More than `true` or `false`

Native Support of Irregular Values in the Automatic Validation & Verification of UML/OCL Models

Nils Przigoda[1,2] Philipp Niemann[3] Judith Peters[4]
Frank Hilken[5] Robert Wille[3,6] Rolf Drechsler[2,3]

[1] Siemens AG, MO MM R&D SYS, Braunschweig, Germany, nils.przigoda@siemens.com
[2] Group of Computer Architecture, University of Bremen, Germany, drechsle@informatik.uni-bremen.de
[3] Cyber-Physical Systems, DFKI GmbH, Bremen, Germany, philipp.niemann@dfki.de
[4] Department of Satellite Ground Systems, OHB System AG, Bremen, Germany, judith.peters@ohb.de
[5] Database Systems Group, University of Bremen, Germany, fhilken@informatik.uni-bremen.de
[6] Institute for Integrated Circuits, Johannes Kepler University Linz, Linz, Austria, robert.wille@jku.at

ABSTRACT

UML/OCL models are used to describe system models in early stages of the design process. In order to detect design flaws in these models as soon as possible (ideally before the implementation phase starts), various methods for the validation and verification of UML/OCL models have been proposed. In particular, automatic solutions (so-called model finders) are of interest here. They provide designers with quick feedback, e. g., on the consistency of their models in a push-button fashion. But thus far, all proposed approaches support a (small) subset of UML/OCL only or employ substantial restrictions. In fact, there are only few solutions that support the extended type system including the irregular values `null` and `invalid` – although these values play an important role for covering exceptional cases. Moreover, these solutions either heavily rely on manual interaction or significantly restrict the supported UML/OCL description means. In this work, we propose a generic formal representation of UML/OCL which can be used for the validation and verification of corresponding models and, at the same time, addresses these shortcomings.

The resulting representation can be used by various reasoning engines and, hence, eventually allows for the validation and verification of UML/OCL models with irregular values.

1 INTRODUCTION

During the last decades, system design has become more and more complex as systems themselves became increasingly large. To handle this immense degree of complexity, several abstraction levels were introduced to lead the designer from a specification in natural language to the desired system by refining the design. Modeling languages such as the *Unified Modeling Language* [13, 20] with its additional *Object Constraint Language* [11] (UML/OCL) or the *Systems Modeling Language* [12] (SysML) assist the designer particularly in the early steps of this process. They allow to formally describe the system to be realized while, at the same time, remain on a rather abstract level.

However, even in these early stages, errors can easily occur and corresponding models may result which, e. g., do not describe the system as intended or include contradictory descriptions. As a consequence, researchers and engineers developed several (automatic) methods for the validation and verification of UML/OCL models (often referred to as *model finders* or solutions for *model finding*). This includes approaches such as USE+ASSL [5–7], USE Model Validator [10], EMFtoCSP [9], OCL2MSFOL [4], HOL-OCL [2] based on the theorem prover Isabelle as well as model finding based on satisfiability (SAT/SMT) solvers [15, 22, 23].

At the same time, UML/OCL—as almost all languages—is subject to frequent changes.One of the changes that probably most seriously affects the entire structure of UML/OCL is related to the type system. Originally, UML/OCL supported only one irregular value, namely `null` which extended the data types with similar semantics as null pointers in programming languages like Java. By this, instead of a two-valued Boolean logic, a three-valued logic was implemented. In the meantime, another irregular value, namely `invalid`, was added in order to represent exceptions that may occur when evaluating OCL constraints—making UML/OCL closer to established programming languages such as Java and yielding a four-valued logic in the current UML/OCL version.

While already the first irregular value `null` has not properly been supported by the proposed validation and verification approaches, so far there are only two approaches that can support irregular values, namely HOL/OCL [2] and OCL2MSFOL [4]. However, the first approach requires a high manual effort and expert knowledge from the designer while the second approach, which is indeed a push-button approach, severely restricts the supported modeling constructs.

In this work, we aim for overcoming these shortcomings and propose a solution which provides native support of irregular values in the automatic validation and verification of UML/OCL models.

To this end, we briefly review the overall flow of existing methods, whose main step is the translation of the considered UML/OCL description into a *symbolic formulation*—representing the model as well as the respective verification objective. Based on this as well as a formal interpretation of UML/OCL models and the used type

MEMOCODE '17, September 29-October 2, 2017, Vienna, Austria

ACM ISBN 978-1-4503-5093-8/17/09…$15.00
https://doi.org/10.1145/3127041.3127053

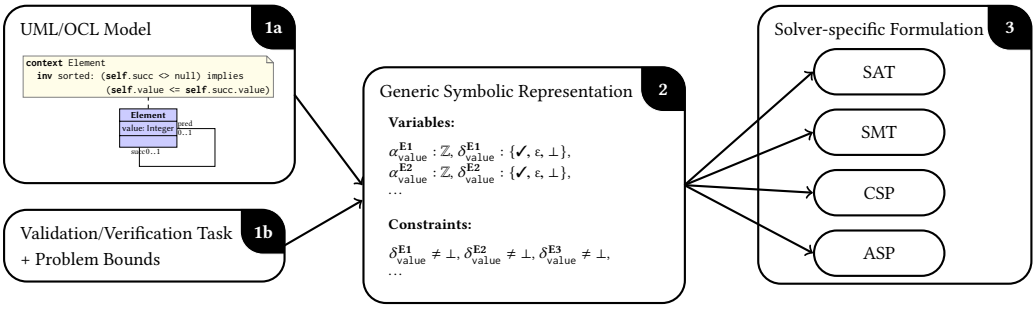


Figure 1: Main flow of automatic methods for the validation and verification of UML/OCL models

system, we present how the translations have to be conducted in order to also support irregular values. These findings are eventually applied to SMT-based model finding which we considered as a representative validation and verification method to exemplarily apply the results of this work. Overall, this provides and demonstrates a solution which supports irregular values and can easily be adapted to many other model finding methods based on reasoning engines.

The remainder of this paper is structured as follows: Section 2 reviews the overall flow of existing methods for automatic validation and verification of UML/OCL models based on reasoning engines. Afterwards, the contribution of this work is provided and demonstrated by means of three sections: Section 3 describes a formal interpretation of UML/OCL models as well as the considered type system. Based on that, a translation from UML/OCL models into a generic formal representation is proposed in Section 4 – providing the main solution how to support irregular values in UML/OCL models for validation and verification methods based on reasoning engines. Section 5 briefly demonstrates the application of the solution for SMT-based model finding. Finally, the work is concluded in Section 6.

2 VALIDATION AND VERIFICATION OF UML/OCL MODELS

In the recent past, several approaches for the automatic validation and verification of UML/OCL models have been proposed (see, e. g., [3, 9, 15, 23]). Although they rely on different reasoning engines such as SAT solvers, SMT solvers, CSP solvers, etc., they all essentially follow the overall flow as sketched in Fig. 1.

More precisely, all approaches take as input a UML model description (class diagram) enriched by OCL constraints together with a verification task which is to be performed on the model. Possible verification tasks comprise, e. g., checking for consistency, deadlocks, the executability of operations, reachability of particular system states, etc. (see, e. g., [15]). Since UML/OCL models allow, in principle, for an infinite number of object instances (which would make the considered verification task undecidable), proper problem bounds are additionally provided which restrict the number of objects per class to some finite value or to be within a given range.

Then, the resulting validation/verification problem is translated into a *symbolic formulation* which consists of variables and constraints over these variables (details are provided later) in an appropriate format for the respectively applied reasoning engine (solver). Although, for this purpose, a direct translation from UML/OCL to a *solver-specific formulation* is possible, we consider a two-stage approach here: First, all instances of the given UML/OCL model (which are possible within the bounds) are symbolically described by means of a *generic symbolic respresentation*. Afterwards, this description is mapped to a solver-specific formulation which, then, can automatically be solved by the respective reasoning engine (see boxes 2 and 3 in Fig. 1). The first translation represents the actual translation, while the second step is basically a syntactical mapping. Following this two-stage scheme allows to describe the symbolic formulation in a generic fashion which, afterwards, can easily be applied to a broad variety of reasoning engines.

Eventually, the resulting formulation is passed to the respective reasoning engine which is supposed to determine an assignment to all variables of the symbolic formulation. If such an assignment exists, it can directly be translated to an actual instance of the UML/OCL model which either serves as witness or counterexample for the particular verification task. If the reasoning engine proves that no such assignment exists, respective conclusions can be drawn (e. g., that the model is inconsistent since it does not allow for an instance satisfying all constraints).

However, as discussed above, all solutions presented thus far and following this overall flow do not support irregular values, but are restricted to a simplified type system without irregular values. Moreover, supporting irregular values is not accomplished by slight modifications of the symbolic formulation, but requires major modifications. In the following, we propose a translation scheme which addresses this issue. To this end, we first provide a solid basis by providing a precise, formal definition of the considered (sub-)set of UML/OCL models in the next section. Based on that, Section 4 provides detailed rules for the translation of UML/OCL models into a generic symbolic formulation. The resulting scheme can afterwards be applied to all methods reviewed above by simply mapping the resulting symbolic formulation into the syntax supported by the reasoning engine. In Section 5 we exemplarily demonstrate that by means of SMT solvers.

3 FORMAL BASIS: TYPE SYSTEM, UML, OCL

In order to apply formal methods for UML/OCL models, a formal definition of them is needed. In this section, we provide this basis as needed for the purposes of this work. This includes a brief definition of the assumed type system (including irregular values), the considered modeling concepts of UML, as well as additional OCL constraints.

3.1 Type System

UML offers several predefined basic data types (such as `Boolean`, `Integer`, `String`) as well as collections (`Set`, `Bag`, `OrderedSet`, `Sequence`) of these types (e. g., `Sequence(Boolean)`). Note that these collections can be nested to an arbitrary depth (e. g., `Bag(Set(Set(Bag(Sequence(Boolean))))`). Besides this, the designer may specify own basic types like *enum-data* types with a finite domain as well as *classes* which are compound structures composed of attributes of arbitrary type and operations (a precise definition of a class is given in Definition 3.5 later in this section).

The OCL type system inherits all these types, but also provides further types not present in the UML, for instance `OclVoid` and `OclInvalid`.[1] As a consequence, besides the *regular*[2] values of a data type (e. g., given by the set $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ for Booleans and by $\mathbb{Z} = \{\ldots, -1, 0, 1, 2, \ldots\}$ for integers), the OCL type system also allows for two additional *irregular* values, namely

- `null` (symbol: ε, from OCL type `OclVoid`) and
- `invalid` (symbol: ⊥, from OCL type `OclInvalid`).

Example 3.1. The ε value has a similar semantic as null pointers in classical programming languages and would, e. g., be returned by the OCL query `OrderedSet{ε}->at(1)` which asks for the first member of an ordered set (which is ε in this case). In contrast, ⊥ is, used to indicate exceptions that may occur when evaluating OCL constraints, e. g., trying to access to the second element of an ordered set with just one element as in `OrderedSet{ε}->at(2)`. This is comparable to an `OutOfBoundsException`, e. g., in Java. Especially, the ⊥ value only arises in this context and is, unlike the ε value, not a valid assignment for class attributes.

More precisely, ε and ⊥ are included in the *universe*, i. e., the set of all possible values of each type t—including classes and collections. In order to consider the corresponding type/universe without ε and/or ⊥, we use the notation $\mathrm{t}_{\not\varepsilon,\not\bot}$ (universe of regular values) and $\mathrm{t}_{\not\varepsilon}$ or $\mathrm{t}_{\not\bot}$, respectively.

Example 3.2. For the type `Boolean`, `true` and `false` are the only regular values in the universe. Together with the two irregular values ε and ⊥, the universe of `Boolean` is complete and we essentially obtain what is commonly called a four-valued logic in the modeling community—even if there is no named counterpart in literature as, e. g., the Belnap logic.

Special care has to be taken for collections: the collection `Collection(t)` may not contain the element ⊥ (invalid), even though ⊥ is an element of the (complete) universe of the type t. However, ⊥ as well as ε themselves are considered valid collections of any type—and are different from the empty collection.

Example 3.3. The complete universe of `Set(Boolean)` is given by: `Set{}` (empty set), `Set{true}`, `Set{false}`, `Set{true, false}`, `Set{ε}`, `Set{true, ε}`, `Set{false, ε}`, `Set{true, false, ε}`, ε, and ⊥.

In this paper, we do not consider the full OCL type system, but restrict to the following subset $\mathfrak{T}$ that is sufficient for UML/OCL class diagrams:

- all primitive types (`Boolean`, `Integer`, `Real`, `UnlimitedNatural`, `String`) are contained in $\mathfrak{T}$,
- all enum-data and class types are contained in $\mathfrak{T}$, and
- with $\mathrm{t} \in \mathfrak{T}$ also $\mathrm{t}_{\not\varepsilon}$, $\mathrm{t}_{\not\bot}$, $\mathrm{t}_{\not\varepsilon,\not\bot}$ and $\mathit{Collection}(\mathrm{t}_{\not\bot})$ are contained in $\mathfrak{T}$ where $\mathit{Collection} \in \{\texttt{Set}, \texttt{Bag}, \texttt{OrderedSet}, \texttt{Sequence}\}$.

As class attributes may not assume the value ⊥, we additionally consider the derived type system $\mathfrak{T}_{\not\bot}$ which contains all types from $\mathfrak{T}$ whose universe does not contain ⊥.

On top of these type systems, variables can be defined: A variable is a tuple (v, t) consisting of an identifier/name v of type $\texttt{String}_{\emptyset,\not\varepsilon,\not\bot}$, i. e., neither empty nor ε nor ⊥, and a type t and is usually denoted as $v\colon \mathrm{t}$. It can be seen as an actual instance of a type t which represents a precise assignment of any value from the (complete) universe of t to v.

Example 3.4. p : `Integer` defines the variable p of the type `Integer` and with $p \leftarrow 17$ an explicit assignment is given.

The set of all variables of a type $\mathrm{t} \in \mathfrak{T}$ is denoted by $\mathcal{V}_{\mathrm{t}}$. Moreover, the short-hand $\mathcal{V}_{\mathfrak{T}} = \bigcup_{\mathrm{t}\in\mathfrak{T}} \mathcal{V}_{\mathrm{t}}$ is used to denote the set of all variables whose type is in $\mathfrak{T}$ (likewise for $\mathfrak{T}_{\not\bot}$). Having the notion of variables, we can now precisely define class types:

Definition 3.5 (Class). A *class* $c = (n\colon \texttt{String}_{\emptyset,\not\varepsilon,\not\bot}, A, O)$ is a 3-tuple composed of a name n and finite sets of *attributes* $A \subset \mathcal{V}_{\mathfrak{T}_{\not\bot}}$ (i. e., attributes may never be assigned the irregular value ⊥) as well as *operations* O.[3] The identifiers of the attributes are unique which means that for $a_1 = (v_1\colon \mathrm{t}_1)$, $a_2 = (v_2\colon \mathrm{t}_2) \in A$ we have $(v_1 = v_2) \Rightarrow (a_1 = a_2)$.

The universe of a class type is given by the set of corresponding objects which can be defined as follows.

Definition 3.6 (Class Instance/Object). An instance of a class $c = (n\colon \texttt{String}_{\emptyset,\not\varepsilon,\not\bot}, A, O)$ is given by a precise assignment of values to n (object name) as well as to all attributes of the class c. In the following, it will be called *object* or *object instance*. The universe of objects of a class c is written as Υ_c.

Remark. *For the sake of brevity, we omit some minor aspects like inheritance between classes as well as tuple types. These do not have a significant impact on the resulting formulations and it is straightforward to extend the presented concepts accordingly in order to cover them explicitly.*

Based on the type system given so far, now the modeling structures can be defined.

3.2 UML Class Diagrams

This subsection provides the formalization of UML class diagrams using the previously defined type system and variables. Besides classes, the main component of UML class diagrams are associations which denote relations between the classes.

Definition 3.7 (Association). An n-ary *association* r (also called *relation*) is formally defined as an n-tuple of pairs $r = \big((role_1\colon c_1, range_1), \ldots, (role_n\colon c_n, range_n)\big)$ where the c_i are

[1] For more details about on the full OCL type system, interested readers are referred to the OCL specification [11, p. 211ff.].

[2] We are using the terms regular as well as irregular values for differentiating purposes. They are not used within the UML/OCL standards.

[3] Operations and behavioral aspects of UML/OCL models are not explicitly covered here due to page limitations, but the concepts presented in the following can easily be extended accordingly.

class types from $\mathfrak{T}$ (not necessarily different) and the $range_i$ are unions of finitely many intervals from $\mathbb{N} \cup \{\infty\}$. The individual elements of r are called *association ends* and the identifiers $role_i$ *role names*. As it is possible in OCL to navigate to another association end using its role name, the i-th association end $(role_i : c_i, range_i)$ is said to be *navigable* from all classes c_j, $1 \leq i, j \leq n$, $i \neq j$. The $range_i$ represent multiplicities between the classes.

For the sake of simplicity, the work at hand is rest*/9ricted to binary associations and single intervals of multiplicities $[a, a+1, \ldots, b]$ only (where $0 \leq a \leq b \leq \infty$). However, this does not restrict the expressiveness as shown in [8].

Instantiations of associations are called links and are defined as follows:

Definition 3.8 (Links). Let $r = \big((role_1 : c_1, range_1), (role_2 : c_2, range_2)\big)$ be a binary association. Then, an instance of r is a so-called *link* between two object instances $v_{c_1} \in \Upsilon_{c_1}$ and $v_{c_2} \in \Upsilon_{c_2}$ (derived from the classes c_1 and c_2, respectively). More precisely, it is a tuple $\lambda = (v_{c_1}, v_{c_2}, role_1, role_2)$ from the Cartesian product $\Upsilon_{c_1} \times \Upsilon_{c_2} \times \{role_1\} \times \{role_2\}$. In case that $c_1 = c_2$, the respective roles are implicitly given by the ordering of the association ends in r.

Having this notation, we are able to provide a formal description of UML class diagrams (simply denoted by *model* in the following):

Definition 3.9 (Model). Let $m = (C, R)$ be a tuple of finitely many *classes* C and *relations* R between classes from C. For a class $c \in C$, let A_c denote its set of attributes and $Roles_c$ denote the set of all navigable association ends of c in R. Then, m is called a *model* if, and only if, for each class $c \in C$

- there is no class $c' \in C$, $c \neq c'$ with the same name (class names are unique),
- there are no two different navigable association ends $e_1 = (role_1 : c_1, range_1), e_2 = (role_2 : c_2, range_2)$ from $Roles_c$ with the same identifier, i. e.,
$$(role_1 = role_2) \Rightarrow (e_1 = e_2),$$
and each navigable association end can be associated with a unique relation $r \in R$, i. e., for $r, r' \in R$ we have
$$\begin{pmatrix} & e_1 \text{ is an association end of } r \\ \wedge & e_2 \text{ is an association end of } r' \end{pmatrix} \Rightarrow (r = r')$$
(navigable role names are unique),
- for each $e = (role : c', range) \in Roles_c$ there is no attribute in A_c with the same identifier $role$ (meaning that attribute identifiers and navigable role names are disjoint).

Example 3.10. Figure 2 illustrates a simple `Integer` typed linked list model. It consists of one class `Element` with an attribute to save the entry value and an association to connect elements with each other. Additionally, an invariant `sorted` makes sure that the values of the elements of such list are in ascending order.

Now, we consider instances of a given model (called system state) which consists of instances of classes as well as relations (also called objects and links, respectively).

Definition 3.11 (System State). Let $m = (C, R)$ be a given model. Then, $\sigma = (\Upsilon, \Lambda)$ is called a *system state* of m, if

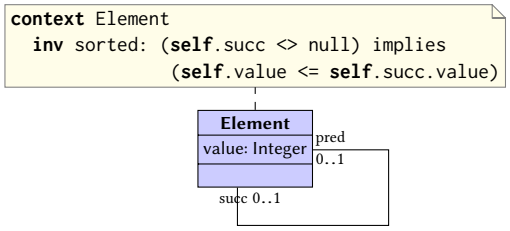
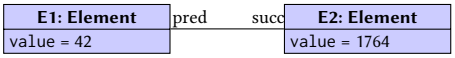
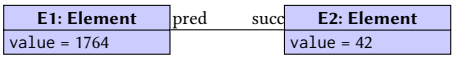


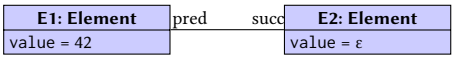
Figure 2: A simple list model



(a) A system state where the invariant `sorted` evaluates to `true`



(b) A system state where the invariant `sorted` evaluates to `false`



(c) A system state where the invariant `sorted` evaluates to $\perp$

Figure 3: System states for the linked list model

- both elements, Υ and Λ, are finite sets and each object from Υ (link from Λ) is an instance of a class from C (association from R),
- the object names for the objects in Υ are unique, and
- all objects used anywhere in any attribute of any object from Υ or in any link from Λ are contained in Υ.

The set of all possible system states for m is denoted by Σ_m.

Note that the above definition does not enforce the multiplicity constraints given by an association. However, in order to be a valid system state according to the model description, these constraints need to be satisfied. This is expressed by the following definition.

Definition 3.12 (Validity of an Association in a System State). Let $\sigma = (\Upsilon, \Lambda)$ be a system state of a model $m = (C, R)$. Then, the relations in R are *valid* (or *satisfied*) in the system state σ if, and only if,

$$\begin{aligned} &\forall\, r = \big((role_1 : c_1, range_1), (role_2 : c_2, range_2)\big) \in R : \\ &\quad \big(\forall\, v \in \Upsilon_{c_1} : \big|\{(v, v', role_1, role_2) \in \Lambda\}\big| \in range_1\big) \\ &\quad \wedge\ \big(\forall\, v \in \Upsilon_{c_2} : \big|\{(v', v, role_1, role_2) \in \Lambda\}\big| \in range_2\big) \end{aligned}$$

Example 3.13. Figure 3 shows three example system states of the linked list model, all of which satisfy all associations of the model as each element has at most one predecessor and successor.

3.3 Class Diagrams with OCL Constraints

So far, we have considered pure UML class diagrams without textual OCL constraints. In the following, we discuss how OCL constraints can additionally be taken into account.

The *Object Constraint Language* (OCL) is a declarative language which mainly consists of

- navigation expressions to access attributes and association ends of a particular object (`self`) or related objects that can be reached using navigable association ends,
- arithmetic operations (i. e., addition, subtraction, multiplication, division etc.),
- collection operations (i. e., intersection, union, element containment, etc.), and
- logic operations (i. e., conjunction, disjunction, negation, etc.) as well as quantifiers (universal and existential).[4]

For the remainder of this work, it is sufficient to know that OCL constraints can be annotated to classes (in terms of so-called *invariants*) in order to express constraints that shall be satisfied by any object instance of that class.[5] In order to refer to the particular object on which an OCL expression is evaluated, the keyword `self` is employed.

Example 3.14. The running example from Fig. 2 has one invariant `sorted` which can evaluate to true, false, or $\perp$ (but not ε). More precisely, in the system state shown in Fig. 3(a) the invariant evaluates to true (if evaluated on E1 as well as E2) as all elements are sorted. In contrast, in the system state shown in Fig. 3(b), the invariant is violated due to an incorrect ordering and, thus, evaluates to false (on E1). Finally, in the system state shown in Fig. 3(c), one value is undefined which leads to an error in the comparison of the values (indicated by $\perp$). This error is propagated upwards in the invariant expression and, finally, the whole invariant evaluates to $\perp$. Overall, the invariant is satisfied if, and only if, all values of all linked elements are defined (not equal to ε) and in ascending order.

The above description leads to the following definition of UML/OCL models:

Definition 3.15 (UML/OCL model). A 3-tuple $m = (C, R, I)$ is called *UML/OCL model* if, and only if, (C, R) is a model and $I = \coprod_{c \in C}(I_c)$ is the disjoint union of sets of invariants I_c for each class $c \in C$. All these sets are finite, possibly empty sets of OCL expressions.

System states of UML/OCL models are simply the system states of the underlying UML models. However, the notion of validity is extended as follows:

Definition 3.16 (Valid System State). Let $m = (C, R, I)$ be a UML/OCL model and $\sigma = (\Upsilon, \Lambda)$ be a system state of the model (C, R). Moreover, let I_c denote the set of invariants from I that are associated with a class $c \in C$.

Then, σ is called a *valid system state* if, and only if,

- the relations in R are satisfied in σ (cf. Definition 3.12) and
- for each class $c \in C$ and each object v of this class that is instantiated in σ (i.e. $v \in \Upsilon_c \cap \Upsilon$), all invariants from I_c evaluate to true when `self` is referring to v.

The set of all valid system states of m is denoted by $\Sigma_m^{\checkmark}$.

[4] A comprehensive overview on all OCL expressions and as keywords as well as a precise semantic definition can be obtained from [11].

[5] In general, OCL constraints can also be annotated to (class) operations in terms of so-called pre- and postconditions, but the translation of the corresponding expressions is essentially identical to invariants.

Example 3.17. Among the system states in Fig. 3 only the first one is a valid system state of the list model as all relations are satisfied and all invariants evaluate to true on any of the elements.

4 GENERIC REPRESENTATION

Given the formal interpretation of UML/OCL models from the previous section, we are now able to describe the transformation rules to obtain a generic representation of the model (box 2 of Fig. 1). This generic symbolic representation represents the following set of system states:

Definition 4.1 (Unassigned system state). Let $m = (C, R, I)$ be a UML/OCL model. For each class $c \in C$, let $PB_c \subset \mathbb{N}$, $|PB_c| < \infty$ denote the allowed range of possible instantiations of c. Then, the set

$$S = \{(\Upsilon, \Lambda) \in \Sigma_m^{\checkmark} \mid \forall\, c \in C : |\Upsilon_c \cap \Upsilon| \in B_c\}$$

is called *unassigned system state* of m with respect to the *problem bounds* $PB = \{PB_c \mid c \in C\}$.

In other words, an unassigned system state represents all valid system states of a model for which the number of instantiated objects of each class is within a given range (e. g., an interval or a fixed number). To this end, the generic symbolic representation of S consists of

a) *unassigned* variables for each possibly instantiated object (for its name, attributes as well as navigable association ends) that may be assigned any value from the corresponding universe, and
b) constraints that enforce the validity of the system state, i. e., ensure that all multiplicity constraints of the relations as well as all OCL invariants hold.

Note that any valid assignment to these variables determines a unique system state $\sigma = (\Upsilon_\sigma, \Lambda_\sigma) \in S$ for which we do not exactly know in advance how many and which objects of a class $c \in C$ are instantiated in σ. Nonetheless, we need to refer to the objects from Υ_σ within the symbolic formulation, e. g., to formulate that class type attributes and association ends may only refer to objects that are actually instantiated. To this end, we consider a generic set Υ_σ that contains $\max PB_c$ *possibly instantiated* objects for each $c \in C$ and defines the respectively applied problem bounds.

In the following, we provide precise transformation rules for generating the symbolic representation from a given UML/OCL model that comprehensively takes into account irregular values. First, the concept of how to distinguish between regular and irregular values is outlined in Section 4.1. Afterwards, the transformation of attributes as basic elements is introduced in Section 4.2—followed by the transformation of associations (including their multiplicity constraints) in Section 4.3. Finally, the translation of OCL constraints is described in Section 4.4.

4.1 Distinction Between Regular and Irregular Values

In Section 3.1, the considered type system has been introduced. As mentioned there, the universes of possible values for all types can be separated in two groups: regular values and irregular values. In order to represent this, for each transformed variable $v \colon \mathrm{t}$ in m, we create a pair of variables: $(\alpha^v \colon \mathrm{t}_{\not\perp}, \delta^v \colon \{\checkmark, \varepsilon, \perp\})$. The α^v-part

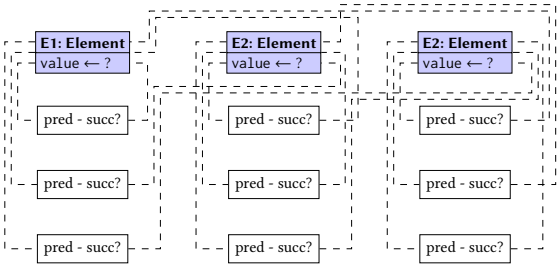


Figure 4: An un-assigned system state

represents regular values of v and the δ-part the *definedness* status of the pair whereby ✓ means that the value of v is neither `null` nor `invalid`, but a *regular* value.

Note that, in order to reduce the search space, it can be helpful to define a regular *default* value 0_{t} for each type $\mathsf{t} \in \mathfrak{T}$ and to enforce this as the value for α^{v} in case that $\delta^{v} \neq ✓$.

4.2 Transforming Attributes

To symbolically represent all possible attribute assignments of the unassigned system state S for each attribute of each possibly instantiated object, a corresponding variable is created.

Transformation Rule 4.2 (Attributes). *For each $c = (n, A, O)$ in C, each attribute $(v_a : \mathsf{t}_{\not\perp}) \in A$, and for each possibly instantiated object $v \in (\Upsilon_c \cap \Upsilon_\sigma)$, a pair of variables $(\alpha_v^{v_a} : \mathsf{t}_{\not\perp,\perp}, \delta_v^{v_a} : \{✓, \varepsilon, \perp\})$ is created. Additionally, it is enforced that $\delta_v^{v_a} \neq \perp$. Moreover,*

- *for class types, i. e., $\mathsf{t} = c'$, a constraint is added stating that only possibly instantiated objects or ε may be assigned to v_a, i. e., $(v_a \leftarrow x) \Rightarrow (x \in \Upsilon_\sigma \vee x = \varepsilon)$.*
- *if $\mathsf{t} = \mathtt{Set}(\mathsf{t}'_{\not\perp})$, a variable $\alpha_v^{v_a} : \mathsf{t}'_{\not\perp} \to \mathbb{B}$ is created, where $\alpha_v^{v_a}(x)$ denotes for the element $x \in \mathsf{t}'_{\not\perp}$ whether it is contained in the set or not.*
- *if $\mathsf{t} = \mathtt{Bag}(\mathsf{t}'_{\not\perp})$ or $\mathsf{t} = \mathtt{OrderedSet}(\mathsf{t}'_{\not\perp})$, a variable $\alpha_v^{v_a} : \mathsf{t}'_{\not\perp} \to \mathbb{N}$ is created, where $\alpha_v^{v_a}(x)$ denotes how often an element $x \in \mathsf{t}'_{\not\perp}$ is contained in the bag or the index of x in the ordered set, respectively. In both cases, $\alpha_v^{v_a}(x) = 0$ indicates that x is not contained in the bag/ordered set.*
- *if $\mathsf{t} = \mathtt{Sequence}(\mathsf{t}'_{\not\perp})$, a variable $\alpha_v^{v_a} : \mathsf{t}'_{\not\perp} \to \mathcal{P}(\mathbb{N} \setminus \{0\}) \cup \{\{0\}\}$ is created, where $\alpha_v^{v_a}(x)$ denotes at which indices/positions in the sequence the element $x \in \mathsf{t}'_{\not\perp}$ appears and $\alpha_v^{v_a}(x) = \{0\}$ means that the element is not contained at all.*

Furthermore, for an `OrderedSet` as well as a `Sequence`, constraints are added such that the UML/OCL collection type definition is respected. One constraint ensures that there is at most one object at position $i \in \mathbb{N}_{\geq 1}$ and another one that if there is an object at position i (with $i > 1$), then there is also an object at position $i - 1$.

Example 4.3. Let's assume the model m is the simple list model as depicted in Fig. 2 (box 1a in Fig. 1). Further, let's assume the problem bound $PB_{Element} = \{3\}$ (box 1b in Fig. 1) yielding to the un-assigned system state (box 2) sketched in Fig. 4.

In a next step, the currently introduced transformation rules for attributes are applied to derive variables which are needed to represent all possible system states from the un-assigned system state in a symbolic fashion. Figure 5 shows a list of all variables created for the attributes in Fig. 4.

Variables:
$\alpha^{\mathbf{E1}}_{\mathsf{value}} : \mathbb{Z}$, $\delta^{\mathbf{E1}}_{\mathsf{value}} : \{✓, \varepsilon, \perp\}$, $\alpha^{\mathbf{E2}}_{\mathsf{value}} : \mathbb{Z}$, $\delta^{\mathbf{E2}}_{\mathsf{value}} : \{✓, \varepsilon, \perp\}$,
$\alpha^{\mathbf{E3}}_{\mathsf{value}} : \mathbb{Z}$, $\delta^{\mathbf{E3}}_{\mathsf{value}} : \{✓, \varepsilon, \perp\}$

Constraints:
$\delta^{\mathbf{E1}}_{\mathsf{value}} \neq \perp$, $\delta^{\mathbf{E2}}_{\mathsf{value}} \neq \perp$, $\delta^{\mathbf{E3}}_{\mathsf{value}} \neq \perp$

Figure 5: List of variables and constraints for the attributes

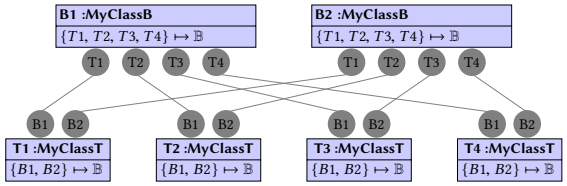


Figure 6: Idea of links in the symbolic representation

4.3 Transforming Associations

To symbolically represent all possible links within a system state $\sigma = (\Upsilon, \Lambda)$, variables are created for each association and the respective object instances of each association end. Before giving the formal transformation rule, the general idea is explained.

Let $(role_{c_1} : c_1, range_1)$ and $(role_{c_2} : c_2, range_2)$ be the two association ends of an association and let us ignore the multiplicities for a moment. Then, each object instance of c_1 can be linked with any arbitrary object instance of c_2. Thus, a function mapping each object instance of c_2 to one of the Boolean values (`true` means linked and `false` not linked) works as a representation for all possible combinations of links. Adding variables for those functions for all objects of c_1 to represent all possible links would already be sufficient. However, for the sake of convenience it is also done for the objects of class c_2. More precisely, for each object of c_2, a function mapping the object instances of c_1 to $\mathbb{B}$ is also added. Furthermore, constraints ensuring the symmetry of associations and the multiplicity constraints must be added.

Example 4.4. In Figure 6, a piece of the symbolic formulation for links between two object instances of a class *MyClassB* and four of a class *MyClassT* is sketched. Since a general symbolic formulation should represent all links between the object instances of the two classes, the symbolic representation must allow all possible links. Thus, applying the ideas explained before, one function mapping the four objects of *MyClassT* $\Upsilon_{MyClassT} = \{T1, T2, T3, T4\}$ to $\mathbb{B}$ is added to the instance *B1* and another to *B2*. Similarly, functions from $\{B1, B2\}$ to $\mathbb{B}$ are added to all four instances of *MyClassT*. The symmetry constraints are drawn with gray lines in Fig. 6 between possibly linked object instances (indicated by gray dots with a matching name), they represent the equality of two evaluations.

Transformation Rule 4.5 (Associations). *Let $r = \big((role_1 : c_1, range_1), (role_2 : c_2, range_2)\big) \in R$ be an association, consider the association end $(role_i : c_i, range_i)$ and let c_j denote the type of the other association end. Then, we essentially treat $role_i : c_i$ as if it was a*

collection typed attribute $role_i : \mathtt{Set}(c_i)_\perp$ *of* c_j*. That is, for each possibly instantiated object* v_j *of class* c_j*, we add a pair of variables* $(\lambda_{v_j}^{role_i} : c_{i,\perp} \to \mathbb{B}, \delta_{v_j}^{role_i} : \{\checkmark, \varepsilon, \perp\})$ *and, in order to respect the multiplicity constraints, we enforce*

$$(\delta_{v_j}^{role_i} = \checkmark) \Rightarrow \sum_{v' \in \Upsilon_{c_i}} \lambda_{v_j}^{role_i}(v') \in range_i, \tag{1}$$

where we identify false with 0 and true with 1 in order to construct the given sum. Then, there are two cases:[6]

- *If* $\max(range_i) > 1$*, the association end may not be* ε*, i. e., we enforce* $\delta_{v_j}^{role_i} \neq \varepsilon$*.*
- *If* $\max(range_i) = 1$*, the association end can be equal to* ε*. More precisely, this is the case if, and only if,* $\min(range_i) = 0$ *and there is no corresponding link for* v_j*. However, as the multiplicity constraints would also permit the empty set in that particular case, we additionally enforce*

$$(\delta_{v_j}^{role_i} = \checkmark) \Rightarrow (\lambda_{v_j}^{role_i} \neq \emptyset).$$

Finally, since a link is symmetric, the following constraints must be added:

$$\forall\, v \in \Upsilon_{c_1} : \forall\, v' \in \Upsilon_{c_2} : \lambda_{role_{c_2}}^{v}(v') = \lambda_{role_{c_1}}^{v'}(v) \tag{2}$$

4.4 Transforming OCL Constraints

After introducing all the different variables, the symbolic formulation represents all possible system states with respect to the given problem bounds—including invalid ones. The reason for this is that all previously introduced constraints only ensure that an assignment respects the defined variable types and that all associations are valid (all multiplicity constraints hold). However, the validity of the OCL constraints (invariants) have not been taken into account so far.

Consequently, the OCL constraints have to be transformed into equivalent logic constraints over the variables from the symbolic formulation. If these constraints are then added to the formulation, any variable assignment finally determined by a reasoning engine will represent a valid system state that satisfies all OCL constraints.

The main idea to this transformation is to consider the *Abstract Syntax Tree* (AST) of the respective OCL expressions and to transform the AST node by node from bottom to top using a *depth-first search* (DFS) and to add a corresponding pair of variables $(\rho_{exp} : \mathrm{t}, \delta_{exp} : \{\checkmark, \varepsilon, \perp\})$ for each node.

Example 4.6. Figure 7 shows the AST of the invariant `sorted` of the list model. The root is an `OperationCallExp`. In OCL, `OperationCall` expressions have an attribute providing the operation name (here: implies). Each operation is called on a source object, also called *calling object*, and has a list of parameters, also called arguments. For binary comparisons and logical operations, the calling object is on the left-hand side and the object given by the (single) argument is on the right-hand side. In this example, the implies operation is called on the result of an `OperationCallExp` (<>) which itself is called on the result of a `PropertyCallExp` using the role name `succ` and is compared with the argument ε. This

[6] The differentiation for max > 1 and max = 1 is caused by a technical detail in the UML and OCL standards which is spread at several positions. It basically says that there is a difference between `Set` and ε for this two cases.

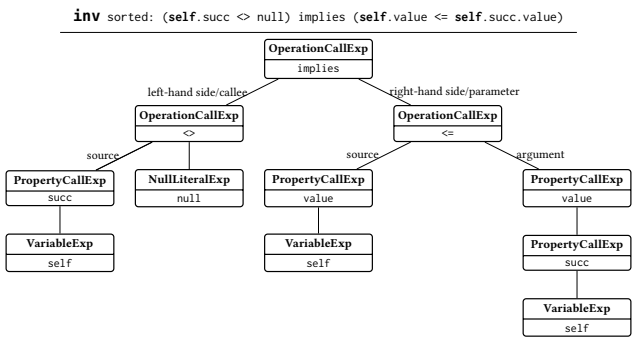


Figure 7: The AST of the invariant `sorted`

means that the left-hand side of the implication is true if, and only if, the current element (`self`) has a link to a successor. Similarly, the right-hand side compares the element's with the successor's value.

For invariants, the OCL standard requires that the type of ρ_{root} of the root node is Boolean, such that we finally enforce $\rho_{root} = \mathtt{true}$ and $\delta_{root} = \checkmark$. Note that this can easily be adapted accordingly in case that one is interested in invalid system states that show exceptional cases (indicated by $\perp$) rather than valid system states only.

While variable and property call (i. e., navigation) expressions are only used to determine the corresponding variables (ρ, δ) within the symbolic formulation, operation calls (e. g., realizing logic operations or comparisons) are the most important expression type for the support of irregular values and the precise transformation rules for these expressions will be outlined in the remainder of this section.

Remark. *The following rules are based on our interpretation of the OCL specification which we gained by a thorough study of [11]. However, we found that some corner cases are not properly covered there and, thus, had to derive our own interpretation. Nevertheless, the following transformation rules provide a general basis, while the precise definition for a specific operation in the OCL standard might change or might be clarified in the future. In this case, the obtained transformation rules can easily be adapted accordingly.*

Transformation Rule 4.7 (`OperationCallExp` – on Boolean expressions). *OCL offers several binary logic operations on Boolean expressions like* `and`, `or`, `xor`, `implies`, `=`, *and* `<>` *which evaluate to a new Boolean expression representing the result of the respective operation. However, all those operations must deal with the four-valued OCL logic including the irregular values* ε *and* $\perp$*. The OCL specification describes those issues in [11, Sect. 7.4.13 on pp. 16f., Sect. 11.3.2 on pp. 154f., Sect. 11.3.3. on pp. 155f., and Sect. 11.5.4 on p. 162]. Only a short part of the definitions from these sections is considered here to give an idea how the corresponding transformation rules have been derived, namely:*

- *True OR-ed with anything is True*
- *False AND-ed with anything is False*
- *False IMPLIES anything is True*
- *anything IMPLIES True is True*

Table 1: Truth table for logic operations

a	b	a or b	a and b	a implies b	$a = b$
false	false	false	false	true	true
false	true	true	false	true	false
false	ε	ε	false	true	false
false	⊥	⊥	false	true	⊥
true	false	true	false	false	false
true	true	true	true	true	true
true	ε	true	ε	ε	false
true	⊥	true	⊥	⊥	⊥
ε	false	ε	false	ε	false
ε	true	true	ε	true	false
ε	ε	ε	ε	ε	true
ε	⊥	⊥	⊥	⊥	⊥
⊥	false	⊥	false	⊥	⊥
⊥	true	true	⊥	true	⊥
⊥	ε	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥

With "anything" one of the four-logic values is meant, i. e., integers or other completely different types are not allowed and should be recognized by the chosen OCL parser beforehand.

For the transformation of and, or, =, *and* implies, *the truth table in Table 1 is used. While the first three columns have been derived from [11, Table A.2 on p. 214], the last one has been derived from several positions of the OCL specification.*

Here, we only provide the transformation for implies, *since the transformations for* or, and, *and* = *are very similar and do not provide much additional insight:*

implies Transformation
Input: $(\rho_{lhs}, \delta_{lhs})$ *and* $(\rho_{rhs}, \delta_{rhs})$
1: ***if*** $(\delta_{lhs} = \checkmark \wedge \rho_{lhs} = \texttt{false}) \vee (\rho_{rhs} = \texttt{true})$ ***then***
2: $(\texttt{true}, \checkmark)$
3: ***else***
4: ***if*** $\delta_{lhs} = \bot \vee \delta_{rhs} = \bot$ ***then***
5: $(\texttt{false}, \bot)$
6: ***else***
7: ***if*** $\delta_{lhs} = \varepsilon \vee \delta_{rhs} = \varepsilon$ ***then***
8: $(\texttt{false}, \varepsilon)$
9: ***else***
10: $(\texttt{false}, \checkmark)$

Other operators like arithmetic expressions, operations on collections etc. have also been covered. However, in the following we will only deal with comparisons of integers. All remaining operation are omitted due to limited space.

Transformation Rule 4.8 (OperationCallExp – comparison of integers). *While the comparison of two regular integer expressions results in a regular Boolean expression where the comparison is the naturally given one, it is not clear how to deal with cases where at least one of the two integer expressions is not regular at a first glance. However, with the following citations from the OCL specification [11, Sect. 11.2.3 and Sect. 11.2.4 on p. 152] it is clear that:*

- *Any operation call applied on* ε *is* ⊥
- *Any operation call applied on* ⊥ *is* ⊥

For the operations = and <> exceptions of interest here are formulated as follows:

- *= applied on* ε *is true*
- *= applied on* ⊥ *is* ⊥
- *<> applied on* ε *is false*
- *<> applied on* ⊥ *is* ⊥

Having this textual specification, the transformation rules are straightforward:

= with two integers Transformation
Input: $(\rho_{lhs}, \delta_{lhs})$ *and* $(\rho_{rhs}, \delta_{rhs})$
1: ***if*** $\delta_{lhs} = \bot \vee \delta_{rhs} = \bot$ ***then***
2: $(\texttt{false}, \bot)$
3: ***else***
4: ***if*** $\delta_{lhs} = \varepsilon \wedge \delta_{rhs} = \varepsilon$ ***then***
5: $(\texttt{true}, \checkmark)$
6: ***else***
7: ***if*** $\delta_{lhs} = \checkmark \wedge \delta_{rhs} = \checkmark$ ***then***
8: $(\rho_{lhs} = \rho_{rhs}, \checkmark)$
9: ***else***
10: $(\texttt{false}, \checkmark)$

The transformation rules for <, <=, >, and >= can be summarized. In the last line of the transformation OP obviously has to be replaced by its natural counterpart:

<, <=, >, => with two integer Transformation
Input: $(\rho_{lhs}, \delta_{lhs})$ *and* $(\rho_{rhs}, \delta_{rhs})$
1: ***if*** $\delta_{lhs} = \bot \vee \delta_{rhs} = \bot \vee \delta_{lhs} = \varepsilon \vee \delta_{rhs} = \varepsilon$ ***then***
2: $(\texttt{false}, \bot)$
3: ***else***
4: $(\rho_{lhs}\ OP\ \rho_{rhs}, \checkmark)$

5 APPLICATION FOR SMT-BASED MODEL FINDING

In this section, we demonstrate how the generic representation obtained using the transformation proposed in the previous section can be utilized for a solver-specific formulation. To this end, we consider SMT-based model finding (proposed in [23]) as a representative. Here, the respective representation has to be mapped to a corresponding SMT-LIB syntax [1]—in this case, to a description in the theory of quantifier-free bit vectors (QF_BV).

Variables for attributes are mapped as follows:

SMT-LIB Realization of Transformation Rule 4.2.
The variables for Boolean attributes $\alpha_v^{v_a} : \mathbb{B}$ *are mapped to the SMT-LIB type* **Bool**. *They are formally denoted by* $\alpha_v^{v_a} : \mathbb{B}^1$ *and the corresponding SMT-LIB declaration is*

```
(declare-fun α_v^{v_a} () Bool)
```

The variables for integer attributes $\alpha_v^{v_a} : \mathbb{Z}$ *are mapped to bit vector variables of a fixed length* l, *i. e., formally* $\alpha_v^{v_a} : \mathbb{B}^l$; *the SMT-LIB declaration is*

```
(declare-fun α_v^{v_a} () (_ BitVec l))
```

The precision l *can be chosen arbitrarily by the designer (but has to be the same for all integer attributes).*

Collections are mapped in a similar fashion. A detailed description of this has been presented in [21]. Here, we only show the realization of associations: The variables $\lambda_v^{role_2} : c_{2,\not\perp} \to \mathbb{B}$ *and* $\lambda_{v'}^{role_1} : c_{1,\not\perp} \to \mathbb{B}$ *created for the general symbolic formulation of an association* $r = \{(role_i : c_i, range_i) \mid i = 1, 2\}$ *and corresponding objects* v, v' *both represent maps that return either* `true` *or* `false`.

Hence, a corresponding bit vector of length $\max PB_{c_2}$ *or* $\max PB_{c_1}$, *respectively, is created.*

To map the symmetry constraints from Eq. (2), *an ordering is applied on the possibly instantiated objects, i. e.,* $\Upsilon_{c_1} = \{v_1, v_2, \ldots, v_{\max PB_{c_1}}\}$ *and* $\Upsilon_{c_2} = \{v_1', v_2', \ldots, v_{\max PB_{c_1}}'\}$, *and the evaluation of the functions* $\lambda_v^{role_2}$ *and* $\lambda_{v'}^{role_1}$ *are realized by extracting individual bits from the bit vectors:*

```
(and (= ((_ extract 0 0) λ^role_c2_v1)
        ((_ extract 0 0) λ^role_c1_v1'))
     (= ((_ extract 1 1) λ^role_c2_v1)
        ((_ extract 0 0) λ^role_c1_v2'))
     ...
```

The multiplicity constraints from Eq. (1) *are mapped to so-called* cardinality constraints *as also used, e. g., in [19].*

The δ*-variables that represent definedness are mapped to bit vector variables of length 2 where one of the four possible values is blocked. More precisely, we identify the bit-vectors* `#b00`, `#b01`, *and* `#b10` *with* ✓, ε, *and* ⊥, *respectively. Moreover, to reduce the search space, a default value (here: a zero string of adequate length) is enforced for the corresponding* α*-variable in case that* $\delta = \varepsilon$*:*

```
; declaration
(declare-fun δ_a^v () (_ BitVec 2))
; general blocking of 4th value
(not (= δ_a^v #b11))
; blocking ⊥ (for attributes only)
(not (= δ_a^v #b10))
; enforcing default value if null
(=> (or (= δ_a^v #b01)
    (= α_a^v false/#b0...0))
```

Now, having an SMT-LIB equivalent of the variables and constraints for attributes and associations, we consider OCL operations.

SMT-LIB Realization of Transformation Rule 4.7.
The transformation of the `implies` *operation is mapped to tbe SMT-LIB format as follows:*

```
(let ( (ρ_implies (ite (or (not ρ_lhs)
                           ρ_rhs)
                       true
                       false))
       (δ_implies (ite (or (not ρ_lhs)
                           ρ_rhs
                       (and (isRegular δ_lhs)
                            (isRegular δ_rhs)))
                       ✓
                       (ite (or (isInvalid δ_lhs)
                                (isInvalid δ_rhs))
                            ⊥
                            ε)))
    )
    transformed OclExpression above
    using ρ_implies and δ_implies
)
```

Other OCL operations can be mapped to the SMT-LIB format in a similar fashion as it has just been shown for `implies`. The main idea is to map every node of the AST to a **`let`** block such that the local variables ρ_{op} and δ_{op} can be used in the node above. By this, the corresponding constraints for OCL invariants can successively be mapped to SMT-LIB.

Following this scheme, the complete UML/OCL model and all according constraints and validation/verification tasks can be combined to finally form one SMT-LIB instance. As reviewed in Section 2, passing the resulting instance to an SMT solver either yields a system state (representing a witness or counterexample) or proves that no such system state exist. But in contrast to existing solutions as proposed in [5–7, 9, 10, 14–18, 22, 23], the translation and mapping scheme proposed here additionally supports irregular values and, thus, the complete expressiveness of UML/OCL analysis. Existing approaches for UML/OCL verification and validation such as [15, 22, 23] can easily adapted to be used with the presented translation and mapping scheme—only the translations and mappings have to be adjusted accordingly. Moreover, the presented translation and mapping can easily be adapted to use a different solving engine such as CSP or ASP. In order to do so, only the mappings from the generic description to the solver input language have to be adjusted. The presented mapping for SMT-LIB works can be used as a standard pattern for those mappings.

6 CONCLUSIONS

In this paper, we presented a translation of UML/OCL models to a generic symbolic formulation that provides a native support for the irregular values `null` and `invalid`. While previously there were only solutions that require a high manual effort or apply severe restrictions to the supported modeling concepts, the proposed solution supports irregular values in a very comprehensive manner and, in principle, does not apply any nameable restrictions to the considered models. Moreover, as we have demonstrated for the case of SMT solvers, the generic symbolic formulation can easily be translated to a solver-specific formulation which allows for the use of powerful reasoning engines. As a consequence, the presented solution, for the first time, enables designers to conduct automatic validation and verification of UML/OCL models including a proper treatment of exceptional cases as they are indicated by OCL's irregular values.

ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SELFIE under grant no. 01IW16001.

REFERENCES

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB).* 2016. URL: http://www.SMT-LIB.org.

[2] Achim D. Brucker and Burkhart Wolff. "HOL-OCL: A Formal Proof Environment for UML/OCL". In: *Int'l Conf. on Fundamental Approaches to Software Engineering.* 2008, pp. 97–100.

[3] Jordi Cabot, Robert Clarisó, and Daniel Riera. "UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming". In: *Int'l Conf. on Automated Software Engineering.* 2007, pp. 547–548.

[4] Carolina Dania and Manuel Clavel. "OCL2MSFOL: A Mapping to Many-sorted First-order Logic for Efficiently Checking the Satisfiability of OCL Constraints". In: *Int'l Conf. on Model Driven Engineering Languages and Systems.* 2016, pp. 65–75.

[5] Martin Gogolla, Jörn Bohling, and Mark Richters. "Validating UML and OCL models in USE by automatic snapshot generation". In: *Software and System Modeling* 4.4 (2005), pp. 386–398.

[6] Martin Gogolla, Jörn Bohling, and Mark Richters. "Validation of UML and OCL Models by Automatic Snapshot Generation". In: *Int'l Conf. on the Unified Modeling Language, Modeling Languages and Applications.* 2003, pp. 265–279.

[7] Martin Gogolla, Fabian Büttner, and Mark Richters. "USE: A UML-based specification environment for validating UML and OCL". In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 27–34.

[8] Martin Gogolla and Mark Richters. "Expressing UML Class Diagrams Properties with OCL". In: *Int'l Workshop in OCL and Textual Modeling.* 2002, pp. 85–114.

[9] Carlos A. González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. "EMFtoCSP: a tool for the lightweight verification of EMF models". In: *Int'l Workshop on Formal Methods in Software Engineering - Rigorous and Agile Approaches.* 2012, pp. 44–50.

[10] Mirco Kuhlmann and Martin 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.

[11] Object Management Group. *Object Constraint Language – Version 2.4.* Version 2.4. Feb. 3, 2014. 230 pp. URL: http://www.omg.org/spec/OCL/2.4.

[12] Object Management Group. *OMG Systems Modeling Language (OMG SysML™).* June 3, 2015. 346 pp.

[13] Object Management Group. *OMG Unified Modeling Language TM (OMG UML) – Version 2.5.* Version 2.5. Mar. 1, 2015. 230 pp. URL: http://www.omg.org/spec/UML/2.5/.

[14] Nils Przigoda, Christoph Hilken, Robert Wille, Jan Peleska, and Rolf Drechsler. "Checking concurrent behavior in UML/OCL models". In: *Int'l Conf. on Model Driven Engineering Languages and Systems.* 2015, pp. 176–185.

[15] Nils Przigoda, Mathias Soeken, Robert Wille, and Rolf Drechsler. "Verifying the structure and behavior in UML/OCL models using satisfiability solvers". In: *IET Cyper-Phys. Syst.: Theory & Appl.* 1.1 (2016), pp. 49–59.

[16] Nils Przigoda, Robert Wille, and Rolf Drechsler. "Analyzing Inconsistencies in UML/OCL Models". In: *Journal of Circuits, Systems, and Computers* 25.3 (2016).

[17] Nils Przigoda, Robert Wille, and Rolf Drechsler. "Contradiction Analysis for Inconsistent Formal Models". In: *Int'l Symp. on Design and Diagnostics of Electronic Circuits & Systems.* 2015, pp. 171–176.

[18] Nils Przigoda, Robert Wille, and Rolf Drechsler. "Leveraging the Analysis for Invariant Independence in Formal System Models". In: *Euromicro Conf. on Digital System Design.* 2015, pp. 359–366.

[19] Heinz Riener, Oliver Keszocze, Rolf Drechsler, and Görschwin Fey. "A Logic for Cardinality Constraints (Extended Abstract)". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen.* 2014, pp. 217–220.

[20] James Rumbaugh, Ivar Jacobson, and Grady Booch, eds. *The Unified Modeling Language reference manual.* Essex, UK, 1999. ISBN: 0-201-30998-X.

[21] Mathias Soeken, Robert Wille, and Rolf Drechsler. "Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models". In: *Tests and Proof.* 2011, pp. 152–170.

[22] Mathias Soeken, Robert Wille, and Rolf Drechsler. "Verifying dynamic aspects of UML models". In: *Design, Automation and Test in Europe.* 2011, pp. 1077–1082.

[23] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. "Verifying UML/OCL models using Boolean satisfiability". In: *Design, Automation and Test in Europe.* 2010, pp. 1341–1344.