Checking Concurrent Behavior in UML/OCL Models

Nils Przigoda[1] Christoph Hilken[2] Robert Wille[1,3] Jan Peleska[2] Rolf Drechsler[1,3]

[1]Group for Computer Architecture, University of Bremen, 28359 Bremen, Germany
[2]Research Group Operating Systems & Distributed Systems, University of Bremen, 28359 Bremen, Germany
[3]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{przigoda,chilken,rwille,jp,drechsle}@informatik.uni-bremen.de

***Abstract*—The *Unified Modeling Language* (UML) is a de-facto standard for software development and, together with the *Object Constraint Language* (OCL), allows for a precise description of a system prior to its implementation. At the same time, these descriptions can be employed to check the consistency and, hence, the correctness of a given UML/OCL model. In the recent past, numerous (automated) approaches have been proposed for this purpose. The behavior of the systems has usually been considered by means of sequence diagrams, state machines, and activity diagrams. But with the increasing popularity of design by contract, also composite structures, classes, and operations are frequently used to describe behavior in UML/OCL. However, for these description means no solution for the validation and verification of *concurrent* behavior is available yet. In this work, we propose such a solution. To this end, we discuss the possible interpretations of "concurrency" which are admissible according to the common UML/OCL interpretation and, afterwards, propose a methodology which exploits solvers for SAT Modulo Theories (i. e., SMT solvers) in order to check the concurrent behavior of UML/OCL models. How to address the resulting problems is described and illustrated by means of a running example. Finally, the application of the proposed method is demonstrated.**

I. Introduction

The *Unified Modeling Language* (UML) supports designers in the description of complex systems in early design phases and is considered as a de-facto standard for software development [1]. By providing several diagram types, it allows the description of a system in different fashions. This includes the global view of the system as a whole as well as the detailed description of one particular component. For the purpose of behavioral descriptions, e. g., the behavior of an operation, the UML also offers different description means including sequence diagrams, state machines, and activity diagrams. Besides that, the *Object Constraint Language* (OCL) [2] can be used to extend a UML model with additional textual constraints that define further properties and relations between the respective parts of the model. Analogously, pre- and postconditions can be added to an operation describing (1) the prerequisites for calling an operation and (2) the desired system state after the execution of the operation. As a result, it is possible to provide a design by contract description (i. e., a formal description of *what* the system is supposed to do), while sequence diagrams, state machines, and activity diagrams focus on the realization (i. e., *how* an operation shall be implemented). In this work, we focus on the design by contract description scheme.

A crucial requirement in the design process of a complex system is its validation and verification, i. e., the question of how to check whether the given system is consistent and described as intended. With increasing design complexity, it is decisive how and when verification is firstly being employed. Due to shortening time-to-market demands, design flaws need to be detected efficiently and as early as possible. Being an abstract methodology, UML/OCL serves as a good starting point for this purpose.

Consequently, numerous (automated) approaches for the validation and verification of UML/OCL models have been proposed (a detailed review on related work is provided later in Section III-A). While these approaches considered both structural and behavioral verification tasks, the *concurrent* execution behavior of composite structure diagrams, class diagrams, and operations have not yet been analyzed in detail until today. This may be due to the fact that composite structures and class diagrams have long been considered as structural elements of the UML only (while behavioral aspects have long been modeled by sequence diagrams, state machines, and activity diagrams only).

However, design by contract becomes more and more popular and relies on a description by means of pre- and postconditions as well as invariants. Consequently composite structures, classes, and operations are increasingly used to describe complete systems *including* their behavior.[1] But, to our best knowledge, no comprehensive methods for the validation and verification of *concurrent* behavior described by means of such UML/OCL models have yet been proposed.

In this work, we propose a methodology which addresses this problem, i. e., a solution is presented which is capable of checking concurrent behavior of UML/OCL models consisting of composite structure diagrams, class diagrams, and operation specifications. To this end, we first discuss which of the alternative interpretations of "concurrency" – both admissible variants of the UML/OCL specification – shall be considered (i. e., whether an interleaving semantic or a true-parallelism interpretation shall be applied). Afterwards, a symbolic formulation is proposed which represents arbitrary concurrent behavior that is possible according to a given UML/OCL model. Solvers for SAT Modulo Theories (i. e., SMT solvers) are eventually applied to derive the respective validation/verification result. The resulting problems as well as the proposed solutions are thereby described and illustrated by means of a running example.

The remainder of this work is structured as follows. The next section introduces the notation used in this paper and briefly reviews the basics on Boolean satisfibility and SAT

[1]In the remainder of the paper, we just use the term "UML/OCL models" for this subset of UML.

978-1-4673-6908-4/15 © 2015 IEEE

MODELS 2015, Ottawa, ON, Canada
Foundations

Accepted for publication by IEEE. © 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Modulo Theories. Related work on concurrency in general as well as concurrency in UML/OCL in particular is discussed in Section III. This also motivates the considered problem and defines the interpretation of concurrent behavior as applied in this work. The proposed solution is then described in detail in Section IV and Section V: First a (simple) extension of existing approaches for validation and verification of sequential behavior is proposed; afterwards, explicit problems caused by concurrent behavior are addressed. Finally, the application of the proposed methodology is demonstrated in Section VI before the paper is concluded in Section VII.

II. Preliminaries

In order to keep the paper self-contained, this section provides a brief review on UML/OCL and introduces the notation used in this paper. Furthermore, the Boolean satisfiability problem and corresponding solvers are briefly reviewed.

A. Applied Notation for UML/OCL Models and States

In this work, we are using the following notation in order to refer to elements of a UML/OCL model:

Definition 1. *A* model $m = (\mathcal{C}, \mathcal{R})$ *is a tuple of* classes $\mathcal{C}$ *and* relations $\mathcal{R}$ *(also known as* associations*). A class $c \in C$ with $c = (A, O, I)$ is a 3-tuple composed of attributes A, operations O, and invariants I*[2]*. An* operation $o \in O$ *is a 5-tuple $o = (P, r, \lhd, \rhd, \mathcal{F})$ composed of a set of parameters P, a return value r, preconditions $\lhd$, postconditions $\rhd$, and frame conditions $\mathcal{F}$. The invariants I from a class as well as pre- $\lhd$ and postconditions $\rhd$ of an operation are sets of* OCL constraint expressions*. A relation $r = (c_1, c_2, (l_1, u_1), (l_2, u_2)) \in \mathcal{R}$ consists of two classes c_1 and c_2 in $\mathcal{C}$ as well as two tuples representing the multiplicities between the classes (i. e., its lower and upper bounds).*

Note that frame conditions are applied in order to precisely define which model elements may change their value during an operation. The definition of frame conditions might be required since, otherwise, unintended side effects may occur. In the past, various approaches for specifying frame conditions have been proposed, e. g., the implicit assumption that only model elements may change their value which are part of the postcondition, invariability clauses, or an explicit coverage of this issue in the postconditions (see, e. g., [3], [4], [5]).

Instances of a UML/OCL model represent a *system state* for which the following notation is applied:

Definition 2. *For a given 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* instance $\upsilon \in \Upsilon$ *is a precise assignment of values to the attributes of the respective class $c \in \mathcal{C}$ respecting the domain of the 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 $c_1, c_2 \in C$ and with c_1, c_2 being connected by an association.*

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 problems considered in this work decidable. However, the proposed approach can be applied on an infinite state space to solve some problems, for example to find a witness for a bug. Furthermore, under certain conditions an infinite state space can be rearranged and reduced to a finite one using additional techniques, e. g. equivalent classes.

Further notations for a model $m = (\mathcal{C}, \mathcal{R})$ are introduced for convenience as follows: All instances of a class $c \in \mathcal{C}$ in a given system state σ are referred to by $\Upsilon(c)$. The set of all operations of a class $c \in \mathcal{C}$ is denoted by $\mathrm{ops}(c)$. The set of all model elements of a system state σ (e. g., attributes and links), are denoted by $m(\sigma)$.

The *Object Constraint Language* (OCL) is a declarative language which allows for the definition of *constraint expressions*. Constraint expressions are applied 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 elements in the model,
- logic expressions (i. e., conjunction, disjunction, etc.),
- arithmetic expressions (i. e., addition, division, etc.), and
- collection expressions (i. e., intersection, union, etc.).

A comprehensive overview of all OCL expressions as well as its keywords is given in [2]. A precise semantic definition can also be obtained from [2]. For a model m, a system state σ, and an arbitrary OCL expression e, we define $[\![e]\!]^{\sigma}_{m}$ as the *evaluation* of e in system state σ derived from the model m. When it is clear from the context, we drop the system state σ and/or the model m and write $[\![e]\!]$ for the sake of convenience.

Definition 3. *Let $m = (\mathcal{C}, \mathcal{R})$ be model and σ, σ' two system states. Then an* operation call ω *is a pair $\omega = (\upsilon, o)$ where υ is an object instance in σ of a class $c \in \mathcal{C}$ which has an operation o. In order to invoke a valid operation call ω, the equation*

$$[\![\lhd_\omega]\!]^{\sigma}_{m} = \texttt{true}$$

must hold. Furthermore, each valid operation call must lead to a successor state σ' in which both, the postcondition and possibly applied frame conditions, are satisfied, i. e.,

$$[\![\rhd_\omega]\!]^{\sigma,\sigma'}_{m} \wedge [\![\mathcal{F}_\omega]\!]^{\sigma,\sigma'}_{m} = \texttt{true}.$$

Obviously, 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 $m = (\mathcal{C}, \mathcal{R})$ and system state σ is determined by

$$\Omega = \bigcup_{c \in \mathcal{C}} \bigcup_{\substack{o \in \mathrm{ops}(c) \\ \upsilon \in \Upsilon(c)}} \{(\upsilon, o)\}.$$

Note that, in the case of a postcondition, the *invoking* state σ as well as the succeeding state σ' 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.

When enforcing a total order on Ω, each operation call can be assigned a distinct index between 0 and $|\Omega| - 1$. A function $r_\Omega : \Omega \to \{0, \ldots, |\Omega| - 1\}$ maps an operation to its index. This assumption is valid since the problem bounds need to be determined before the considered system state

[2]In the remainder of this paper, it is not important to understand that the attributes rely on a precise underlying type system. Hence, we omit this fact.

is created. Dynamic allocation and destruction of objects is handled by select variables as described in [6]. Furthermore, a total order on the model elements can be defined by the following function $r_{m(\sigma)} : m(\sigma) \rightarrow \{0, \ldots, |m(\sigma)| - 1\}$.

B. Boolean Satisfiability

The *Boolean Satisfiability* (SAT) problem is defined as follows: Let $f : \mathbb{B}^n \rightarrow \mathbb{B}$ be a Boolean function. Then, the SAT problem is to determine an assignment for the variables of f so that f evaluates to 1 or to prove that no such assignment exists.

Example 1. *Let $f(x_1, x_2, x_3) = (x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + x_2)(\overline{x}_2 + x_3)$. Then, $x_1 = 1, x_2 = 1$, and $x_3 = 1$ is a satisfying assignment for f. The value of x_1 ensures that the first clause becomes satisfied, the value of x_2 ensures this for the second clause, and the value of x_3 ensures this for the remaining clause.*

The SAT problem is one of the central $\mathcal{NP}$-complete problems. In fact, it was the first known $\mathcal{NP}$-complete problem that was proven by Cook in 1971 [7]. But, in the past efficient solving algorithms (so called *SAT solvers*) have been proposed (see, e. g., [8]). Instead of simply traversing the complete space of assignments, intelligent decision heuristics, *conflict based learning*, and sophisticated engineering of the implication algorithm by *Boolean Constraint Propagation* (BCP) lead to an effective search procedure. Once it is proven that no solution exist, an instance is called *unsatisfiable* (UNSAT), otherwise *satisfiable* (SAT). Due to these efficient algorithms, problem instances composed of hundreds of thousands of variables, millions of clauses, and tens of millions of literals can be handled.

Besides Boolean Satisfiability, the *Satisfiability Modulo Theories* (SMT) problem is a very similar decision problem. SMT can be seen as special SAT problem which allows to work on bit vector logic rather than pure Boolean logic. By this, e. g. integer attributes of UML/OCL can be represented by a single (bit vector) variable only. In the following, SMT instances are provided in SMTlib syntax specified in [9]. Here, each constraint is provided following the Polish notation, i. e., each operation is encapsulated by parenthesis and the operator is provided before the list of (ordered) operands.

Example 2. *Consider the following SMTlib formula, where* `bv1` *and* `bv2` *are bit vectors of size 4:*[3]

```
1 (not (= bv1 bv2) )
2 (= ((_ extract 1 1) bv2) #b1)
```

Line 1 shows a constraint composed of two operations, namely negation (`not`) and equivalence (`=`), and states that the bit vector `bv1` *and* `bv2` *are not supposed to assume the same value, i. e., in a symbolical notation* $\texttt{bv1} \neq \texttt{bv2}$.

Line 2 illustrates a constraint which manages the access of single bits within a vector. The constraints enforces that bit number 1 of the bit vector `bv2` *has to be assigned* `1`.[4] *Solving this SMT instance may lead to the assignments* `bv1 = #b1101` *and* `bv2 = #b1010`.

For a detailed list of all operators and the logic descriptions the reader is refereed to [9].

[3]The prefix `#b` indicates that a binary string is following.

[4]The bits of a bit vector are ordered from right to the left and the numbering starts with 0.

III. Problem Formulation and Related Work

This section briefly reviews the previous work which has been conducted on the verification of behavior in UML/OCL models and, by this, motivates the problem considered in this paper: How to verify concurrent behavior in UML/OCL models? Afterwards, a review of the respective models for concurrent execution in general is provided. The underlying concepts partially provide the basis for the solution proposed in this work.

A. Related Work and Considered Problem

Having a model of a design does not necessarily imply that the actually desired implementation can be derived from it. In fact, the model may inherit constraints which contradict each other or may describe unwanted and erroneous behavior. As a consequence, researchers and engineers intensely investigated how to validate and verify models given, e. g., in UML/OCL [6], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21]. Automated validation and verification must be based on formal semantic models. Due to their similarity to Harel's Statecharts [22], early definitions of behavioral UML semantics focused on UML state machines [23]. In [24], [25], it has been shown how UML/SysML models whose behavior is encoded by concurrent state machines, operation calls, and timing conditions can be associated with a formal semantics. There, it has also been explained how verification by bounded model checking and automated test data generation can be performed, if the behavioral semantics are described by means of a transition relation in propositional form.

For the behavioral interpretation of class diagrams and their operations, existing approaches utilize the constraints provided by the pre- and postconditions, as well as the frame conditions given for each operation in the model, as explained, for example, in [26]. More precisely, in order to show whether a specific behavior indeed is represented by the model, all valid sequences of operation calls are (explicitly or symbolically) checked. If a sequence of operation calls can be determined which satisfies the respective constraints and eventually leads, e. g., to previously defined bad states or good states, the erroneous or correct behavior has been shown, respectively. In order to conduct these checks, various approaches relying, for example, on the so-called film-strip model [27], satisfiability solvers [28], or alternative solving techniques (e. g., ILP) have been proposed. A comparison on these approaches has been conducted, e. g., in [29].

Most of these approaches, however, rely on a sequential behavioral interpretation. This means that in each transition from one system state to another, only a single operation call is considered. This contrasts with modern systems which typically rely on the parallel execution of operation calls and, hence, concurrent behavior. To our best knowledge, no comprehensive method for the validation and verification of concurrent behavior described by means of UML/OCL models consisting of composite structures, class diagrams, and operations have been proposed yet.

A multitude of solutions for modelling concurrent designs and associated tool support have been introduced in other formalisms outside the UML domain. Popular tools are, e. g., FDR for the CSP process algebra [30] and the more recent mCRL2 process algebra with its tool set [31]. Despite their

expressive power and their noteworthy tool capabilities, however, these and similar formalisms have the disadvantage that they do not represent industrial de-facto standards, as it is the case of UML, SysML, and OCL.

Besides that, there exists a wide variety of tools for checking the concurrent behavior represented in terms of (1) Harel's statecharts (see, e. g., [22], [32], [33], [34]), (2) variants of Petrinets (see, e. g., [35]), as well as (3) graph transformations (see, e. g., [36]). Since these representations are quite similar to UML state machines (for (1)) and UML activity diagrams (for (2) and (3)), they may, in principle, offer a solution for checking the concurrent behavior of these UML description means. However, UML state machines and UML activity diagrams serves an entirely different purpose than the definition in terms of contracts (i. e., pre- and postconditions) considered in this work. In fact, they focus on implementation aspects while contracts provide a specification of the intended behavior.

Overall, to the best of our knowledge there is no solution available which can automatically verify the concurrent behavior defined by UML models enriched with OCL pre- and postconditions. In this work, we are proposing such a solution.

B. Considered Computational Model

Obviously, an approach aiming for checking the behavior of a system description has to rely on a proper computational model. With respect to concurrency, a significant amount of corresponding theoretical foundations to be used have been considered – [37] and [30] provide good overviews of some of them. Besides that, there are attempts to compare and classify these models (see, e. g., [38]). In case of the design by contract scheme considered here, "concurrency" basically translates to invoking and executing two or more operation calls at once. In general, such a system is nondeterministic. Therefore, two options for a computational model are left to be chosen from:

- *Interleaving Model*: In this model, concurrently called operation calls are executed in an "interleaved" fashion, i. e., they are executed sequentially, but their atomic steps are executed nondeterministically. In this case, it is not possible to split these operation calls into smaller units. In other words, one atomic step is the execution of one operation, another atomic step is the execution of another operation, and so on. Therefore, concurrently executing nondeterministic operation calls in interleaving semantics result in the same state space as if they would be executed sequentially. This model is typical for concurrent processes on a single-core CPU.
- *Non-interleaving Model (true-parallelism)*: In this model, concurrently called operation calls are executed in a "true parallel" fashion. The result is the merged result of the concurrently executed operation calls, or a deadlock, or an illegal racing condition if those operation calls cannot run in parallel because they change the same attributes in contradictory ways. In case of a UML/OCL model, two operation calls can only be executed in parallel if *all* their preconditions are satisfied before and *all* their postconditions are satisfied after the call. Moreover, the frame conditions of *all* operation calls must be considered in such a parallel execution. This model is typical for many real world scenarios.

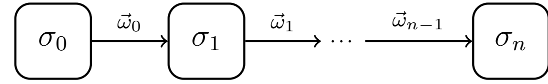

Figure 1: Transitions with sequential operation calls

In this case, considering a nondeterministic behavior, the interleaving model results in the same state space as in sequential execution. Therefore, true-parallelism fits better for today's complex systems which run truly parallel. Hence, we consider the non-interleaving model with its true-parallelism in the remainder of this work.

IV. Applied SMT-based Validation and Verification

In order to validate and verify concurrent behavior defined by means of UML/OCL models, we propose an approach which relies on previously proposed solutions for checking sequential behavior. In this section, the background on the respectively utilized approach is reviewed. Afterwards, corresponding extensions aiming to support concurrency are introduced. While this provides a first step towards the desired verification approach, further obstacles remain open. These are considered afterwards in Section V.

A. Validation and Verification of Sequential Behavior

In the recent past, several approaches for the validation and verification of sequential behavior have been proposed (see, e. g., [27], [28], [29]). In this work, we rely on the solution presented in [28] which makes use of solvers for *SAT Modulo Theories* (SMT solvers). They allow for an efficient traversal of large search spaces and, hence, are suitable to determine whether certain sequences of operations leading to bad/good states indeed can be derived from the UML/OCL model.

The general idea is sketched by means of Fig. 1: A symbolic representation of all possible sequences of system states (denoted by $\sigma_0, \sigma_1, \ldots, \sigma_n$ and bounded by $n \in \mathbb{N}$) is considered. Each transition from a system state σ_i to a system state σ_{i+1} (with $i = 0, 1, \ldots, n-1$) is triggered by a (single) operation call which is symbolically defined by $\vec{\omega}_i$. Initial states as well as the following states can be restricted (e. g., in order to define a bad/good state for which a sequence of operation calls is to be determined). Then, it remains open how to explicitly assign all operation calls $\vec{\omega}_i$ so that, with respect to the corresponding pre- and postcondition of the chosen operations, a valid sequence of transitions from σ_0 to σ_n is derived.

This formulation is eventually formulated into SMT solver syntax; here, a representation in terms of *bit vector logic*. More precisely, the following formulation is applied:

Formulation 1. *For a model $m = (\mathcal{C}, \mathcal{R})$ and a sequence of system states $\sigma_0, \sigma_1, \ldots, \sigma_n$, let Ω be the set of all operation calls which can be performed within one system state. The operation call leading to the transition from a system state σ_i to a successor state σ_{i+1} (with $i = 0, 1, \ldots, n-1$) is represented by the bit vector $\vec{\omega}_i$ of size $\lceil \log_2 |\Omega| \rceil$.*[5] *Then, each transition is symbolically represented by*

$$\bigwedge_{\omega \in \Omega} (\vec{\omega}_i = r_\Omega(\omega)) \Rightarrow (\llbracket \lhd_\omega \rrbracket \wedge \llbracket \rhd_\omega \rrbracket \wedge \llbracket \mathcal{F}_\omega \rrbracket) \qquad (1)$$

[5]We assume that $|\Omega| > 1$, i. e., there is more than one operation call.

where

- $r_\Omega(\omega)$ *is a unique identifier representing the operation call* $\omega \in \Omega$,
- $[\![\lhd_\omega]\!]$ *is a constraint enforcing the precondition for system state* σ_i,
- $[\![\rhd_\omega]\!]$ *is a constraint enforcing the postcondition for system state* σ_{i+1}*, maybe by using* σ_i *as well, and*
- $[\![\mathcal{F}_\omega]\!]$ *is a constraint enforcing the frame conditions for the entire transition (i. e., for both system states).*

Besides that, the possible assignments to the bit vector $\vec{\omega}_i$ *is restricted by*

$$\bigwedge \vec{\omega}_i < |\Omega| \quad (2)$$

in order to ensure that the value of $\vec{\omega}_i$ *is an element of the image set* $r_\Omega(\Omega) = \{0, \ldots, |\Omega| - 1\}$.

Using this formulation, a satisfying assignment to all $\vec{\omega}_i$-variables must exist if a sequence from the (possibly restricted) initial system state σ_0 to the (possible restricted) terminal system state σ_n exist. From this assignment, the respective operation calls for this sequence can eventually been obtained. If no such assignment exists, it has been proven that no corresponding sequence exists. SMT solvers are capable of determining such assignments or proving their non-existence in an efficient fashion.

Example 3. *Consider the model of a simple counter as given in Fig. 2. The model consists of one class* Counter *which has an integer attribute* `value` *representing the current value of the counter and an operation* `count`*. Furthermore, one invariant* `greaterZero` *belongs to the class which requires that the value must be always greater or equal 0. The behavior of the operation is described by a postcondition, which requires that the value of the counter of the calling object should be increased by one, and a postcondition, which requires that all remaining counters remain their value.*

Further, let σ *be a system state as given in Fig. 3. There are three object instantiation of the class* Counter*, namely* Counter@0*,* Counter@1*, and* Counter@2*. As it is possible to call the operation* `count` *on each object, it follows that* $\Omega = \{(\text{Counter@0}, \texttt{count}), (\text{Counter@1}, \texttt{count}), (\text{Counter@2}, \texttt{count})\}$*. Based on that, the resulting SMT syntax of the transition for operation call* $(\text{Counter@0}, \texttt{count})$ *can be formulated. The used variable-identifiers are structured as follows: The state number followed by the object name and the attributes; further parts are connected with a* `::`*. The invoking (succeeding) state in denoted with* `State0` *(*`State1`*). Altogether, the formula reads as follows:*

```
1 (=> (= omega #b00)
2     (and (= State1::Counter0::value
3             (bvadd State0::Counter0::value
4                    #x01))
5          (and (=> (not (= #b001 #b001))
6                   (= State1::Counter0::value
7                      State0::Counter0::value))
8               (=> (not (= #b001 #b010))
9                   (= State1::Counter1::value
10                     State0::Counter1::value))
11              (=> (not (= #b001 #b100))
12                  (= State1::Counter2::value
13                     State0::Counter2::value))
14         )
15     )
16 )
```

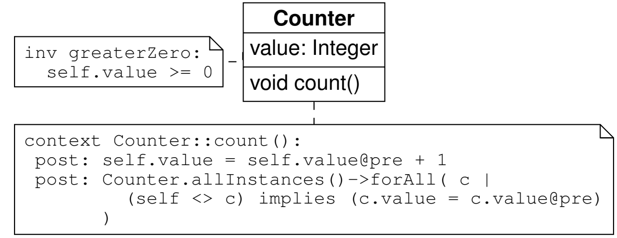

Figure 2: A model of a simple counter

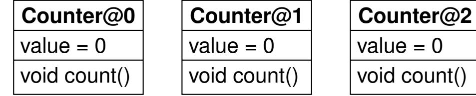

Figure 3: A system state for the simple counter model

The first line, realizes the left-hand side of the implication sketched in Eq. 1 (assuming that 00 *is the unique identifier* $r_\Omega(\omega)$ *representing the operation* $(\text{Counter@0}, \texttt{count})$*). Afterwards, the postconditions (which inherently also represent the frame condition) of the operation are enforced (see Line 2–4 and Line 5–13). Hereby, the system states are represented by the assignments to the respective attributes of the instantiated classes. For example, the value of the attribute* value *of the object instance* Counter@0 *is represented by* `State0::Counter0::value` *and* `State1::Counter0::value` *for the initial system state and the succeeding system state, respectively. Based on that, the first postcondition (defining the increase of the* value*) is enforced by the SMT constraint in Line 2–4. The other postconditions are realized in a similar fashion.*

B. Supporting Concurrent Behavior

The approach reviewed in the previous section obviously does not support the consideration of concurrent behavior. In fact, the SMT formulation allows for the execution of a single operation call per transition only. In order to extend this concept accordingly, a revised formulation has to be applied.

Fig. 4 sketches the general idea of such an extended formulation. Again, a symbolic representation of all possible sequences of system states (denoted by $\sigma_0, \sigma_1, \ldots, \sigma_n$ and bounded by $n \in \mathbb{N}$) is considered. But instead of having a variable $\vec{\omega}_i$ which symbolically represents a (single) operation call triggering the transition from σ_i to σ_{i+1} (as in Fig. 1), we now consider $|\Omega|$ possible (concurrent) operation calls – each of them symbolically represented by an array of Boolean variables denoted by $\vec{\omega}_i$ and with size $|\Omega|$. More precisely, the Boolean $\vec{\omega}_i[k]$ evaluates to true iff an operation call with the unique identifier k is executed in the ith transition, i. e., a one-hot encoding is employed. By this, it becomes possible to apply more than one operation call. Moreover, the total number of concurrent operation calls can be restricted by limiting the number of $\vec{\omega}_i[k]$-variables which are assigned `1`.

Overall, this leads to the extended symbolic formulation of a each transition as follows:

Formulation 2. *For a model* $m = (\mathcal{C}, \mathcal{R})$ *and a sequence of system states* $\sigma_0, \sigma_1, \ldots, \sigma_n$*, let* Ω *be the set of all operation calls which can be conducted within one system state. The (concurrent) operation calls whose parallel execution*

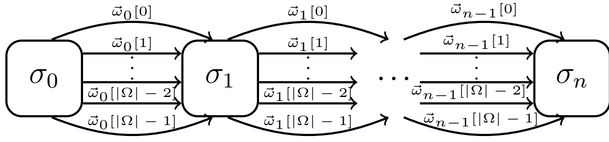

Figure 4: Transitions with concurrent operation calls

– conducted in a true-parallelism fashion – leads to the transition from a system state σ_i to a successor state σ_{i+1} are represented by the bit vector $\vec{\omega}_i$ of size $|\Omega|$. Then, each transition is symbolically represented by

$$\bigwedge_{\omega\in\Omega} (\vec{\omega}_i[r_\Omega(\omega)] = 1) \Rightarrow (\llbracket\lhd_\omega\rrbracket \wedge \llbracket\rhd_\omega\rrbracket \wedge \llbracket\mathcal{F}_\omega\rrbracket) \tag{3}$$

which is almost identical to the formulation from Eq. 1 except for the fact that the operations are now symbolically represented by means of a one-hot encoding. This in turn allows for the consideration of more than one operation call (since the antecedent of the implication may evaluate to true for more than one operation $\omega \in \Omega$). Moreover, by additionally enforcing

$$\bigwedge_{i=0}^{n-1} \left(d_1 \leq \sum_{k=0}^{|\Omega|-1} \vec{\omega}_i[k] \leq d_2 \right) \tag{4}$$

with $d_1, d_2 \in \mathbb{N}$ and $d_1 \leq d_2$, it is possible to restrict the number of concurrent calls to be between d_1 and d_2.

Again, if the resulting formulation leads to a satisfying assignment to all the $\vec{\omega}_i[k]$-variables, the existence of a sequence of corresponding operation calls has been proven. In contrast to the formulation from the previous section, this sequence now may include concurrent operation calls in one transition. More precisely, two operations $\omega, \omega' \in \Omega$ $(\omega \neq \omega')$ with the unique identifiers $r_\Omega(\omega) = k$ and $r_\Omega(\omega') = k'$ are executed in parallel in the ith transition iff both $\vec{\omega}_i[k]$ and $\vec{\omega}_i[k']$ are assigned `1` by the SMT solver.

While this provides a simple solution to support concurrent behavior in the validation and verification of UML/OCL models, it does not consider that operations may have contradictory effects and, hence, provoke conflicts in the succeeding system state. The following example illustrates the resulting problem:

Example 4. *Consider again, the simple counter model from Example 3. Following the extended formulation, the resulting SMT syntax of the transition from the system state with respect to the operation call* (Counter@0, *count*) *reads as follows:*

```
1  (=> (= ((_ extract 0 0) omega) #b1)
2      (and (= State1::Counter0::value
3               (bvadd State0::Counter0::value
4                      #x01))
5           (and (=> (not (= #b001 #b001))
6                    (= State1::Counter0::value
7                       State0::Counter0::value))
8                (=> (not (= #b001 #b010))
9                    (= State1::Counter1::value
10                      State0::Counter1::value))
11               (=> (not (= #b001 #b100))
12                   (= State1::Counter2::value
13                      State0::Counter2::value))
14          )
15     )
16 )
```

As we are interested in a concurrent execution of operation calls, additionally also the resulting SMT formulation for the operation call (Counter@1, *count*) *is considered in detail:*

```
1  (=> (= ((_ extract 1 1) omega) #b1)
2      (and (= State1::Counter1::value
3               (bvadd State0::Counter1::value
4                      #x01))
5           (and (=> (not (= #b010 #b001))
6                    (= State1::Counter0::value
7                       State0::Counter0::value))
8                (=> (not (= #b010 #b010))
9                    (= State1::Counter1::value
10                      State0::Counter1::value))
11               (=> (not (= #b010 #b100))
12                   (= State1::Counter2::value
13                      State0::Counter2::value))
14          )
15     )
16 )
```

Both sets of constraints enforce that, when the respective operations are called (represented by setting the 0th and the 1st bit of bit vector $\vec{\omega}$ to `#b1`), (1) the attribute `value` of the respective object is increased by one and (2) the values of all remaining attributes remain unchanged. This obviously leads to a conflict: An attribute cannot be increased by one and, at the same time, keep its value. While this is in accordance to the postconditions of the model as shown in Fig. 2, it requires an explicit handling of contradictory conditions.

Consequently, only extending existing approaches for validation and verification of sequential behavior does not lead to satisfactory solutions addressing the “concurrent case”. Instead, a further analysis on possible contradictions of operation contracts has to be conducted for each model. The result of such an analysis eventually has to be incorporated into an accordingly revised SMT formulation. How this can be accomplished is covered in the next section.

V. Handling Contradictory Conditions

Conditions defined in contracts (either by means of postconditions or frame conditions) are supposed to completely describe the effect of an operation. As illustrated in the example from the previous section, this often also affects model elements which are not really in the scope of a particular operation. For example, the operation (*Counter@0*, `count`) could entirely focus on the attribute `value` of object *Counter@0*. But, in order to avoid arbitrary changes to attributes from all the other objects, the postcondition additionally restricts `value` from *Counter@1* and *Counter@2*.[6]

As illustrated in Example 4, this constitutes a problem. In fact, when several operation calls are executed in parallel it indeed might be acceptable to reduce the scope of the conditions. In other words, postconditions and frame conditions of a single operation do not necessarily have to cover *all* model elements – in particular when another (concurrently executed) operation already covers those elements anyway. In order to address this, a modeling scheme is assumed in the following in which postconditions only restrict model elements that are relevant to the respective operation. For all remaining model elements, a “nothing else changes”-assumption is employed as long as no other (concurrently executed) operation modifies them.

[6]Note that, instead of a postcondition, also a frame condition could have been applied for this purpose.

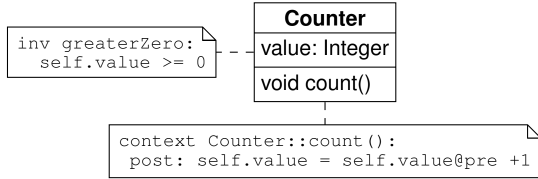

Figure 5: Modified model of a simple counter

Example 5. *Consider again the example from Fig. 2. Following the "nothing else changes"-assumption, this model can be defined as shown in Fig. 5. Here, the postcondition only restricts the attribute of the respectively calling object. All other attributes are assumed to keep their value (except another operation is restricting it).*

Following this assumption, it remains open to symbolically represent which model elements are affected by an arbitrary combination of concurrent operation calls. The values of all these model elements are already defined by the conditions of these operations. For all remaining model elements, constraints have to be employed which ensure that their values are not supposed to change during the transition. To this end, for each operation call $\omega \in \Omega$, a dedicated bit-mask b_ω of size $|m(\sigma)|$ is generated where

- each model element $\mu \in m(\sigma)$ corresponds to one bit within the bit-mask b_ω and
- an assignment $b_\omega[r_{m(\sigma)}(\mu)] = 0$ states that the model element μ is affected by the operation call ω as well as an assignment $b_\omega[r_{m(\sigma)}(\mu)] = 1$ states that the model element μ is not affected by the operation call ω.

Note that, if an operation call $\omega \in \Omega$ is *not* called for a transition, then the all bits of the corresponding bit-mask b_ω are set to 1 (stating that no model element is affected by the operation ω in this transition).

Then, the conjunction of all bit-masks for all model elements, represented by $\mathcal{B}$ and defined by

$$\bigwedge_{\mu \in m(\sigma)} \left(\mathcal{B}[r_{m(\sigma)}(\mu)] = \left(\bigwedge_{\omega \in \Omega} b_\omega[r_{m(\sigma)}(\mu)] \right) \right), \quad (5)$$

leads to a symbolic representation (i. e., for arbitrary operation calls) which model element $\mu \in m(\sigma)$ indeed is affected by the transition ($\mathcal{B}[r_{m(\sigma)}(\mu)] = 0$) and which model element is not affected by the transition ($\mathcal{B}[r_{m(\sigma)}(\mu)] = 1$).

Example 6. *Consider again the example from Fig. 5 with three instances of class* Counter *denoted by* Counter@0*,* Counter@1*,* Counter@2*. Additionally assume a transition in which two operations $\omega, \omega' \in \Omega$ with $\omega = (\text{Counter@0}, count)$ and $\omega' = (\text{Counter@1}, count)$ are called. Then, Table I shows the resulting bit-masks b_ω, $b_{\omega'}$, $b_{\omega''}$ for each operation. The bit-wise conjunction $\mathcal{B}$ of these bitmasks is shown at the bottom of Table I. From this, it can be concluded that, in this transition, the model elements* `Counter@0::value` *and* `Counter@1::value` *are affected (and, hence, restricted by the respective postconditions), while model element* `Counter@3::value` *is supposed to keep its value.*

Table I: Bit-masks for a transition

Operation call ω	Model element μ *Counter@0* ::value	*Counter@1* ::value	*Counter@2* ::value
(*Counter@0*, *count*) $b_\omega =$	0	1	1
(*Counter@1*, *count*) $b_{\omega'} =$	1	0	1
(*Counter@2*, *count*) $b_{\omega''} =$	1	1	1
$\mathcal{B} =$	0	0	1

Overall, this leads to a new formulation to be used in order to check concurrent behavior in UML/OCL models as follows:

Formulation 3. *For a model $m = (\mathcal{C}, \mathcal{R})$ and a sequence of system states $\sigma_0, \sigma_1, \ldots, \sigma_n$, let Ω be the set of all operation calls which can be conducted within one system state, $m(\sigma)$ the set of all model elements, and $\mathcal{B}_i$ as well as $b_{i,\omega}$ with $i = 0, \ldots, n-1$ and $\omega \in \Omega$ the bit-masks as introduced above. Furthermore, the (concurrent) operation calls whose parallel execution – conducted in a true-parallelism fashion – leads to the transition from a system state σ_i to a successor state σ_{i+1} are represented by the bit vector $\vec{\omega}_i$ of size $|\Omega|$. Then, each transition is symbolically represented by*

$$\bigwedge_{\omega \in \Omega} (\vec{\omega}_i[r_\Omega(\omega)] = 1) \Rightarrow \left(\begin{array}{c} [\![\lhd_\omega]\!] \wedge [\![\rhd_\omega]\!] \wedge [\![\mathcal{F}_\omega]\!]) \wedge \\ \bigwedge_{\mu \in m(\sigma_i)} b_{i,\omega}[r_{m(\sigma_i)}(\mu)] = \begin{cases} 0 & \text{if } \mu \text{ is affected by } \omega \\ 1 & \text{else} \end{cases} \end{array} \right) \quad (6)$$

which ensures that, if an operation call $\omega \in \Omega$ is called, its corresponding conditions are applied and the bit-mask $b_{i,\omega}[r_{m(\sigma)}(\mu)]$ is set accordingly and

$$(\vec{\omega}_i[r_\Omega(\omega)] = 0) \Rightarrow (b_{i,\omega} = 1 \ldots 1_2) \quad (7)$$

which ensures that, if an operation call $\omega \in \Omega$ is not called, the bit-mask $b_{i,\omega}[r_{m(\sigma)}(\mu)]$ is set to $1 \ldots 1$ (stating that ω does not affect any model element). Besides that, the number of concurrent calls is, again, restricted to be between d_1 and d_2, i. e.,

$$\bigwedge_{i=0}^{n-1} \left(d_1 \leq \sum_{k=0}^{|\Omega|-1} \vec{\omega}_i[k] \leq d_2 \right). \quad (8)$$

Finally, all model elements which are not covered by any operation call have to keep their value, which is ensured by

$$\bigwedge_{\mu \in m(\sigma)} \big(\mathcal{B}_i[r_{m(\sigma)}(\mu)] = 1\big) \Rightarrow (\sigma_i(\mu) = \sigma_{i+1}(\mu)). \quad (9)$$

From satisfying assignments, respective results can be determined as already discussed for the formulations before.

Example 7. *Consider again the example from Fig. 5 with its three instances of class* Counter *as well as the transition from a system state represented by* `State0` *to a succeeding system state represented by* `State1` *in which the operation calls* (Counter@0, `count`) *and* (Counter@1, `count`) *are called. Following the new formulation, the resulting SMT syntax of this transition with respect to the operation* (Counter@0, `count`) *reads as follows:*

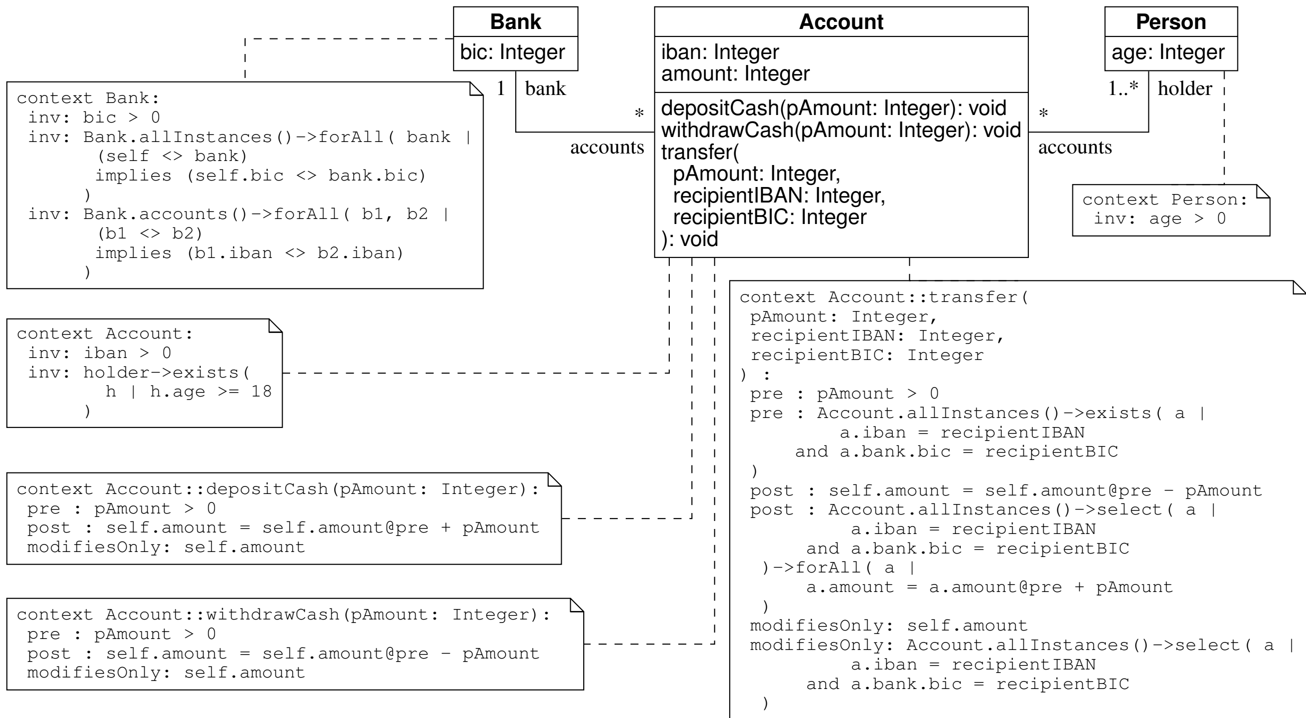

Figure 6: Considered model

```
1 (=> (= ((_ extract 0 0) omega) #b1)
2     (and (= State1::Counter0::value
3             (bvadd State0::Counter0::value
4                    #x01))
5          (b0 = #b110)))
6 (=> (= ((_ extract 0 0) omega) #b0)
7     (b0 = #b111))
8 (=> (= ((_ extract 1 1) omega) #b1)
9     (and (= State1::Counter1::value
10            (bvadd State0::Counter1::value
11                   #x01))
12         (b1 = #b110)))
13 (=> (= ((_ extract 1 1) omega) #b0)
14    (b1 = #b111))
15 (= B (bvand b0 b1 b2))
16 (=> (= ((extract 0 0) B) #b1)
17    (= State1::Counter0::value
18       State0::Counter0::value))
19 (=> (= ((extract 1 1) B) #b1)
20    (= State1::Counter1::value
21       State0::Counter1::value))
22 (=> (= ((extract 2 2) B) #b1)
23    (= State1::Counter2::value
24       State0::Counter2::value))
```

Overall, the resulting formulation allows for automatically checking the concurrent behavior of UML/OCL models. The problems discussed in Section IV are avoided by the introduction of an additional symbolic bit-mask representation together with corresponding constraints.

VI. Implementation and Application

In order to apply the proposed solution, we implemented the methodology described in the previous sections as an Eclipse plugin using both, Java and Xtend. As solving engine, we utilized Satisfiability Modulo Theories (SMT) [9] which has already been successfully applied in order to verify and validate UML/OCL models [21], [28]. Using the resulting implementation, the concurrent behavior of UML/OCL models following the design by contract scheme can be checked. In this section, the application of the resulting tool is illustrated by means of an example.

A. Considered Model

In order to illustrate the application, we considered a model of an international banking system in which money may concurrently be deposited into an account, withdrawn from an account, and transferred between accounts. The UML model and its OCL constraints are shown in Fig. 6. The system consists of three classes: *Person*, *Account* and *Bank*. Every Person has an age and can have multiple accounts by different banks. The account has attributes for its *International Bank Account Number* (`iban`), which identifies the account in the international banking system, and the current account balance `amount`. An account is always provided by a bank. Every Bank has a unique *Business Identifier Code* (`bic`). In this model, `bic` and `iban` are represented by a unique positive number. In addition, the class *Account* has the following operations:

- `depositCash`: A person can call this operation to deposit money at the bank. The current balance is increased by the amount given by the parameter (which has to be a positive number).
- `withdrawCash`: A person can call this operation to withdraw cash. The current balance of the account is decreased by the amount given by the parameter (which has to be a positive number).
- `transfer`: A person can call this operation to transfer money from his/her account to another account. The recipient account is identified by its `iban` and the

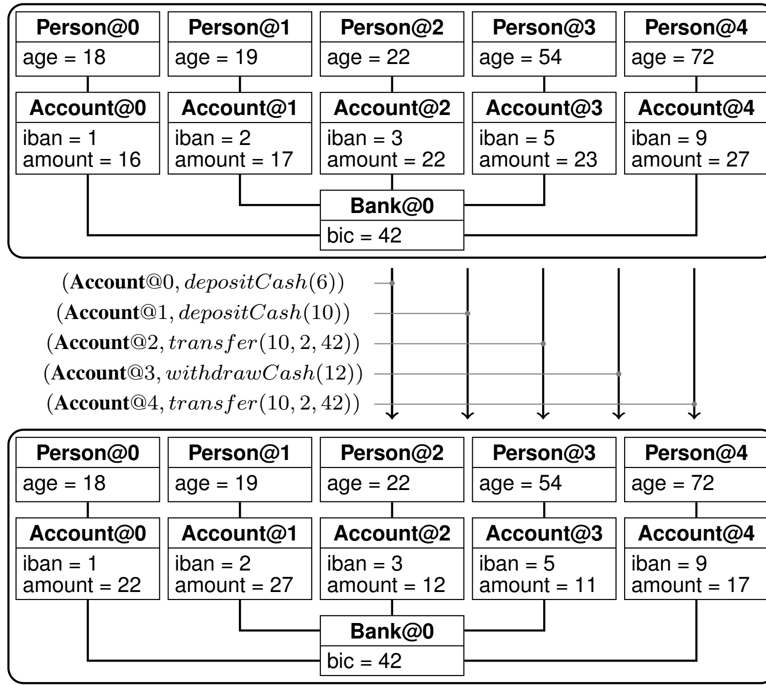

Figure 7: Considered sequence

corresponding `bic`. The current balance of the account calling this operation is decreased by *pAmount*, while the account of the recipient is increased by this amount.

For our evaluation, we consider an instantiation of this model composed of one bank and five persons where each of them has one account at the bank. We additionally consider one transition with at most five concurrent operation calls.

B. Application

Using the proposed methodology, various checks can be conducted on the considered model/instantiation. These checks can be divided into two classes: universal and problem specific checks. As a typical example for a universal check, we were able to validate that the considered configuration does not run into deadlocks, i. e., if a valid initial state is assumed, no sequence of transitions exists that leads to a system state out of which no further operation calls are possible anymore. In order to check that, we automatically generated the symbolic formulation proposed above for a total of 50 transitions and, additionally, constrained the terminal state σ_{50} accordingly (e. g., by explicitly prohibiting attribute values which may satisfy a precondition of an operation). Afterwards, we passed the resulting formulation to a SMT solver which proved that no satisfying assignment exists for this instance. From this result, it can be concluded that no deadlock state can be reached from an arbitrary (but valid) system state within 50 transitions. In addition to previously conducted consistency checks and by following k-induction [39], this allows for the conclusion that the behavior of the considered model is free of deadlocks.

On the other side, also model (or problem) specific checks can be applied. For the given banking system such a check would be, e. g., that the overall amount of money with respect to deposits and withdrawals stays the same. Executing this check, we obtain the sequence as shown in Fig. 7 in which five concurrent operation calls are conducted. The upper part of Fig. 7 shows the obtained (arbitrary but valid) initial system state, while the bottom shows the resulting state of the transition when the five operation calls depicted in the middle of Fig. 7 are invoked. Although all constraints of the model are satisfied by this sequence, it is very likely that this does not represent the desired behavior: This is because *Person@2* as well as *Person@4* transfer 10 money units to the account of *Person@1*, and *Person@1* additionally deposits another 10 money units to his/her account, but the total value of *Person@1::amount* increases by 10 only.

Due to the problem specific check, the designer has been pin-pointed to a serious modeling error which can easily be fixed afterwards in a manual fashion. In this particular case, the designer could, e. g., add auxiliary variables for critical sections to model locking mechanisms or add postconditions which avoid different amounts of the overall money.

Using the methodology proposed here, all these checks can automatically be conducted within negligible run-time on a modern computer (i. e., less than a second) and, hence, provide useful aid to the designers of such models.

VII. Conclusion

In this work, we proposed a methodology for the validation and verification of concurrent behavior which is possible based on the design by contract descriptions provided in UML/OCL models. To this end, a computational model based on a “true-parallelism”-scheme is assumed and a symbolic formulation representing arbitrary concurrent operation calls has been proposed. Solvers for SAT Modulo Theories are then capable of addressing various checks for a respectively

given UML/OCL model. The addressed problems have been described and illustrated by means of a running example and an exemplary application demonstrated how designers are supported by the resulting tool.

ACKNOWLEDGEMENTS

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SPECifIC under grant no. 01IW13001, the German Research Foundation (DFG) within the Reinhart Koselleck project under grant no. DR 287/23-1 and a research project under grant no. WI 3401/5-1, the Graduate School SyDe funded by the German Excellence Initiative within the University of Bremen's institutional strategy as well as the Siemens AG.

REFERENCES

[1] J. Rumbaugh, I. Jacobson, and G. Booch, Eds., *The Unified Modeling Language reference manual*. Essex, UK: Addison-Wesley Longman Ltd., 1999.
[2] OMG – Object Management Group, "Object Constraint Language," 2014, version 2.4, February 2014. [Online]. Available: http://www.omg.org/spec/OCL/2.4
[3] P. Kosiuczenko, "Specification of Invariability in OCL," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, 2006, pp. 676–691.
[4] ——, "Specification of invariability in OCL - Specifying invariable system parts and views," *Software and System Modeling*, vol. 12, no. 2, pp. 415–434, 2013.
[5] P. Niemann, F. Hilken, M. Gogolla, and R. Wille, "Extracting frame conditions from operation contracts," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, 2015.
[6] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*, 2006.
[7] S. A. Cook, "The Complexity of Theorem-Proving Procedures," in *Proceedings of the Symposium on Theory of Computing*, M. A. Harrison, R. B. Banerji, and J. D. Ullman, Eds., 1971, pp. 151–158.
[8] R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays," in *Tools and Algorithms for Construction and Analysis of Systems*, 2009, pp. 174–177.
[9] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0,," 2010. [Online]. Available: https://www.smt-lib.org
[10] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 27–34, 2007.
[11] 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*, vol. 115, pp. 39–47, 2005.
[12] A. D. Brucker and B. Wolff, "A Proposal for a Formal OCL Semantics in Isabelle/HOL," in *TPHOLs*, 2002, pp. 99–114.
[13] B. Beckert, R. Hähnle, and P. H. Schmitt, *Verification of Object-Oriented Software: The KeY Approach*, 2007.
[14] J. Cabot, R. Clarisó, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," in *Proceedings of Conference on Software Testing Verification and Validation*, 2008, pp. 73–80.
[15] T. Mancini, "Finite satisfiability of UML class diagrams by constraint programming," in *Description Logics*, 2004.
[16] H. Malgouyres and G. Motet, "A UML model consistency verification approach based on meta-modeling formalization," in *Proceedings of the Symposium on Applied computing*, 2006, pp. 1804–1809.
[17] D. Berardi, D. Calvanese, and G. De Giacomo, "Reasoning on UML class diagrams," *Artif. Intell.*, vol. 168, no. 1, pp. 70–118, 2005.
[18] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers, "Using description logic to maintain consistency between UML models," in *UML*, 2003, pp. 326–340.
[19] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A Challenging Model Transformation," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, 2007, pp. 436–450.
[20] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, O. Grumberg and M. Huth, Eds., vol. 4424. Springer, 2007, pp. 632–647.
[21] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using Boolean satisfiability," in *Design, Automation and Test in Europe*, 2010, pp. 1341–1344.
[22] D. Harel and A. Naamad, "The statemate semantics of statecharts," *Transactions on Software Engineering and Methodology*, vol. 5, no. 4, pp. 293–333, 1996.
[23] D. Latella, I. Majzik, and M. Massink, "Towards a formal operational semantics of UML statechart diagrams," in *Int'l Conf. Conference on Formal Methods for Open Object-Based Distributed Systems*, 1999.
[24] W.-l. Huang, J. Peleska, and U. Schulze, "Test automation support," COMPASS Comprehensive Modelling for Advanced Systems of Systems, Tech. Rep. D34.1, 2013, available under `http://www.compass-research.eu/deliverables.html`.
[25] J. Peleska, "Industrial-strength model-based testing - state of the art and current challenges," in *Workshop on Model-Based Testing*, ser. Electronic Proceedings in Theoretical Computer Science, vol. 111, 2013, pp. 3–28.
[26] C. Hilken, J. Peleska, and R. Wille, "A Unified Formulation of Behavioral Semantics for SysML Models:," in *Int'l Conf. on Model-Driven Engineering and Software Development*, 2015, pp. 263–271.
[27] F. Hilken, L. Hamann, and M. Gogolla, "Transformation of UML and OCL models into filmstrip models," in *Int'l Conf. on Theory and Practice of Model Transformations*, ser. Lecture Notes in Computer Science, vol. 8568, 2014, pp. 170–185.
[28] M. Soeken, R. Wille, and R. Drechsler, "Verifying Dynamic Aspects of UML models," in *Design, Automation and Test in Europe*, 2011, pp. 1077–1082.
[29] P. Hilken, Frankand Niemann, M. Gogolla, and R. Wille, "Filmstripping and unrolling: A comparison of verification approaches for UML and OCL behavioral models," in *Tests and Proof*, 2014, pp. 99–116.
[30] A. W. Roscoe, C. A. R. Hoare, and R. Bird, *The Theory and Practice of Concurrency*, 1997. [Online]. Available: https://www.cs.ox.ac.uk/people/bill.roscoe/publications/68b.pdf
[31] J. F. Groote and M. R. Mousavi, *Modeling and Analysis of Concurrent Systems*, 2014.
[32] J. Lilius and I. Paltor, "vUML: A Tool for Verifying UML Models," in *Int'l Conf. on Automated Software Engineering*, 1999, pp. 255–258.
[33] A. Knapp and S. Merz, "Model checking and code generation for UML state machines and collaborations," *Workshop on Tools for System Design and Verification*, pp. 59–64, 2002.
[34] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.
[35] T. S. Staines, "Intuitive Mapping of UML 2 Activity Diagrams into Fundamental Modeling Concept Petri Net Diagrams and Colored Petri Nets," in *Int'l Conf. on Workshop on Engineering of Computer Based Systems*, 2008, pp. 191–200.
[36] G. Engels, C. Soltenborn, and H. Wehrheim, "Analysis of UML Activities Using Dynamic Meta Modeling," in *Int'l Conf. Conference on Formal Methods for Open Object-Based Distributed Systems*, ser. Lecture Notes in Computer Science, vol. 4468, 2007, pp. 76–90.
[37] G. Winskel and M. Nielsen, "Models for concurrency," *DAIMI Report Series*, vol. 22, no. 463, 1993.
[38] V. Sassone, M. Nielsen, and G. Winskel, "Models for concurrency: Towards a classification," *Theor. Comput. Sci.*, vol. 170, no. 1-2, pp. 297–348, 1996.
[39] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Conference on Formal Methods in Computer-Aided Design*, 2000, pp. 108–125.