Frame Conditions in Symbolic Representations of UML/OCL Models

Nils Przigoda[1,2] Jonas Gomes Filho[1] Philipp Niemann[1] Robert Wille[3,2] 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]Institute for Integrated Circuits, Johannes Kepler University Linz, 4040 Linz, Austria
{przigoda,gomes.filho,pniemann,drechsle}@informatik.uni-bremen.de robert.wille@jku.at

***Abstract*—Verification and validation of UML/OCL models is a crucial task in the design of complex software/hardware systems. The behavior in those models is expressed in terms of operations with pre- and post-conditions. These, however, are often not precise enough to describe what may or may not be modified in a transition between two system states. This frame problem is commonly addressed by providing additional constraints in terms of so-called frame conditions and has already been considered in different research areas in the last decades – except for UML/OCL where corresponding approaches have been investigated only recently. Besides that, several approaches for the verification of the behavior specified in UML/OCL models have been proposed. They rely on a symbolic representation of all possible system states and transitions between them. But here, frame conditions have not been considered yet – a significant drawback for the underlying verification approaches. In this paper, we describe how to integrate frame conditions to symbolic representations. This enables designers to verify the behavior of UML/OCL models while, at the same time, respecting the given frame conditions.**

I. Introduction & Motivation

The *Unified Modeling Language* (UML) [1] is the de facto standard modeling language for software development, but its high level of abstraction also allows for the modeling of complex systems in general. In addition, 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. This way, it is, e. g., possible to restrict valid system states by invariants or to describe the behavior of operations by means of pre- and postconditions. The combination of UML and OCL is a promising abstraction in order to deal the complexity of today's electronic system. Thus, researchers started to investigate the integration of modeling languages in the design of hardware systems such as embedded systems [3].

However, particularly the descriptions of operations, i. e., pre- and postconditions, are often not precise enough to describe what may or may not be modified in a transition between two system states. While they usually allow to define restrictions of the calling and the succeeding system state, they are unsuitable to describe which parts of the model are allowed to change inside a frame. Recently, this *frame problem* has been addressed by providing additional constraints in terms of so-called *frame conditions*. Frame conditions describe which model elements may change during the transition from one system state to another and, thus, avoid unintended behavior.

A crucial step in the design process of a complex system is its verification, i. e., checking whether the derived model is correct. Several approaches have considered the verification of behavioral aspects of UML/OCL models using methods such as *Constraint-Satisfaction-Problem* (CSP) solvers [4], *Boolean Satisfiability* (SAT) [5] or the extended *SAT Modulo Theories* (SMT) [6], [7], among others. Most of these verification methodologies directly or indirectly rely on a symbolic representation of all possible system states and transitions between them. Nevertheless, previous works have either ignored the problem of frame conditions or treated it manually, which is a significant drawback for the underlying verification approaches.

In this work, a solution for an explicit integration of frame conditions in a symbolic representation is described and illustrated by means of a running example. Furthermore, a corresponding SAT/SMT-based realization is presented which enables designers to verify the behavior of UML/OCL models while, at the same time, respecting the given frame conditions.

II. Preliminaries and Notation

In order to keep the paper self-contained, we briefly review basic concepts of UML/OCL models and introduce the formal notation used in this work. After that, we explain the running example that will be used through the whole paper to illustrate the presented concepts in terms of the corresponding SMT constraints. For this purpose, we finally introduce the basics and notions of SMT.

A. UML/OCL

The *Unified Modeling Language* (UML) together with the *Object Constraint Language* (OCL) allows for modeling complex systems at an abstract level without the need to provide detailed implementations. While the general structure and behavior of the system is expressed graphically in terms of UML (class) diagrams, textual OCL constraints are used in order to add further restrictions. OCL is a declarative language that mainly consists of logic, arithmetic, navigation, and collection expressions. A comprehensive overview of all OCL expressions, its keywords and the semantic definitions, is given in [2].

From a formal perspective, a *model* $m = (\mathcal{C}, \mathcal{R})$ is a tuple of *classes* $\mathcal{C}$ and *relations* $\mathcal{R}$ (also known as *associations*). A *class* $c \in \mathcal{C}$ is a 3-tuple composed of sets of attributes A (of a distinct type), invariants I, and operations O. An *operation* $o \in O$ is a 4-tuple $o = (P, r, \lhd, \rhd)$ composed of a set of parameters P, a return value r, as well as sets of pre- and postconditions (denoted by $\lhd$ and $\rhd$, respectively).

A *relation* $r = (c_1, c_2, (l_1, u_1), (l_2, u_2)) \in \mathcal{R}$ between two classes $c_1, c_2 \in \mathcal{C}$ (not necessarily different) has multiplicity constraints, i. e., lower bounds l_i and upper bounds u_i that restrict how often the relation is allowed to be instantiated for each object of the class c_i ($i = 1, 2$). All relations of a model together with the union of all attributes (of all its classes) form the set of *model elements*.

UML/OCL models represent blueprints for possible instantiations of a system. In terms of UML, these instantiations are commonly given as *object diagrams*. Formally, an instantiation can be represented in terms of a *system state* $\sigma = (\Upsilon, \Lambda)$ composed of a set of objects Υ (instantiations of classes from $\mathcal{C}$ with a unique identifier) and a set of links Λ (instantiations of relations from $\mathcal{R}$). To this end, the set containing all instances of model elements is denoted by $m(\sigma)$.

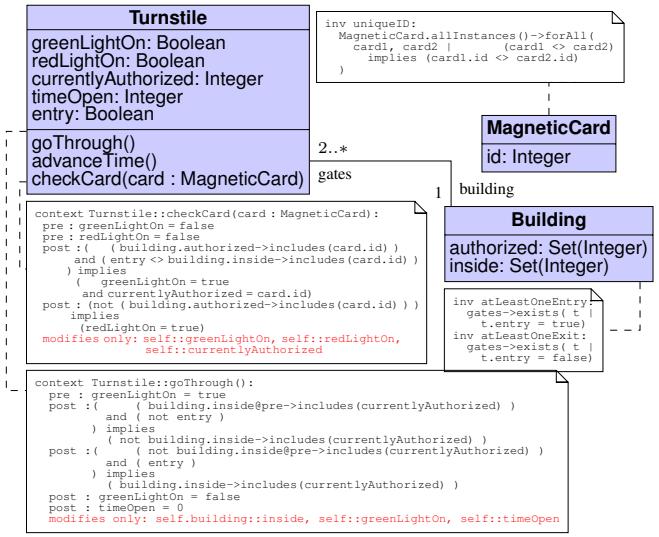

Figure 1: Class diagram of the access control system

A system can evolve from one state σ to another state σ' as the result of an *operation call*, i. e., by calling an operation on one of σ's objects. Formally, an operation call is denoted by $\omega = (\upsilon, o)$, where υ is an object instance in σ of a class $c \in \mathcal{C}$ with an operation o. Moreover, the set of all operation calls Ω for a system state σ is determined by

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

where $\mathrm{ops}(c)$ denotes the set of all operations of a class $c \in \mathcal{C}$ and $\Upsilon(c)$ denotes the set of all object instances of a class $c \in \mathcal{C}$ in a given system state σ.

B. Running Example

All UML/OCL concepts reviewed above are illustrated using the model given in Fig. 1. In the remainder of the paper, this model will be used in order to illustrate the problem as well as the proposed solution. The model, originally presented in [8], specifies a control system which grants access to buildings based on magnetic cards as authentication method. The cards are checked at turnstiles at the buildings' entries.

The model comprises three classes, namely *Building*, *MagneticCard*, and *Turnstile*. For instance, the attributes, invariants and operations of the class *Building* are formally interpreted as the sets $A = \{\texttt{authorized}, \texttt{inside}\}$, $I = \{\texttt{atLeastOneEntry}, \texttt{atLeastOneExit}\}$, and $O = \emptyset$. In the entire model, there is one relation/association, which relates turnstiles to the corresponding building ensuring that each *Building* has at least two *Turnstiles*. Formally, this relation is given by the 4-tuple $(\mathit{Building}, \mathit{Turnstile}, (1,1), (2,\infty))$. The set of model elements contains all eight attributes of the three classes and the relation.

The behavior of the operations of the class *Turnstile* is defined by means of pre- and postconditions as follows:[1]

- The operation `checkCard` checks whether a person inserting a *MagneticCard* may pass the respective turnstile and
- the operation `goThrough` allows the currently authorized person to pass the turnstile.

[1]The "modifies only" conditions (highlighted in red) will be covered later in Section IV.

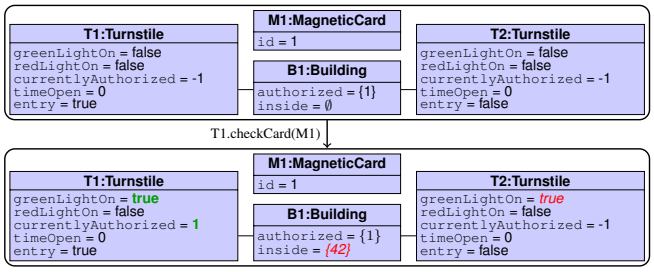

Figure 2: A valid transition between two valid system states

Two possible instantiations of the model, i. e., system states, representing a single building and two turnstiles are shown in Fig. 2. As indicated there, calling the operation `checkCard(M1)` on `T1` leads to the transition from the system state depicted on the top of Fig. 2 to the system state depicted on the bottom of Fig. 2.

C. Boolean Satisfiability and Satisfiability Modulo Theories

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

Example 1. *Let* $f(x_1, x_2, x_3) = (x_1 + \overline{x}_2 + \overline{x}_3) \cdot (\overline{x}_1 + x_2) \cdot (\overline{x}_2 + x_3)$ *where* $+$ *and* $\cdot$ *denote the logical OR and AND operation, respectively. Then,* $x_1 = x_2 = x_3 = 1$ *is a satisfying assignment for* f*. The value of* x_1 *ensures that the first clause becomes satisfied, the value of* x_2 *does that for the second clause, and so on.*

Satisfiability Modulo Theories (SMT) is an extension of SAT which allows to formulate decision problems at a higher level of abstraction. In this work, we will make use of the theory of *Quantifier-free Bit Vectors* (QF_BV) which allows to work on bit vector logic rather than pure Boolean logic. In the following, we will provide SMT formulas using the SMT-LIB syntax [9] where each constraint is provided in Polish notation, i. e., each operation is encapsulated by parentheses and the operator is provided before the (ordered) list of operands.

Example 2. *Consider the following SMT formulas over two bit-vectors* `bv1`*,* `bv2`*:*

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

The first formula is composed of two operations, namely negation (`not`*) and equivalence (*`=`*) and states that the bit vectors* `bv1` *and* `bv2` *shall differ in at least one position, i. e., in a symbolical notation* $\texttt{bv1} \neq \texttt{bv2}$*. The second formula illustrates how to access individual bits of a bit vector, namely using the* `extract` *operation with the indices of the first and last extracted bit being equal, as well as how to use constant values (*`#b...`*). The constraint enforces that the second last bit*[2] *of the bit vector* `bv2` *is set to 1. Solving this SMT instance may lead to the assignments* `bv1 = #b1101` *and* `bv2 = #b1010`*.*

III. Related Work and Considered Problem

This section reviews related work on frame conditions and discusses the use of symbolic representations for verification and validation of behavioral UML/OCL models. Based on that, the problem considered in this work is explicitly stated.

[2]The bits of a bit vector are enumerated from right to left starting with 0.

A. Frame Problem and Frame Conditions

In UML/OCL class diagrams, behavior can be expressed in terms of operations with pre- and postconditions. At a first glance, these declarative descriptions of the operation's behavior ideally fit to the paradigm of designing systems without the need to provide detailed implementations. However, when we look more closely, they may allow for undesired behavior.

Example 3. *Consider again Fig. 2 which shows two system states of our running example from Fig. 1. Both system states are valid, i. e., all model constraints are satisfied. Moreover, it is possible to get from the system state in the top to the one in the bottom, i. e., an operation whose preconditions (postconditions) are satisfied in the top (bottom) system state. As intended by the operation's postconditions, the green light of turnstile* `T1` *is turned on by the operation call. However, at the same time it would also be possible to to turn on the green light of the other turnstile* `T2` *and to add an arbitrary id (e. g., 42) to the* `inside` *attribute of the building* `B1` *(as highlighted in red in Fig.2). Although such a behavior is obviously not intended, it is completely in line with the postconditions.*

In general, the shortcoming of declarative descriptions like pre- and postconditions is that they often do not make clear enough which model elements are allowed to change during an operation call. In other words: they do not specify what is within the *frame* that might be modified by an operation – the so-called *frame problem* [10]. In order to address the resulting under specification of the model, so-called *frame conditions* have been proposed. In the past, various approaches have been developed for this purpose for UML/OCL models:

- A straightforward approach is to manually specify what is *not* in the frame (as it is done in [11]) by extending the postconditions with constraints like `modelElement = modelElement@pre`. This case study, however, illustrates very impressively how time-consuming it is to create these constraints in the first place and to maintain them later on in the case of design changes.
- Another approach to frame conditions is to automatically derive them from the postconditions using a paradigm such as *nothing else changes* [12], [13] which includes every model element that is referenced within postconditions in the frame of what may change (and nothing else). Again, the results of this implicit approach are also often not exactly what the designer intended and, besides that, it is hard to manually adjust them – which would have to be done by rewriting the postconditions themselves.
- Finally, a more thorough approach has been suggested by Kosiuczenko [14], [15]. The idea is to specify the set of variable model elements, i. e., model elements that are allowed to be changed during an operation call, at the same level as pre- and postconditions in terms of so-called *invariability clauses*[3] which are of the form `modifies only: scope::modelElement`. For instance, the clause `modifies only: self::greenLightOn` expresses that the operation may only change the attribute `greenLightOn` of the turnstile on which the operation is called (`self`). However, the scope can also be more complex and may contain navigation or collections. Besides that, it is even possible to allow objects of a certain class to be created or deleted during an operation call using the construct `Class::allInstances()`.
  This approach enables the designer to precisely define frame conditions in a very comfortable, understandable and maintainable fashion. Moreover, there exists a methodology to assist the designer in the initial generation of the frame conditions [17] and an approach that does most of the work automatically and requests feedback of the designer in ambiguous cases only [18].

Overall, frame conditions are very important for obtaining precise model descriptions and are a key ingredient when considering behavior of UML/OCL models.

B. Validation and Verification Using Symbolic Representations

UML/OCL allows to explicitly specify the design of complex systems without a precise implementation. At this high-level of abstraction, however, descriptions might result which are, e. g., over-constrained such that no valid system state can be derived (inconsistent models) or in which some operations could never be executed due to too restrictive pre- and postconditions. But even if this is not the case, the specification may still allow for reaching "bad states" such as deadlocks or other unwanted behavior, e. g., due to inadequate frame conditions.

In the recent past, several approaches for the validation and verification of behavioral models have been proposed. In order to verify the model, SAT/SMT-based approaches such as introduced in [5], [6] do not rely on explicitly enumerating all possible system states and operation calls. Instead, they utilize a symbolic representation of the given UML/OCL model which allows to consider all possible system states and operation calls at the same time.[4] This symbolic representation consists of a set of variables which can describe an arbitrary system state and arbitrary transitions (also invalid ones). Passing this problem instance to a solver would yield an arbitrary behavior of the system. Consequently, additional constraints are applied to enforce that all system states and transitions are valid and, beyond that, satisfy the considered verification objective.

In this work, we rely on the solution presented in [6]. Here, the authors propose to translate the verification task into an instance of a *Satisfiability Modulo Theories* (SMT) problem. The corresponding symbolic representation is created in terms of the SMT bit-vector logic QF_BV (cf. Section II-C). Then, the problem instance can be solved using so-called SMT solvers. These solvers allow for an efficient traversal of large search spaces and, hence, are suitable to determine precise assignments to the symbolic formulation and, by this, a sequence of transitions satisfying the considered verification objective.

C. Considered Problem

As reviewed in the previous section, powerful approaches for verifying and validating the behavior of UML/OCL models using symbolic representations have been suggested. However, to the best of our knowledge, all of these approaches either require *frame conditions* to be specified manually within the postconditions or apply an implicit interpretation in order to automatically extract them from the model – with all the weaknesses and drawbacks already discussed above in Section III-A.

So far, there is no approach for behavioral verification that supports *explicit* frame conditions as, e. g., provided by the "modifies only" clauses. This is mainly caused by the non-availability of translations from these frame conditions into the corresponding symbolic representation. Consequently, in the remainder of this work we consider the dedicated translation of explicit frame conditions and their integration into (existing) symbolic representations. We keep our description generic such that it can be used for any means of symbolic

[3] A variation of this idea is to specify the set of variable model elements within the postconditions using an OCL primitive `modifiedOnly(Set)` [16].

[4] However, all these approaches require so-called *problem bounds* in order to limit the search space, i. e., they need to be provided with a fixed number or at least a range of objects that shall be instantiated as well as a finite domain for all data types.

representation. Nonetheless, we will always present precise formulations in SMT in order to illustrate the method.

The resulting symbolic representation will, for the first time, allow for behavioral verification of designs while, at the same time, respecting frame conditions. Moreover, besides conventional verification tasks, the proposed solution can additionally be used to validate the frame conditions themselves, i. e., to check whether they are consistent (do not completely prohibit the execution of the corresponding operation) or whether they express what the designer intended.

IV. Symbolic Representation of Frame Conditions

In this section, we will first review how sequential behavior, i. e., a transition with exactly one operation call between two system states, is translated into a symbolic representation. Based on this, we will explain, how explicit frame conditions can be added to the symbolic representation.

A. Symbolic Representation of a State Transition

In [6], [7], the authors explain how a sequence of operation calls can symbolically be formulated. Here, we will just review the idea for the formulation of a single transition representing one operation call between two system states.

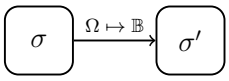

Figure 3: State transition

The general idea is sketched by means of Fig. 3: Two system states, σ, σ', are connected with a transition. In order to describe which operation call is executed on the transition, a helper function from the set of all possible operation calls Ω to $\mathbb{B}$ is used. Furthermore, for all instantiated model elements of both system states, a set of variables representing the respective values, e. g., of attributes or relations are introduced.

Then, it remains open how to explicitly assign all variables so that, with respect to the corresponding pre- and postconditions of a respective operation call, a valid transition from σ to σ' is derived. More precisely:

Formulation 1. *For a model $m = (\mathcal{C}, \mathcal{R})$ and two system states σ and σ', let Ω be the set of all operation calls which can be performed for the transition between σ and σ'. Further, let $\alpha : \Omega \mapsto \mathbb{B}$ be a map indicating if an operation call is executed or not. Then, all possible operation calls for the transition are symbolically represented by*

$$\bigwedge_{\omega \in \Omega} (\alpha(\omega) = 1) \Rightarrow ([\![\triangleleft_\omega]\!] \wedge [\![\triangleright_\omega]\!]) \tag{1}$$

where

- *$[\![\triangleleft_\omega]\!]$ is a constraint enforcing the precondition for system state σ, and*
- *$[\![\triangleright_\omega]\!]$ is a constraint enforcing the postcondition for system state σ' (maybe by using σ as well).*

Besides that, the number of possible operation calls is restricted by $\left(\sum_{\omega \in \Omega} \alpha(\omega)\right) = 1$ in order to ensure that only one operation call is executed on the transition.[5]

Using this formulation, a satisfying assignment to all used variables including the exact mapping for α must exist if a transition from the calling system state σ to the terminal system state σ' exists. From this assignment, the respective operation call for the transition can eventually be obtained. If no such assignment exists, it has been proven that a transition between the two given system states does not exist.

In [6], Formulation 1 is realized in SMT-LIB syntax in terms of the QF_BV theory and passed to an SMT solver. Those solvers are capable of determining such assignments or proving their non-existence in an efficient fashion. More precisely, the formulation for a single transition is applied as follows:

Example 4. *Consider again the running example model as given in Fig. 1. The α function for the transition is represented by the bit vector $\vec{\omega}$ of size $\lceil \log_2 |\Omega| \rceil$,*[6] *where each possible assignment is related to exactly one operation call. More precisely, the possible operation calls are counted beginning from 0 and, therefore, the possible assignment is restricted by: $\vec{\omega} < |\Omega|$. Then, the constraint for the operation call $\omega = (T1, checkcard(M1))$ read as follows:*

```
1 (=> (= ω⃗ #b000)
2     (and (= σ::T1::greenLightOn false)
3          (= σ::T1::redLightOn false)
4          ... ) )
```

The first line realizes the left-hand side of the implication sketched in Eq. (1) *(assuming that* 000 *is the unique identifier of ω). Afterwards, the preconditions of the operation, represented by the bitvector* 000*, are enforced (see Line 2–3). Hereby, the system states are represented by the assignments to the respective attributes of the instantiated classes. For example, the value of the attribute* `greenLightOn` *of the object instance* T1 *of the class* Turnstile *is represented by* `σ::T1::greenLightOn` *for the calling system state and* `σ'::T1::greenLightOn` *for the succeeding system state, respectively. Based on that, the two preconditions (checking the values of* `greenLightOn` *and* `redLightOn`*) are enforced by the SMT constraint in Line 2–3. The postconditions are realized in a similar fashion.*

B. Adding Constraints for Frame Conditions

To support frame conditions in symbolic representations, a description of their effect on the model elements has to be added. To this end, further variables are introduced which symbolically represent whether a respective model element is supposed to be affected by a transition or not. This is described by a set of functions which map each instantiated model element to a Boolean value indicating if the corresponding element is allowed to change or not.

Formulation 2. *For the impact of frame conditions on a transition, three different types of* variability maps *$\mathcal{V}$ from $m(\sigma)$ to $\mathbb{B}$ are created. If $\mathcal{V}(\mu)$ evaluates to* `true`*, then the value of the model element μ is variable during the operation execution. Otherwise, it is not allowed to change its value.*

Since the evaluation of each explicit frame condition statement depends on the scope which itself mostly depends on the calling object, the impact for each possible frame conditions statement must be considered. Thus, $\mathcal{F}_\omega$ denotes the set of all frame conditions of the operation call ω. Then, the maps:

$$\begin{aligned} \mathcal{V} &: m(\sigma) \mapsto \mathbb{B}, \\ \forall\, \omega \in \Omega : \mathcal{V}_\omega &: m(\sigma) \mapsto \mathbb{B}, \text{and} \\ \forall\, \omega \in \Omega : \forall\, f \in \mathcal{F}_\omega : \mathcal{V}_{\omega,f} &: m(\sigma) \mapsto \mathbb{B} \end{aligned}$$

are introduced, whereby

1) *$\mathcal{V}$ represents whether a model element μ is affected in general by a transition,*
2) *$\mathcal{V}_\omega$ represents whether a model element μ is affected by the operation call ω, and*
3) *$\mathcal{V}_{\omega,f}$ represents whether the model element μ is affected by the frame condition $f \in \mathcal{F}_\omega$ of the operation call ω.*

Example 5. *Consider again the running example: Each map is represented with a bit mask of length $|m(\sigma)|$, where each bit represents the result of the map regarding a unique model element. Since the system state given in Fig. 2 has 16 instantiated*

[5] In both system states also the invariants have to be satisfied. But since this is not the focus of this work, details are omitted for brevity.

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

model elements, each map requires a bit mask with 16 bits. As stated in Formulation 2, maps are needed for the overall frame conditions of the transition, the six different operation calls, and within them for each frame condition statement, leading to a total of 25 maps.

The maps allow for a representation about what model elements are to be changed during a transition. Now this information has to be explicitly employed to the symbolic formulation. To this end for the constraints have to be added. First, constraints for the overall constraint map, i. e., $\mathcal{V}$, are added:

Formulation 3. *The instance of a model element $\mu \in m(\sigma)$ belonging to an attribute[7] of a class is variable by the transition if $\mathcal{V}(\mu) = 1$ holds and it is not allowed to change its value if $\mathcal{V}(\mu) = 0$. This yields the following constraint:*

$$\bigwedge_{\substack{\mu \in m(\sigma) \\ \mu \text{ belongs to attribute}}} (\mathcal{V}(\mu) = 0) \Rightarrow (\sigma(\mu) = \sigma'(\mu)) \,, \qquad (2)$$

where $\sigma(\mu)$ represents the precise value of μ in system state σ.

Example 6. *Assume that, in the running example, the attributes* `greenLightOn` *and* `redLightOn` *of the object instance* T1 *are related to bit number 0 and 1 of the bit mask $\mathcal{V}$ representing the overall frame conditions map. Then, the precise SMT constraints obtained by Eq.* (2) *are:*

```
(and (=> (= ((_ extract 0 0) V) #b0)
         (= σ::T1::greenLightOn
            σ'::T1::greenLightOn))
     (=> (= ((_ extract 1 1) V) #b0)
         (= σ::T1::redLightOn
            σ'::T1::redLightOn)) )
```

As a next step, it should be ensured that the overall frame condition map equals the frame condition map of the executed operation call. For this purpose, a slight modification of Eq. (1) is required:

Formulation 4. *To support frame conditions in a symbolic representation, Eq.* (1) *is refined to:*

$$\bigwedge_{\omega \in \Omega} (\alpha(\omega) = 1) \Rightarrow ([\![\lhd_\omega]\!] \wedge [\![\rhd_\omega]\!] \wedge \mathcal{V} = \mathcal{V}_\omega) \,.$$

Since only one of the premises of the implications for the operation calls can be satisfied, only satisfying assignments are possible in which the overall variability map will be equal to the overall variability map of the corresponding operation call. In the next step, the overall variability map is defined:

Formulation 5. *If the disjunction of the values of all variability maps $\mathcal{V}_{\omega,f}$ for a fixed ω and a fixed instance of a model element μ is* `true`*, then the variability map $\mathcal{V}_\omega$ should also state the model element is variable. This is ensured by:*

$$\forall\, \mu \in m(\sigma) : \bigwedge_{\omega \in \Omega} \left(\mathcal{V}_\omega(\mu) = \bigvee_{f \in \mathcal{F}_\omega} \mathcal{V}_{\omega,f}(\mu) \right) .$$

Example 7. *For the operation call* T1.`checkCard` *of the running example, the following constraints in the SMT-LIB format are added:*[8]

```
(= V::T1::cC
   (bvor V::T1::cC::fc1
         V::T1::cC::fc2
         V::T1::cC::fc3))
```

Finally, a precise definition of all $\mathcal{V}_{\omega,f}$ respecting the corresponding frame condition f given by `modifies only`-statements must be given.

Formulation 6. *Let f be a frame condition of an operation o and $\omega = (\upsilon, o)$ a corresponding possible operation call. Then, $\mathcal{V}_{\omega,f}$ is defined for each instantiated model element μ as follows:*

$$\bigwedge_{\mu \in m(\sigma)} \Big| \begin{array}{l} \textbf{if} \;\; \text{the object of } \mu \text{ is in the scope of } f \\ \quad \textbf{and} \;\; \mu \text{ is an instance of the model element of } f \\ \textbf{then} \;\; \mathcal{V}_{\omega,f}(\mu) = 1 \\ \textbf{else} \;\; \mathcal{V}_{\omega,f}(\mu) = 0 \\ \textbf{endif} \end{array}$$

Example 8. *For the operation call* T1.`checkCard` *of our running example, consider the three frame conditions:*

```
modifies only: self::greenLightOn,
               self::redLightOn,
               self::currentlyAuthorized
```

In all three frame conditions the scope is `self`*, which is* T1 *for the assumed operation call. Assuming again that* T1.`greenLightOn` *has the index 0 in the corresponding bit mask of $\mathcal{V}_{\omega,f}$, the following SMT constraints are generated for the first frame condition:*

```
(= ((_ extract 0 0) ω::T1::cC::fc1) #b1)
(= ((_ extract 1 1) ω::T1::cC::fc1) #b0)
(= ((_ extract 2 2) ω::T1::cC::fc1) #b0)
...
(= ((_ extract 15 15) ω::T1::cC::fc1) #b0)
```

In a similar fashion, the SMT constraints for the other two frame conditions are generated.

Example 9. *For the operation call* T1.`goThrough` *of our running example, let us now consider the frame condition*

```
modifies only: self.building::inside.
```

Here, the scope is `self.building` *which, in principle, could be any building. Thus, we cannot directly assign all bits as in the previous example. However, the scope can be evaluated similar to navigation expressions in any OCL constraint (i. e., in an invariant, a pre- and/or postconditions). To this end, we use the placeholder* `?self.building?` *to denote the complete SMT constraints for the scope. This will be a bit mask where a* `1` *states that the corresponding object is in the scope and a* `0` *states that the object is not in the scope. Further, assume that the model element* `inside` *of* B1 *has the index 12. Then the constraints are:*

```
(= ((_ extract 0 0) ω::T1::gT::fc1) #b0)
...
(= ((_ extract 11 11) ω::T1::gT::fc1) #b0)
(ite (= ((_ extract 0 0) ?self.building?
         #b1)
     (= ((_ extract 12 12) ω::T1::gT::fc1)
         #b1)
     (= ((_ extract 12 12) ω::T1::gT::fc1)
         #b0)) )
...
```

Let us now come back to the precise handling of associations. An association has two ends and each end can be flagged as variable using frame conditions or not. Since flagging one end of an association as being variable should not only make the flagged association end variable but also the other end of the association. Hence, we have to adjust the formulation of links in the symbolic representation in order to get a more detailed constraint for handling frame conditions on associations.

[7] The precise constraint for associations is a bit more complicated and is given later in more detail, but the main idea is the same one.

[8] The operations `checkCard` and `goThrough` are abbreviated to `cC` and `gT`, respectively.

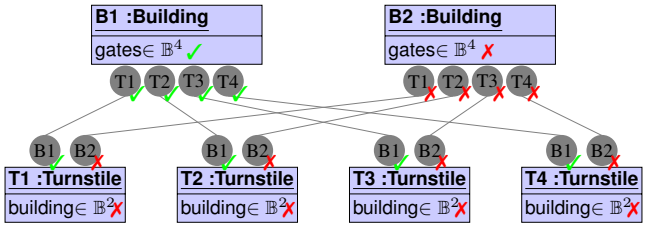

Figure 4: Idea of links in the symbolic representation

In Fig. 4, a partial sketch of the symbolic representation of a system state with two *Building*s and four *Turnstile*s (*MagneticCard*s are omitted) is provided. A satisfying assignment should obtain valid links between the object instances of the two classes. But since no details are known, the symbolic representation must allow all possible links. Thus, for each object instance owning an association end, a relation can be represented by a map from all possible object instances of the class of the other end of the relation to $\mathbb{B}$. For the gates relation of *B1*, where the opposite end is an instance of *Turnstile*, this is the map $\lambda_{B1,gates} : \{T1, T2, T3, T4\} \mapsto \mathbb{B}$. Obviously, links are symmetric and, thus, corresponding constraints must be added for the *Turnstile* objects. They are indicated by gray lines in Fig. 4 between possibly linked object instances (indicated by gray dots with a matching name).

Having this idea in mind, it is easy to understand that the variability of associations cannot work on the single result of the variability map. More precisely, the premise of the implication for every possible link must check the values of both connected instances of the model elements. If at least one of the two values allows for a change of the link status, the link may change. On the other side, if both values are not allowing for a change, the link must be constant.

Formulation 7. *For a model element $\mu \in m(\sigma)$ belonging to an association $r \in \mathcal{R}$, a formulation similar to the one from Eq.* (2) *is used:*

$$\bigwedge_{\substack{\mu \in m(\sigma) \text{ where} \\ \mu \text{ of } r=(c,c',\_,\_) \in \mathcal{R} \\ \text{object of } \mu \text{ is } v \\ v \in \Upsilon(c)}} \bigwedge_{\tilde{v} \in \Upsilon(c')} \left(\begin{array}{l} (\mathcal{V}(\mu) = 0 \wedge \mathcal{V}(r, \tilde{v}) = 0) \\ \quad \Rightarrow \left(\lambda^{\sigma}_{v,r}(\tilde{v}) = \lambda^{\sigma'}_{v,r}(\tilde{v}) \right) \end{array} \right), \tag{3}$$

where $\lambda^{\sigma}_{v,r}(\tilde{v})$ represents the status of the possible link between v and $\tilde{v}$ for the relation r and $\mathcal{V}(r, \tilde{v})$ refers to the opposite instantiated model element. As relations have two ends, Eq. (3) *must also be applied with the changed order of the classes.*

Example 10. *For the running example, the precise SMT constraints for the associations obtained from Eq.* (3) *are:*

```
(=> (and (= ((_ extract 13 13) V) #b0)
         (= ((_ extract 15 15) V) #b0))
    (= ((_ extract 0 0) σ::T1::building)
       ((_ extract 0 0) σ'::T1::building)))
(=> (and (= ((_ extract 14 14) V) #b0)
         (= ((_ extract 15 15) V) #b0))
    (= ((_ extract 0 0) σ::T1::building)
       ((_ extract 0 0) σ'::T1::building)))
(=> (and (= ((_ extract 15 15) V) #b0)
         (= ((_ extract 13 13) V) #b0))
    (= ((_ extract 0 0) σ::B1::gates)
       ((_ extract 0 0) σ'::B1::gates)))
(=> (and (= ((_ extract 15 15) V) #b0)
         (= ((_ extract 14 14) V) #b0))
    (= ((_ extract 1 1) σ::B1::gates)
       ((_ extract 1 1) σ'::B1::gates)))
```

With all those revised and additional constraints, the symbolic representation does now explicitly consider all frame conditions.

V. Closing Remarks

This work provided a concept for integrating frame conditions in symbolic representations to be used in the verification and validation of UML/OCL models. This enables the designer to verify the behavior of UML/OCL models while, at the same time, respecting the given frame conditions. Future work now focuses on the implementation and evaluation of these concepts as well as their extension to the creation and deletion of objects between two system states or to the support of concurrent operation calls [19].

VI. Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SPECifIC under grant no. 01IW13001 and the project SELFIE under grant no. 01IW16001, the German Research Foundation (DFG) within the Reinhart Koselleck project under grant no. DR 287/23-1, the Graduate School SyDe funded by the German Excellence Initiative within the University of Bremen's institutional strategy, Siemens AG, and the Brazilian program Science Without Borders, through the National Council for Scientific and Technological Development (CNPq).

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] Y. Vanderperren, W. Müller, and W. Dehaene, "UML for electronic systems design: a comprehensive overview," *Design Automation for Embedded Systems*, vol. 12, no. 4, pp. 261–292, 2008.

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

[5] M. Kuhlmann and M. Gogolla, "From UML and OCL to Relational Logic and Back," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, 2012, pp. 415–431.

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

[7] F. Hilken, P. Niemann, M. Gogolla, and R. Wille, "Filmstripping and unrolling: A comparison of verification approaches for UML and OCL behavioral models," in *Tests and Proof*, 2014, pp. 99–116.

[8] N. Przigoda, J. Stoppe, J. Seiter, R. Wille, and R. Drechsler, "Verification-driven design across abstraction levels: A case study," in *Conf. on Digital System Design (DSD)*. IEEE, 2015, pp. 375–382.

[9] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," 2010. [Online]. Available: https://www.smt-lib.org

[10] A. Borgida, J. Mylopoulos, and R. Reiter, "On the Frame Problem in Procedure Specifications," *IEEE Trans. Software Eng.*, pp. 785–798, 1995.

[11] M. A. G. de Dios, C. Dania, D. A. Basin, and M. Clavel, "Model-driven development of a secure ehealth application," in *Engineering Secure Future Internet Services and Systems - Current Research*, 2014, pp. 97–118.

[12] J. Cabot, "Ambiguity issues in OCL postconditions," in *OCL Workshop*, 2006, pp. 194–204.

[13] ——, "From Declarative to Imperative UML/OCL Operation Specifications," in *Conceptual Modeling*, 2007, pp. 198–213.

[14] P. Kosiuczenko, "Specification of Invariability in OCL," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, 2006, pp. 676–691.

[15] ——, "Specification of invariability in OCL - Specifying invariable system parts and views," *Software and System Modeling*, vol. 12, no. 2, pp. 415–434, 2013.

[16] A. D. Brucker, F. Tuong, and B. Wolff, "Featherweight OCL: A Proposal for a Machine-Checked Formal Semantics for OCL 2.5," *Archive of Formal Proofs*, 2014.

[17] P. Niemann, F. Hilken, M. Gogolla, and R. Wille, "Assisted Generation of Frame Conditions for Formal Models," in *Design, Automation and Test in Europe*, 2015, pp. 309–312.

[18] ——, "Extracting frame conditions from operation contracts," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, 2015.

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