluates to a set of model elements. These model elements are allowed to change their value during an operation call, while all other model elements are not allowed to change their value. In this work, they are introduced with the keyword **`modifies`** or **`modifiesOnly`**. More precisely, object instances and links can only be added or dropped through a transition that is given by an operation, if the model element is contained by any evaluated frame conditions. For more detailed and complete formal definition, we refer to [10, 11]. Approaches to semi-automatically determine frame conditions for a given model have been introduced in [12, 13]. Note that the model elements of frame conditions are not forced to change their values, while a change of any *non-selected* model element will make the transition immediately invalid.

Example 4: The running example from Fig. 1 consists of three operations whose contracts are defined in terms of pre-, post-, and frame conditions as follows:

- The operation **`request`** of class **`Button`** can only be called when the pedestrian light shows 'red', i.e. only then pedestrians are allowed to request for a 'green'-phase. After the operation has been executed, the request – attribute of the controller must be enabled.
- The single frame condition means that a transition is only valid, if only the attribute **`request`** of the connected **`controller`** has changed its value.
- The operation **`switchPedLight`** of class **`Controller`** can only be called if the **`request`** is set to true, i.e. the light for the pedestrian only switches if someone requested that before. After its execution, the signal of the pedestrian light changes and the request is set to false.
- The frame conditions allow only a change of the connected **`pedlight.light`** and the **`request`**.
- Finally, the operation **`switchCarLight`** switches the signal of the car light. This operation has no precondition.
- The sole frame condition allows only a change of the connected **`carlight.light`**.

3 Model verification

Having a formal model as defined in Section 2, it is often not obvious if the model is free of contradictions or other flaws or if it indeed describes the system as intended by the designer. Hence, verification of formal models received significant interest in the recent past. This section briefly reviews corresponding problems frequently considered in the literature and, afterwards, discusses related work. We distinguish between two categories: The verification of structural issues and the verification of behavioural issues. Later in this work, more precise verification tasks within these categories are considered.

3.1 Verification of structural aspects

Formal models provide a selection of constraints which all have to be satisfied by all valid system states. However, the complexity of the description may lead to over-constrained (or inconsistent) models, i.e. models from which no valid system state can be derived. As this would not allow a valid instantiation of the system to be realised, such models obviously are considered erroneous. Hence, checking for an inconsistent model (alternatively known as *consistency checking*) is considered as one of the most important structural verification issues. More precisely, a consistency check proves whether the model is free of contradictions by determining a valid system state of the given model. If it is shown that no valid system state exists, the model has been proven to be inconsistent.

Example 5: For the model given in Fig. 1, each system state is valid in which to each instance of class **`Controller`** there exists two associated instances of class **`Button`** and two associated instances of class **`Signal`**, one serving as pedestrian light and one as car light. Furthermore, the **`light`** attributes of associated pairs of **`Signal`** instances cannot be both assigned **`#Green`**. If associations are not set properly or if two associated signals are assigned **`#Green`**, the system state is invalid. As shown in Fig. 2, it is possible to derive a valid system for the model, and, thus, the model is consistent.

As mentioned above problem bounds are considered in order to make the verification tasks decidable. Some UML primitive types do not have bounds, e.g. *UnlimitedNatural*, *Integer*. Hence, we assign bounds based on hardware and/or software restrictions that arise from the target application. For example, an integer is usually bounded by 32 or 64 bits – in any case, it is restricted, e.g. by the available memory. The same holds for the number of objects to be instantiated from a UML model which, theoretically, may be infinite but, for practical applications, it can always be assumed bounded (there is no infinitely large system to be build). Hence, consistency checking, although undecidable in the general case, indeed can be made to an automatically decidable problem by adding reasonable bounds (motivated by practical applications).

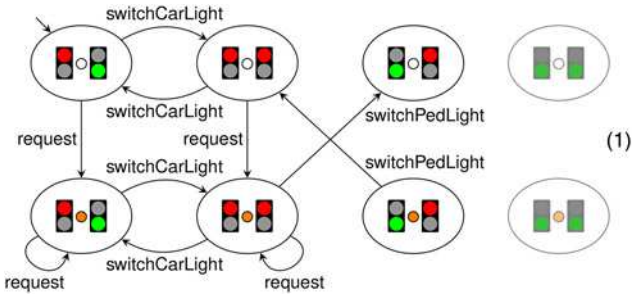
3.2 Verification of behavioural issues

Besides pure structural issues to be considered, models also include behavioural specifications in terms of operations with pre- and post-conditions. Hence, the correctness of these descriptions may also be of interest to the designer. Although all necessary information is provided by the operation contracts, the full model's behaviour can hardly be grasped from the class diagram representation. Instead, we illustrate behavioural verification in formal specifications by unfolding the complete possible behaviour of a model using a so-called *state-space graph*.

Given a model m, a state-space graph is a finite state machine $M = (\mathcal{V}(m), \sigma_0, \delta)$ with the state space being all valid states of the

model $\mathcal{V}(m)$, an initial state $\sigma_0 \in V(m)$, and a partial transition relation $\delta{:}\mathcal{V}(m) \times \text{ops}(m) \rightarrow \mathcal{V}(m)$ which maps a state and an operation to a *next state*. The set ops(m) returns the union of all operations from all classes in m. Given a current state σ, a next state σ', and an operation $o = (P, r, \triangleleft, \triangleright, \mathcal{F})$, then $\delta(\sigma, o) = \sigma_j$ holds if $[\![\triangleleft]\!]_m^{\sigma} \wedge [\![\triangleright]\!]_m^{\sigma,\sigma'} = \mathbf{true}$, i.e. if the preconditions hold in the current state σ and the post-conditions hold for the next state σ'. Besides this, the frame conditions have to be considered for valid operation as well, more precisely, an operation call is only valid if each changed model elements is contained in at least one evaluated frame condition f.

Example 6: Consider the model discussed in Section 2 and shown in Fig. 1. The complete state-space graph for two **Signal**s, one **Controller**, and two **Button**s is given as



where we assume  as an initial state in which the cars are allowed to drive (traffic light on the right) and the pedestrians are required to wait (traffic light on the left). The request signal, illustrated as a circle between the traffic lights, is filled orange if and only if it is set to true. Note that the invalid states in which both signals are green are not part of the state-space graph and are being shown greyed out only for a more comprehensive illustration. All states are connected by transitions labelled by the respective operation call.

Based on the state-space graph, different (universal) behavioural verification tasks can now easily be described, for example:

- *Is a certain state reachable?* The set of all *reachable states* $R = R_i$ with at least i operation calls is given as

$$R_i = \begin{cases} \{\sigma_0\} & \text{if } i = 0, \\ R_{i-1} \cup \text{succ}(R_{i-1}) & \text{otherwise} \end{cases} \quad (2)$$

  where

$$\text{succ}(R) = \bigcup_{\sigma \in R} \{\sigma' \in V(m) | \exists o \in \text{ops}(m){:}\delta(\sigma, o) = \sigma'\} \quad (3)$$

  and σ_0 is an initial state. In (1), the system state  is not reachable, because it has no incoming edge.
- *Are deadlock states possible?* A *deadlock* $\sigma \in R$ is a system state such that

$$\forall o \in \text{ops}(m){:}\delta(\sigma, o) \in V(m), \quad (4)$$

  i.e. there exists no valid transition from σ to any other valid system state. Given this information a serious design flaw in the model can be immediately determined from inspecting the state-space graph. In (1), a deadlock state is given by  , since the corresponding node has no outgoing edge.
- *Are livelocks possible?* A simple *livelock* scenario can be described as well. Let a *simple livelock state* be a system state σ in which only one operation can be called and that operation leads to the same state and that state can never be left again. Based on the set of reachable states R, a livelock $\sigma \in R$ is a system state with

$$\text{succ}(\{\sigma\}) = \{\sigma\}$$

Several other verification tasks can be described in a similar fashion – both based on the state-space graph and the underlying modelling languages such as OCL.

Considering these discussions, the verification of behavioural aspects is similar to the verification of structural aspects. In both cases, all possible system states need to be considered and, in case of the verification of behavioural aspects, a sequence of system states as well as the transitions between them need to be additionally considered. This leads to decision problems (either 'Does there exist a valid system state?' or 'Does there exist a corresponding sequence of systems states and operations?', respectively) which are decidable. However, for practically relevant models that are larger than the one given in Fig. 1, the search for those system states and sequences may be a rather complex and cumbersome task. Hence, efficient approaches are required for this purpose.

3.3 Related work

In the past, many approaches for structural verification tasks have been published [7, 14–17]. Almost all of them have dedicated advantages and disadvantages. As an example, the *UML-based Specification Environment* (USE) [16] provides well-established methods that can be applied, e.g. to (semi-)automatically generate system states for the respective UML/OCL models [18]. To determine such system states the designer has to write an ASSL script, which is executed enumeratively, i.e. in a worst-case scenario all system states, one at a time, have to be checked. Even for small models, this can make the runtime escalate.

In newer versions of USE, relation logic based on Alloy and the constraint solver KodKod can be used. This also replaces the ASSL idea. Furthermore, the authors of USE have also published an approach how behavioural aspects of models can be verified [19, 20]. For this purpose, the source model (called application model in their work) is transformed into a filmstrip model. By doing so, all dynamic models aspects are transformed into static ones. Some parts for a verification task have to be performed manually like rewriting the OCL constraints and formulating the exact verification task. For detailed comparison of the filmstrip model approach and the unrolling approach which is the underlying concept of our symbolic formulation the reader is referred to [21].

Approaches based on theorem provers like prototype verification system (PVS) [22], HOL-OCL/Isabelle [23], and KeY [24] can be used to check very large models, but often require a strong formal background of the designer as most steps have to be performed manually.

Consequently, researchers started to investigate the application of fully automatic proof engines including methods based on constraint programming (CSP) [15, 25, 26], description logic [27, 28], the modelling language Alloy based on relational logic [5, 14], or Boolean satisfiability (SAT) [7, 17]. The following sections will explain how a single system state or a sequence of them can be represented as an SAT problem. Furthermore, we will explain how, on top of this, generic verification tasks for models can be formulated. Due to the represented symbolic formulations, a number of verification tasks for models can be formulated in a fully automatic fashion into SAT problems which in turn can be solved fully automatically by using the deductive power of SAT solvers.

4 General idea

The general idea of the proposed methodology is to formulate the different UML/OCL model instantiations in a symbolic manner which represents all possible object instantiations, attributes and

link assignments, sequences of operations and so on. Afterwards, precise verification tasks will be addressed by pre-assigning certain parameters of the formulation (i.e., setting attributes of objects to certain values) and passing the resulting instance to a proper solving engine which efficiently solves the problem.

In this work, we propose to utilise satisfiability solvers (such as SAT solvers [29] or satisfiability modulo theories (SMT) solvers [30]) for this purpose. These solvers are highly-optimised algorithms that determine a satisfying solution for a given Boolean function $f:\mathbb{B}^n \rightarrow \mathbb{B}$, i.e. they derive an assignment $\boldsymbol{x}$ to all variables of f such that $f(\boldsymbol{x}) = 1$ or prove that no such assignment exists. Due to powerful implication and learning schemes, current state-of-the-art solvers are capable of solving instances composed of hundreds of thousands of variables and constraints [Note that, in the rest of this work, we will focus on formulations for SMT solvers rather than SAT solvers. This is motivated by the bit vector logic support of SMT solvers which makes the formulation easier to comprehend. A corresponding SAT formulation of those bit vector descriptions can easily be derived by bit blasting.].

In this section, the general idea of how to symbolically formulate corresponding UML/OCL model instantiations and how to solve verification tasks using them is sketched first. Afterwards, details on the resulting formulation are provided in the following sections.

4.1 Addressing structural verification tasks

To exploit the deductive power of solving engines such as SMT solvers, a verification flow as depicted in Fig. 3 is applied. For this purpose, a model must be given and problem bounds have to be defined. Those problem bounds can often be derived from the specification but there are also preliminary techniques that automatically extract appropriate problem bounds from a given model [31, 32]. On top of the model and the according problem bounds, a verification task can be formulated and a corresponding symbolic formulation in terms of a bit vector formula is derived. Note that in order to automatically obtain the symbolic formulation of a specific verification task the model and problem bounds are normally used again. Then, the resulting formula is given to an off-the-shelf solver which tries to determine a satisfying assignment of the formula. If such an assignment can be determined, a witness is extracted from it. In case the formula is proven to be unsatisfiable, it can be concluded that no instantiation for the respectively considered verification task can be generated – at least for the considered problem bounds.

For the symbolic representation in this work bit vector variables as well as bit vector constraints are used and passed as an input to an SMT solver. However, determining an efficient and correct bit vector constraint for each constraint given by the UML model and a possible extension with OCL is a non-trivial task. As a first step the necessary bit vector variables have to be declared. They are needed to represent attributes of instances and links between instances of a system state. If the number of valid instances is variable which depends on the problem bounds more bit vector variables are needed. This is illustrated by means of Fig. 5 for an encoded system state derived from the model in Fig. 1 for two controllers, four buttons, and four signals. This template serves as structure for the final symbolic representation which is further enriched by constraints that are derived from both, associations and their multiplicities as well as OCL invariants. Detailed

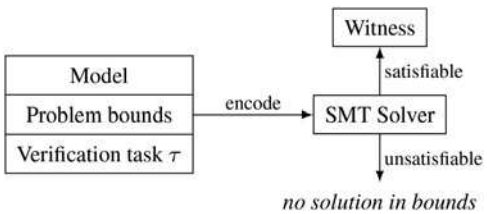


Fig. 3 *Verification of structural aspects using SMT solvers*

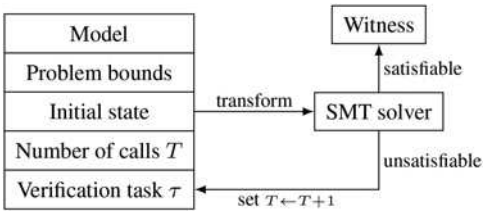


Fig. 4 *Verification of behavioural aspects using SMT solvers*

information on how to encode a system state will be presented later in Section 5.

By determining a satisfying assignment from an SMT solver, one also obtains a valid system state when reinterpreting the bit vector values as precise attribute values or links between instances. Such a satisfying assignment is called a *witness*, e.g. a witness for the validity or consistency of the model.

4.2 Addressing behavioural verification tasks

To additionally address behavioural verification tasks, we propose an approach that does not explicitly create a state-space graph (as discussed in Section 3.2), but again formulates the respective model instantiations, sequences, and verification task in a symbolic fashion. This is conducted by adjusting the verification flow discussed for structural verification tasks as shown in Fig. 4. The length of the sequence, i.e. the number of system states or number of operation calls denoted by the *number of calls T*, additionally has to be considered when creating the symbolic representation. Again, bounds may be assumed here in order to make the problem decidable. Also an initial state is provided. This is important since, otherwise, the set of reachable states cannot be determined.

Based on these inputs, the main idea is to symbolically formulate all possible *sequences* of system states in terms of a bit vector formula. Fig. 6 sketches the resulting structure of the formulation. System states, i.e. respective instantiations of the model, are encoded as already discussed above for structural verification

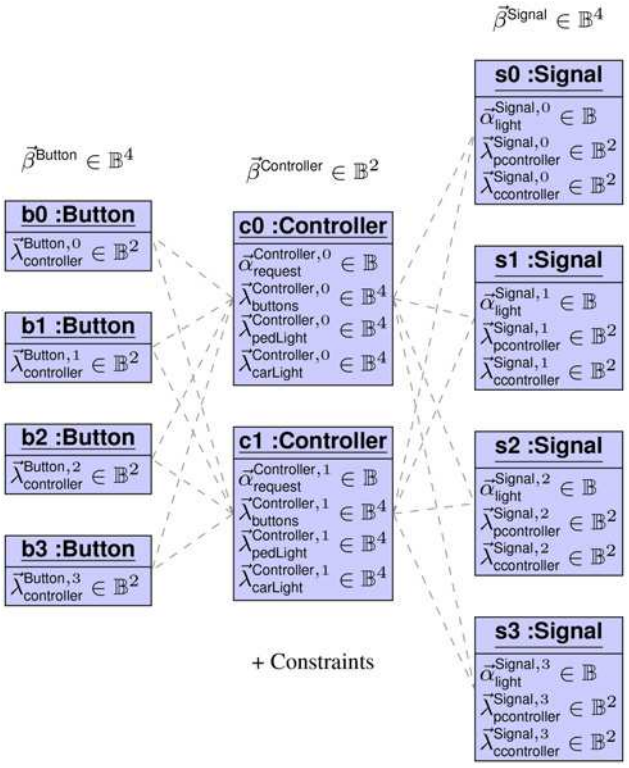


Fig. 5 *Bit vector representation of the system state*

tasks. In addition to that, operation calls are encoded which link successive system states to each other.

For this purpose, ω_i variables are applied. They represent all possible operations which may be called between two system states. Furthermore, additional constraints are added which ensure that the contracts of all operations are always satisfied. More precisely, if a solving engine sets one of the ω_i variables to a value representing the operation o, these constraints enforce that the calling system state σ_i as well as the succeeding system state σ_{i+1} indeed satisfy the pre-conditions $\triangleleft_o$ and post-condition $\triangleright_o$ of that operation o. Of course, also the frame conditions have to hold. Details on these constraints will be provided later in Section 6.

5 Symbolic formulation of a system state

To create a symbolic formulation representing all possible system states, instantiations of a model, assignments to attributes and links have to be represented as sketched in Fig. 5. Furthermore, constraints invoked on model elements through OCL invariants also have to be properly represented. Details on the respective formulations are provided in this section.

5.1 Symbolic formulation of attributes

To symbolically represent a system state $\sigma = (\Upsilon, \Lambda)$ in terms of a bit vector formula, for each object $\upsilon \in \Upsilon$ and for each of its attributes $a \in A$, a bit vector $\boldsymbol{\alpha}_a^\upsilon \in \mathbb{B}^k$ is created. The natural number k depends on the number of possible assignments that the attribute may assume. One can represent 2^k different values with $\boldsymbol{\alpha}_a^\upsilon$. Hence, for Booleans k is set to 1 and for 32-bit integer attributes to 32.

To translate an attribute representing an enum data type, k depends on the number of the possible values for the enum data type. Let m be the number of possible values, then $k = \max\{1, \log_2 m\}$. Furthermore a bijective mapping between the values and the set $\{0, 1, \ldots, m-1\}$ is used to represent any value like an integer.

Translation 1: Let $a \in A$ be an attribute of an enum data type, then the corresponding bit vector variables α_a^υ for all object instances must be restricted by the constraints

$$0 \leq \alpha_a^\upsilon \leq m - 1.$$

Real numbers are not considered throughout this paper. Strings usually allow a very large set of possible values which impedes the symbolic formulation and slows down the solving time significantly. Hence, we are proposing to use an abstraction technique for strings by bounding their number of values to the number of overall strings which may occur in the possible system states. This ensures that each string can be assigned a different value if required, but disables the use of more complex expressions such as substrings at the same time.

Example 7: As shown before, Fig. 5 shows the bit vector variables which are used in order to represent possible system states derived from the model shown in Fig. 1 for two controllers, four buttons, and four signals. More precisely, the respective α-variables either represent assignments to the **`request`**- or **`light`**-attributes of the **`Controller`**- or **`Signal`**-instances, respectively. Since all these attributes are of type Boolean, a bit vector of size 1 is sufficient to represent all possible assignments.

5.2 Symbolic formulations of links

Let $r = (c_1, c_2, (l_1, u_1), (l_2, u_2)) \in R$ be an association which belongs to the model. Then, for each object instance υ of the class c_1 a bit vector $\lambda_{c_2}^\upsilon \in \mathbb{B}^{|\text{oid}(c_2)|}$ and for each object instance υ' of the class c_2 a bit vector $\lambda_{c_1}^{\upsilon'} \in \mathbb{B}^{|\text{oid}(c_1)|}$ is created. These variables help to formulate whether two object instances are connected by a link or not.

It is assumed that if the ith bit of the $\lambda_{c_2}^\upsilon$-bit vector is 1, then the object instances υ is connected to the ith instance of c_2 [Note that a fixed order for the order for the object instances have to be defined which also have to be equal for all system states! However, as this can be easily done completely automatic, even in the case of generalisation.]. Otherwise υ is not connected. Constraints are additionally added to ensure that the respective links are consistent, i.e. when an instance υ_1 is connected with an instance υ_2, then υ_2 is also connected to υ_1. Also, cardinality constraints are added that ensure semantics of multiplicities by restricting the number of bits in the corresponding bit vectors. The cardinality constraints are translated as follows.

Translation 2: For a given model $m = (\mathcal{C}, \mathcal{R})$ and a derived system state $\sigma = (\Upsilon, \Lambda)$, the following formula for all associations $r = (c_1, c_2, (l_1, u_1), (l_2, u_2)) \in \mathcal{R}$ must be satisfied in order to have a valid system state

$$\bigwedge_{\upsilon \in \text{oid}(c_1)} l_1 \leq \sum_{i=0}^{|\text{oid}(c_2)|-1} \lambda_{c_2}^\upsilon[i] \leq u_1$$

$$\bigwedge_{\upsilon \in \text{oid}(c_2)} l_2 \leq \sum_{i=0}^{|\text{oid}(c_1)|-1} \lambda_{c_1}^\upsilon[i] \leq u_2$$

$$\bigwedge_{i=0}^{|\text{oid}(c_1)|-1} \bigwedge_{j=0}^{|\text{oid}(c_2)|-1} \lambda_{c_2}^{c_1,i}[j] = \lambda_{c_1}^{c_2,j}[i]$$

Example 8: Fig. 5 shows the respective λ-variables to be added for the running example. Due to the fact that two instances of the class **`Controller`** and four instances of the class **`Button`** are considered, the λ-variables of the **`Controller`** are of size 4, while the λ-variables of the **`Button`** are of size 2. Having this representation, two object instances are considered to be connected by a link, if their corresponding bits are set to 1. For example, the first instance of **`Controller`** and the third instance of **`Button`** are connected by a link in the system state, if the third bit of $\lambda_{\text{buttons}}^{\textbf{Controller},1}$ and the first bit of $\lambda_{\text{controller}}^{\textbf{Button},3}$ are both set to 1. Otherwise, there is only one other possible assignment for the

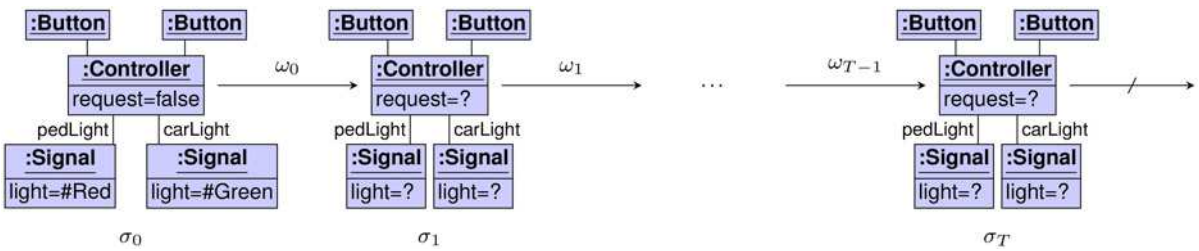


Fig. 6 *Bit vector representation of a sequence of the system states*

IET Cyber-Phys. Syst., Theory Appl., 2016, Vol. 1, Iss. 1, pp. 49–59

two bits: they are set to 0. In this case, the system state does not contain a link between the two objects. Note that both bits must be always equal in a satisfying assignment because of $\lambda_{c_2}^{c_1,i}[j] = \lambda_{c_1}^{c_2,j}[i]$ constraints which ensures the symmetry of a link the symbolic formulation.

5.3 Translating OCL expressions into bit vector formula

Thus far, the symbolic formulation allows for almost arbitrary assignments and, by this, arbitrary system instantiations. However, system models are usually restricted by OCL expressions and those have to be enforced in the symbolic formulation as well. In this section, we outline how OCL expressions are translated into bit vector constraints which can be used in the symbolic formulation. For this purpose, we will start from an abstract point of view and refine the translation rules step by step until everything is mapped into a symbolic formulation.

Note that handling all parts of the translation from OCL constraints into bit vector constraints would blow up the size of the present work extremely. Thus, we are concentrating on the main concepts and ideas. For more details the reader is referred to [6, 33].

To illustrate the main concept of the translation of an OCL expression into a bit vector constraint, the invariant **safety** will be used. Each invariant (as well as pre- and post-conditions) has a specific context [Note that in the former sections the context has been omitted in order to keep all explanations as easy as possible.] in which it should be applied to a system state. More precisely, context means here that, at least, the variable **self** must be defined for an evaluation. This is necessary, e.g. to get the correct variable of the connected **pedLight**.

Considering the system state as depicted in Fig. 5, the invariants have to be satisfied for both instances of the **Controller** and, thus, the invariants have to translate two times. In the first case, with the context information that **self** is the first instance of the **Controller** and in the second case, with the context information that **self** is the second instance of the **Controller**. In a general case, this can be formulated as follows:

Translation 3: For a given model $m = (\mathcal{C}, \mathcal{R})$ and a derived system state $\sigma = (\Upsilon, \Lambda)$, an invariant $i \in \mathcal{I}$ of a class $c \in \mathcal{C}$ is symbolically formulated as

$$\bigwedge_{v \in \mathrm{oid}(c)} [\![i]\!]_m^{\sigma,\{\mathbf{self} \mapsto v\}}. \tag{5}$$

Let us know differentiated between two different OCL navigation expressions: the ones pointing to an attribute like, e.g. **self.request**, and the ones using a reference to point to another object instance like, e.g. **self.pedLight**. The first case can be formulated as follows:

Translation 4: Given an OCL navigation expression like **self.request** (from which **request** is just a short form), the corresponding bit vector variable for further usage in bit vector expressions can be determined using the context information $\alpha_{\mathrm{request}}^{v}$ with **self** is mapped to v.

However, even with the context information it is not clear which object instance is meant in case of a reference. For example, it is not clear which **Signal** instance belongs to **pedLight** in the evaluation of $[\![\mathbf{inv}\ \text{safety}]\!]_m^{\sigma,\{\mathbf{self} \mapsto \text{Controller},0\}}$. However, this information can be received from the bit vector variable $\boldsymbol{\lambda}_{\mathrm{pedLight}}^{\mathrm{Controller},0}$. More precisely, if the ith bit is assigned **1**, then **pedLight** evaluates to the ith instance of **Signal**.

Translation 5: Given a navigation expression like **self.pedLight** (from which **pedLight** is just a short form), is translated into

$$\begin{aligned}&(\mathrm{ite}(=\boldsymbol{\lambda}_{\mathrm{pedLight}}^{\mathrm{Controller},0}[0]\ 1)^{\diamond \mathrm{Signal},0}\\ &\quad(\mathrm{ite}(=\boldsymbol{\lambda}_{\mathrm{pedLight}}^{\mathrm{Controller},0}[1]\ 1)^{\diamond \mathrm{Signal},1}\ \ldots\\ &\qquad(\mathrm{ite}(=\boldsymbol{\lambda}_{\mathrm{pedLight}}^{\mathrm{Controller},0}[|\mathrm{oid}(\mathit{Signal})-1|]\quad 1)\\ &\qquad\quad{}^{\mathrm{Signal},|\mathrm{oid}(\diamond \mathit{Signal})|-1}\perp)))\end{aligned} \tag{6}$$

where $\diamond^v$ is a placeholder variable for the corresponding object instance v and $\perp$ a placeholder for a null reference in case when the bit vector variable consists of zeros only.

Afterwards, when the object instance is clear, the variable for the attribute of a nested navigation expression, e.g. **self.pedLight.light**, can be determined:

Translation 6: Given a navigation expression like **self.pedLight.light**, Translation 5 is used to first translate **self.pedLight**. In a second step, the idea of Translation 4 is used on top of the preceding result. More precisely, all placeholder variables $\diamond^v$ are replaced with $\alpha_{\mathrm{light}}^{v}$.

The obtained constraints will return the variable belonging to a specific attribute or $\perp$ and can be used in bit vector operations.

As constants, e.g. for comparison or assigning specific values to an attribute, are common in OCL, they have to be translated as well.

Translation 7: The constant Boolean OCL expression **false** and **true** are translated to the constant bit vectors 0_2, respectively, 1_2. Constant integers are translated into the corresponding constant bit vectors representing the number and respecting a preconfigured bit length, e.g. 32 bit.

To translate the value of an enum data type, the bijective mapping between the values and the set $\{0, 1, \ldots, m-1\}$ is used to translate any value like a constant integer, but with the bit length $\max\{1, \log_2 m\}$.

For completing the translation of the OCL invariant **safety** into a bit vector constraint, only the operands, i.e. **not**, =, and **and**, have to translated into bit vector operands using the bit vector variables or constraints obtained from the translation rules mentioned before.

The presented translation rules have to be applied for all invariants in order to represent restrictions as expressed by OCL invariants.

Besides that, additional information guiding the designer through the translation process, e.g. in terms of ground setting properties as introduced in [34] may be added. Ground setting properties allow for a more efficient translation of a system state, if the values of some model elements are known, e.g. the links/relation between components are given and fixed.

6 Symbolic formulation of transitions

The considerations of the previous section lead to a new verification flow that is capable of handling verification tasks for behavioural aspects. For this purpose, the flow given in Fig. 3 is extended as depicted in Fig. 4. New inputs to the flow are the initial state and the number of operation calls to consider. Again, given all inputs an instance of the SMT problem is generated and solved by an off-the-shelf SMT solver. In case it yields a satisfying assignment, a witness can be deduced from it, otherwise it can be concluded that no such witness exists at least for the specified problem bounds and the number of operation calls. In this situation, the number of operation calls can be incremented and the algorithm is executed again.

To consider sequences of operations a notion of discrete time must be introduced to the formalism presented in Section 5. For this purpose, an object v is extended by a subscript $t \in \mathbb{N}_0$ where v_0 denotes an object from the *initial state*. Note that this formulation

of time is not directly connected to the timing properties of the targeted design but is used to employ a total ordering on system states. Furthermore, a notion of operation calls is required, which will act as the transitions between successive system states. An *operation call* $\omega_t = (\upsilon_t, o)$ invoked at time t is a tuple consisting of an operation o in state σ_t and an operation o that must be an operation of the object υ_t's class. That is, for each operation several operation calls may exist in a system state depending on the number of objects. The set of all operation calls Ω for a model $m = (C, R)$ is determined by

$$\Omega = \bigcup_{c \in C} \bigcup_{\substack{o \in \mathrm{ops}(c) \\ \upsilon \in \mathrm{oid}(c)}} \{(\upsilon, o)\}. \tag{7}$$

When enforcing a total order on Ω each operation call can be assigned a distinct index between 0 and $|\Omega| - 1$. A bijective function $r_\Omega : \Omega \to \{0, \ldots, |\Omega| - 1\}$ maps an operation to its index. The time subscript on the operation call does not have an influence as we assume that the number of objects does not change when changing the system state. This assumption is valid since the problem bounds need to be determined before the translated system state is created, dynamic allocation and destruction of objects is handled by select variables as described in [5] using the β^c bit vector variables.

Translation 8: When T is the number of operation calls, i.e. a sequence of $T + 1$ system states $\sigma_0, \ldots, \sigma_T$ is being considered, then the indices of the T operation calls $\omega_0, \ldots, \omega_{T-1}$ are encoded as bit vector variables

$$\boldsymbol{\omega}_t \in \mathbb{B}^{\lceil \log_2 |\Omega| \rceil} \tag{8}$$

where for all $t \in \mathbb{N}_T$

$$\boldsymbol{\omega}_t \leq |\Omega| - 1 \tag{9}$$

must be satisfied.

Example 9: Consider the traffic light model in Fig. 1. Assuming problem bounds of one controller, two signals, and two buttons, in each system state four operations may be called, i.e. **`switchPedLight`** and **`switchCarLight`** on the single instance of the controller and **`request`** on one of the two button instances. Therefore

$$\Omega = \{(c0, \textbf{switchPedLight}), (c0, \textbf{switchCarLight}), (b0, \textbf{request}), (b1, \textbf{request})\}$$

and

$$r_\Omega((c0, \textbf{switchPedLight})) = 0,$$
$$r_\Omega((c0, \textbf{switchCarLight})) = 1,$$
$$r_\Omega((b0, \textbf{request})) = 2,\ r_\Omega((b1, \textbf{request})) = 3$$

are possible.

Given a formalism for the index of operation calls on sequences, we still have to ensure that the corresponding OCL expressions, i.e. the pre- and post-conditions, of the operation are holding.

Translation 9: For the transition represented by the bit vector $\boldsymbol{\omega}_t$ the following constraints are added

$$\bigwedge_{\omega \in \Omega} \left(\boldsymbol{\omega}_t = r_\Omega(\omega)\right) \Rightarrow \left(\zeta_t(\,\triangleleft_\omega\,) \wedge \zeta_t(\,\triangleright_\omega\,)\right) \tag{10}$$

where

$$\zeta_t(\,\triangleleft_\omega\,) = \bigwedge_{f \in \triangleleft(\mathrm{op}(\omega))} [\![f]\!]_m^{\sigma_t, \{\mathbf{self} \mapsto \mathrm{obj}(\omega)_t\}}$$

evaluates the pre-condition constraints and

$$\zeta_t(\,\triangleright_\omega\,) = \bigwedge_{f \in \triangleright(\mathrm{op}(\omega))} [\![f]\!]_m^{\sigma_{t+1}, \{\mathbf{self} \mapsto \mathrm{obj}(\omega)_{t+1}\}}$$

post-condition constraints. Obviously these constraints have to be added for all t in $\mathbb{B}_{|\Omega|}$.

For a complete symbolic formulation of behavioural aspects, one also has to restrict every system state with by the constraint determined by the associations and the invariant as explained in the previous section.

Besides the pre- and post-conditions also the frame conditions have to satisfy. As a general translation into a symbolic formulation is unknown so far. However, in [10, 11] the author proposes to generate additional post-conditions based on the frame conditions such that all model elements which are not allowed to change their value are enforced to do so.

Example 10: For the **`request`** operation of the running example, the additional post-conditions given in Fig. 7 are generated from the modifies statement.

Besides the generation of additional post-conditions, it is also possible to directly integrate the information from the **`modifies`** statements [35]. Furthermore, in [36] the idea of single transition between two system states has been extended such that concurrent operation calls can also be considered.

```
context Button::request():  ...
  post: Button.allInstances()->forAll( b | b.controller = b.controller@pre )
  post: Controller.allInstances()->forAll( c | c.buttons = c.buttons@pre )
  post: Controller.allInstances()->forAll( c | c.pedLight = c.pedLight@pre )
  post: Controller.allInstances()->forAll( c | c.carLight = c.carLight@pre )
  post: Signal.allInstances()->forAll( s | s.pcontroller = s.pcontroller@pre )
  post: Signal.allInstances()->forAll( s | s.ccontroller = s.ccontroller@pre )
  post: Signal.allInstances()->forAll( s | s.ccontroller = s.ccontroller@pre )

  post: Signal.allInstances()->forAll( s | s.light = s.light@pre )
  post: Controller.allInstances()->forAll( c |
          (c <> self.controler) implies (c.request = c.request@pre )
        )
```

Fig. 7 *Additional post-conditions*

IET Cyber-Phys. Syst., Theory Appl., 2016, Vol. 1, Iss. 1, pp. 49–59

7 Verification tasks

In the previous section, a symbolic formulation was derived that represents a general UML/OCL model including its operation contracts based on a bit vector variables. Relying on this symbolic formulation, a variety of different verification tasks can be considered. We are referring to such verification tasks by τ in the formula. In this section, translations for special verification tasks are described, i.e. we explain what exactly have to be done for certain applications with the previously used τ.

7.1 Consistency

Let m be a model. Then the consistency of the model can be checked by encoding a single system state as a bit vector formula as explained in Section 5. The gained formula can be passed to an SMT solver. If the SMT solver determines a solution, we have witness for the consistency of the model and further verification tasks can be applied to the model. If the SMT solver has proven that no satisfying solution exists, the designer can check problem bounds and try again to determine a valid system state. Another issue can be that the model contains errors, e.g. in form of contradictions. In this case, the designer wants to find and fix the error.

For more details on debugging inconsistent UML models, the reader is referred to [37–40].

7.2 Executability

Given (10) a verification task τ can readily be specified. As an example, checking whether an operation o of a class c can be executed, is done by adding the following constraint

$$\bigvee_{v\in \mathrm{oid}(c)} \boldsymbol{\omega}_{T-1} = r_{\Omega}\big((v, o)\big). \tag{11}$$

There may be several operation calls ω which refer to the operation o. Each of the operation calls is associated to a number which can be obtained from the relation r_Ω. Executability is ensured if the last operation call $\boldsymbol{\omega}_{T-1}$ meets one of these ids. In (11), v is the operation's caller. Since the equation requires that o is called in the last operation, it is tailored for a scenario in which the flow depicted in Fig. 4 is initialised with one operation call that is incremented until a satisfying solution has been found. One may found a witness faster using an alternative flow in which a higher number of operation calls are given initially and allows the operation to be called in any of the system states. This can be done by extending (11) to

$$\bigvee_{t=0}^{T-1} \bigvee_{v\in \mathrm{oid}(c)} \boldsymbol{\omega}_t = \mathrm{id}_{\Omega}\big((v, o)\big). \tag{12}$$

In practice both formulas should be used in a hybrid manner: Starting with a reasonable number of operation calls it is first checked whether the operation can be called using (12). If this is not possible, the number of operation calls is incremented iteratively and (11) is used to check whether the additional time step enables the execution of the operation.

7.3 Reachability

If the verification task is to check whether a specific state can be reached or not. The dynamic encoding of (10) have been used again. In this case we just have to add constraints for the last system state which ensure that the specific system state is reached. For this purpose, a complete or just a partial initial system state can be encoded.

Obviously the flow of Fig. 4 can be used again.

The restrictions for the last system state can be done in two different ways. On the one hand, the designer can directly add constraints to the symbolic formula using the α and λ variables. On the other hand, it is also possible to formulate invariant, translate them with specific context to the last system state in order to restrict the valid values there. While the first way will normally be faster in a solving process, the second way does not require expert knowledge.

Such restrictions can also be used in order to describe any other system state instead of giving a complete state like done for the initial state.

7.4 Deadlock

To formalise a verification task that checks for a deadlock, the preconditions as well as the interaction between pre-conditions, post-conditions and invariants need to be taken into account. More precisely, two different kinds of checks have to be done.

At first, it should be ensured that at least one pre-condition of all operations cannot be satisfied in the last system state σ_T, i.e.

$$\bigwedge_{\omega\in\Omega} \neg \bigwedge_{f\in\triangleleft(\omega)} [\![f]\!]_m^{\sigma_T,\{\mathtt{self}\mapsto v_T\}}. \tag{13}$$

While the outer operator iterates over the set Ω which contains all possible operation calls, the inner operator requires that all preconditions evaluates to true, and by negating this evaluation, it is assured that at least one precondition of each operation for every operation is not satisfied.

For the second check consider the following deadlock scenario: For a system state exists an operation which satisfies all pre-conditions, but applying the post-conditions will cause a contradiction with at least one invariant. Thus, another clause has to be considered. At first, it has to be ensured that for each operation of each object a so-called *pseudo state* is added, noted by $\tilde{\sigma}$, respectively, specific bit vectors for all objects and its attributes. Then the following constraint must be added

$$\begin{aligned}\bigwedge_{\omega=(v,o)\in\Omega}\Bigg(& \bigwedge_{f\in\triangleleft(o)} [\![f]\!]_m^{\sigma_T,\{\mathtt{self}\mapsto v_T\}} \\ & \bigwedge_{f\in\triangleright(o)} [\![f]\!]_m^{\sigma_T,\tilde{\sigma}_{r_\Omega(\omega)},\{\mathtt{self}\mapsto v_{\tilde{\sigma}_{r_\Omega(\omega)}}\}} \\ & \wedge \neg\zeta_{\tilde{\sigma}_{r_\Omega(\omega)}}(I)\Bigg)\end{aligned} \tag{14}$$

As a deadlock only has to *satisfy* one of the two scenarios, the complete encoding is an expression of the inner constraints of (13) and (14).

Note: For the pseudo states the *modified* invariant evaluation constraint is applied by the last constraint.

8 Application

The approach described above enables designers to automatically verify the structure and the behaviour of UML/OCL models using satisfiability solvers. To this end, a symbolic representation of one or more than one system states has to be created (as illustrated in Fig. 5 or Fig. 6, respectively) and enriched by a formulation of the verification tasks τ (as described in Section 7). Afterwards, the resulting formula simply has to be passed to the SMT solver Z3 [41]. If the solver returns a satisfiable assignment, a system state or a sequence of system states can be derived from the assignments to the respective α-variables (allowing to obtain the values of all attributes), λ-variables (allowing to obtain the links in a system state), and ω-variables (allowing to obtain the chosen operation call). This either may work as counterexample (e.g., proving that a bad state indeed can be reached) or as witness (e.g., proving that the model is consistent, i.e., a valid system state indeed can be created). If the solver showed the non-existence of a satisfying assignment, it has been proven that no system state or sequence of system states exists which satisfies all constraints.

Table 1 Application and evaluation

Model	$\|C\|$	$\|A\|$	$\|R\|$	$\|O\|$	$\|I\|$	$\|\sigma\|$	$\|\mathcal{O}\|$	Task	Result	Run-time, s
BoardingSystem	2(0)	5	3	3	4	1	4	consistency	SAT	<1
CPU	6(0)	10	12	5	9	1	4	consistency	SAT	<1
Demo	3(0)	7	6	0	4	1	4	consistency	SAT	<1
CarRental	10(1)	21	26	6	6	1	4	consistency	SAT	<1
CarRental2	10(1)	21	26	6	7	1	4	consistency	UNSAT	<1
CivStat	1(0)	2	2	0	7	1	4	consistency	SAT	<1
PersonCompany	3(0)	3	6	5	5	1	4	consistency	SAT	<1
Phone	3(0)	3	6	4	0	1	4	consistency	SAT	<1
CPU	6(0)	10	12	5	9	2	4	executability	SAT	<1
Phone	3(0)	3	6	4	0	6	4	executability	SAT	<1
TrafficLights	2(0)	4	2	3	1	5	2	executability	SAT	<1
CPU	6(0)	10	12	5	9	10	4	deadlock	SAT	2.5
Phone	3(0)	3	6	4	0	5	4	deadlock	SAT	2.0
TrafficLights	2(0)	4	2	3	1	3	2	deadlock	SAT	<1

$|C|$: number of classes, in parenthesis the number of abstract classes, $|A|$: number of attributes, $|R|$: number of relations, $|O|$: number of operations, $|I|$: number of invariants, $|\sigma|$: number of system states, $|\mathcal{O}|$: maximal number of considered object instantiations per class in each system state

This may work as proof that, e.g. a particular system state cannot be reached.

All these checks are conducted automatically. Moreover, despite the (exponential) complexity of the search space to be considered for the respective tasks, evaluations showed that the applied solvers are capable of solving the resulting instances in an efficient fashion. This has exemplarily been demonstrated using models which are frequently applied in model finders in the past. More precisely, we considered models taken from the USE package [16] (i.e., *CPU*, *Demo*, *CarRental*, *CarRental2*, *CivStat*, *PersonCompany*) as well as applied in previous work [7, 42] (i.e., *CPU*, *Phone*, *TrafficLights*). While detailed descriptions of the models are available in the respective references, Table 1 briefly summarises their core characteristics, i.e. their number $|C|$ of classes, number $|A|$ of attributes, number $|R|$ of relations, number $|O|$ of operations, number $|I|$ of invariants, number $|\sigma|$ of considered system states, and maximal number $|\mathcal{O}|$ of considered object instantiations per class in each system state.

Using these models various verification tasks have been applied, i.e. consistency checks, executability checks, and checks for deadlocks. Both, satisfying instances (denoted by *SAT*) where either a counterexample or a witness has been obtained and unsatisfiable instances (denoted by *UNSAT*) where the non-existence of either one has been proved) were considered. Table 1 also lists the results of those checks (time is provided in CPU seconds; all executions have been conducted on an Intel i5-machine with 2.6 GHz and 16 GB of main memory). As can be clearly seen, the respective verification tasks can be conducted on all considered models in negligible run-time.

The proposed approach has been successfully applied to larger examples. Detailed results of the evaluations can be found in the literature (e.g., [43, 44]).

9 Conclusions

In this work, we have presented a methodology to translate a UML/OCL system state or a sequence of them based on UML class diagrams enriched with OCL description means into a symbolic formulation. Furthermore, it is explained how this can be used as a base to add different verifications task on top, e.g. consistency, executability, reachability, deadlock and so on. Using the proposed methodology, all parts of the symbolic formulation can be generated in nearly fully automatic fashion. More precisely, the designer has only to specify so-called problem bounds (i.e., the number of object instance of all classes, the length of the sequence) and the verification task. Afterwards, all needed variables and constraints for the symbolic formulation can be derived.

The methodology has also been implemented as a framework and the applicability has been tested for a bunch of models combined with a verification task. The resulting symbolic formulation was passed to an SMT solver and solved there. The run-times show that the automatic translation including the solving time is efficient.

10 Acknowledgments

This work supported by the Graduate School SyDe, funded by the German Excellence Initiative within the University of Bremen's institutional strategy, by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1, 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 as well as the Siemens AG.

11 References

1 Rumbaugh, J., Jacobson, I., Booch, G.: 'The unified modeling language reference manual', 1999
2 Object Management Group: 'Object Constraint Language, 2014. Version 2.4', February 2014
3 France, R.B., Rumpe, B.: 'Model-driven development of complex software: a research roadmap'. Workshop on the Future of Software Engineering, 2007, pp. 37–54
4 Selic, B.: 'The pragmatics of model-driven development', *IEEE Softw.*, 2003, **20**, (5), pp. 19–25
5 Jackson, D.: 'Software abstractions – logic, language, and analysis' (MIT Press, 2006)
6 Soeken, M., Drechsler, R.: 'Formal specification level – concepts, methods, and algorithms' (Springer, 2015)
7 Soeken, M., Wille, R., Kuhlmann, M., *et al.*: 'Verifying UML/OCL models using Boolean satisfiability'. Design, Automation and Test in Europe, 2010, pp. 1341–1344
8 Gogolla, M., Richters, M.: 'Expressing UML class diagrams properties with OCL'. Object Modeling with the OCL, 2002, pp. 85–114
9 Steinberg, D., Budinsky, F., Paternostro, M., *et al.*: 'EMF: eclipse modeling framework 2.0' (Addison-Wesley Professional, 2009, 2nd edn.)
10 Kosiuczenko, P.: 'Specification of invariability in OCL'. Int. Conf. on Model Driven Engineering Languages and Systems, 2006, pp. 676–691
11 Kosiuczenko, P.: 'Specification of invariability in OCL – specifying invariable system parts and views', *Softw. Syst. Model.*, 2013, **12**, (2), pp. 415–434
12 Niemann, P., Hilken, F., Gogolla, M., *et al.*: 'Assisted generation of frame conditions for formal models'. Design, Automation and Test in Europe, 2015, pp. 309–312
13 Niemann, P., Hilken, F., Gogolla, M., *et al.*: 'Extracting frame conditions from operation contracts'. Int. Conf. on Model Driven Engineering Languages and Systems, 2015, pp. 266–275
14 Anastasakis, K., Bordbar, B., Georg, G., *et al.*: 'UML2Alloy: a challenging model transformation'. Int. Conf. on Model Driven Engineering Languages and Systems, 2007, pp. 436–450
15 Cabot, J., Clarisó, R., Riera, D.: 'Verification of UML/OCL class diagrams using constraint programming'. ICST Workshops, 2008, pp. 73–80
16 Gogolla, M., Büttner, F., Richters, M.: 'USE: a UML-based specification environment for validating UML and OCL', *Sci. Comput. Program.*, 2007, **69**, (1–3), pp. 27–34
17 Torlak, E., Jackson, D.: 'Kodkod: a relational model finder'. Tools and Algorithms for Construction and Analysis of Systems, 2007, pp. 632–647
18 Gogolla, M., Kuhlmann, M., Hamann, L.: 'Consistency, independence and consequences in UML and OCL models'. Tests and Proof, 2009, pp. 90–104

19 Gogolla, M., Hamann, L., Hilken, F., *et al*.: 'From application models to filmstrip models: an approach to automatic validation of model dynamics'. Modellierung, 2014, pp. 273–288
20 Hilken, F., Hamann, L., Gogolla, M.: 'Transformation of UML and OCL models into filmstrip models'. Int. Conf. on Theory and Practice of Model Transformations, 2014, pp. 170–185
21 Hilken, F., Niemann, P., Gogolla, M., *et al*.: 'Filmstripping and unrolling: a comparison of verification approaches for UML and OCL behavioral models'. Tests and Proof, 2014, pp. 99–116
22 Kyas, M., Fecher, H., de Boer, F.S., *et al*.: 'Formalizing UML models and OCL constraints in PVS', *Electron. Notes Theor. Comput. Sci.*, 2005, **115**, pp. 39–47
23 Brucker, A.D., Wolff, B.: 'A proposal for a formal OCL semantics in Isabelle/HOL'. Theorem Proving in Higher Order Logics, 2002, pp. 99–114
24 Beckert, B., Hähnle, R., Schmitt, P.H.: 'Verification of object-oriented software: The KeY approach' (Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007)
25 Malgouyres, H., Motet, G.: 'A UML model consistency verification approach based on meta-modeling formalization'. ACM Symp. on Applied computing, 2006, pp. 1804–1809
26 Mancini, T.: 'Finite satisfiability of UML class diagrams by constraint programming'. Description Logics, 2004
27 Berardi, D., Calvanese, D., De Giacomo, G.: 'Reasoning on UML class diagrams', *Artif. Intell.*, 2005, **168**, (1–2), pp. 70–118
28 Van Der Straeten, R., Mens, T., Simmonds, J., *et al*.: 'Using description logic to maintain consistency between UML models'. UML, 2003, pp. 326–340
29 Eén, N., Sörensson, N.: 'An extensible SAT-solver'. SAT, 2003, pp. 502–518
30 Brummayer, R., Biere, A.: 'Boolector: an efficient SMT solver for bit-vectors and arrays'. Tools and Algorithms for Construction and Analysis of Systems, 2009, pp. 174–177
31 Clarisó, R., González, C.A., Cabot, J.: 'Towards domain refinement for UML/OCL bounded verification'. Int. Conf. on Software Engineering and Formal Methods, 2015, pp. 108–114
32 Soeken, M., Wille, R., Drechsler, R.: 'Towards automatic determination of problem bounds for object instantiation in static model verification'. Workshop on Model-Driven Engineering, Verification and Validation, 2011, pp. 2:1–2:4
33 Soeken, M., Wille, R., Drechsler, R.: 'Encoding OCL data types for SAT-based verification of UML/OCL models'. Tests and Proof, 2011, pp. 152–170
34 Przigoda, N., Wille, R., Drechsler, R.: 'Ground setting properties for an efficient translation of OCL in SMT-based model finding'. Int. Conf. on Model Driven Engineering Languages and Systems, 2016, pp. 261–271
35 Przigoda, N., Filho, J.G., Niemann, P., *et al*.: 'Frame conditions in symbolic representations of UML/OCL models'. Int. Conf. on Formal Methods and Models for System Design, 2016
36 Przigoda, N., Hilken, C., Wille, R., *et al*.: 'Checking concurrent behavior in UML/OCL models'. Int. Conf. on Model Driven Engineering Languages and Systems, 2015, pp. 176–185
37 Przigoda, N., Wille, R., Drechsler, R.: 'Contradiction analysis for inconsistent formal models'. Int. Symp. on Design and Diagnostics of Electronic Circuits & Systems, 2015, pp. 171–176
38 Przigoda, N., Wille, R., Drechsler, R.: 'Analyzing inconsistencies in UML/OCL models', *J. Circuits Syst. Comput.*, 2016
39 Torlak, E., Chang, F.S.-H., Jackson, D.: 'Finding minimal unsatisfiable cores of declarative specifications'. Formal Methods, 2008, pp. 326–341
40 Wille, R., Soeken, M., Drechsler, R.: 'Debugging of inconsistent UML/OCL models'. Design, Automation and Test in Europe, 2012, pp. 1078–1083
41 de Moura, L.M., Bjørner, N.: 'Z3: an efficient SMT solver'. Tools and Algorithms for Construction and Analysis of Systems, 2008, pp. 337–340
42 Soeken, M., Wille, R., Drechsler, R.: 'Verifying dynamic aspects of UML models'. Design, Automation and Test in Europe, 2011, pp. 1077–1082
43 Delmas, R., Doose, D., Pires, A.F., *et al*.: 'Supporting model based design'. Int. Conf. on Model and Data Engineering, 2011, pp. 237–248
44 Guerra, E., Soeken, M.: 'Specification-driven model transformation testing', *Softw. Syst. Model.*, 2015, **14**, (2), pp. 623–644