Ground Setting Properties for an Efficient Translation of OCL in SMT-based Model Finding

Nils Przigoda[*†]
[*]Group of Computer Architecture
University of Bremen
28359 Bremen, Germany

Robert Wille[‡†]
[‡]Institute for Integrated Circuits
Johannes Kepler University Linz
4040 Linz, Austria

Rolf Drechsler[*†]
[†]Cyber-Physical Systems
DFKI GmbH
28359 Bremen, Germany

{przigoda,drechsle}@informatik.uni-bremen.de

robert.wille@jku.at

ABSTRACT

Model Finding is an established method to increase the confidence in the correctness of a UML/OCL model, e. g., by automatically determining valid system states or counterexamples. In the recent past, numerous approaches have been proposed for this purpose. In order to cope with the underlying complexity, approaches based on satisfiability solvers have been found promising. They require a translation of all OCL constraints of the model for a corresponding solver.

In this paper, SMT-based model finding is investigated. It is shown that certain OCL operations are causing huge SMT formulations which harm the solving process. However, this is not necessary if a fixed structure of the model can be assumed. Motivated by this, a new concept called ground setting properties is introduced which allows for an efficient translation of OCL into SMT. This concept is illustrated by means of a running example and compared to existing solutions.

1. INTRODUCTION

With the increasing complexity of software as well as hardware systems, researchers started to investigate the integration of modeling languages in the design of all kinds of systems. In this context, modeling languages, such as the *Unified Modeling Language* (UML [1]) as one of the best-known representatives, received much attention. Related languages such as the *Object Constraint Language* (OCL) [2] additionally allow to extend a UML model with additional textual constraints, e. g., to define invariants or operation contracts.

A crucial requirement in the design process of a system in general is its validation and verification, i. e., the question how to check whether the given system is consistent and will work as intended. Consequently, researchers and engineers developed corresponding methods and tools for the validation and verification of system descriptions in UML/OCL. As an example, the *UML-based Specification Environment* (USE) [3] provides well-established methods that can be applied. Besides that, researchers began to exploit formal methods for the validation and verification of UML/OCL models. Approaches based on theorem provers like PVS [4], HOL-OCL/Isabelle [5], and KeY [6] have been applied for this purpose. They are capable of checking large models, but often require a strong formal background of the designer. As a consequence, researchers started to investigate the application of fully automatic proof engines including methods based on constraint programming (CSP) [7, 8, 9], description logic [10, 11], the modeling language Alloy based on relational logic [12, 13, 14], or Boolean satisfiability (SAT) [15, 16, 17, 18, 19].

Most of the automated approaches translate the UML model into a problem formulation of another domain where powerful solving engines can be applied. After the problem formulation has been solved, the respective results are translated back to the UML domain, e. g., in terms of a system state or a sequence of them. The process of obtaining a valid model instance (or proving that none exists) is called *model finding* and the respective tools are called *model finders*. However, for all automated model finders scalability is almost always an issue.

In this paper, we consider in detail how the translation of OCL constraints to the SAT-based SMT domain is conducted for model finding. These investigations unveil that for certain OCL operations (in particular, nested OCL navigation and iterator expressions) frequently a huge SMT formulation is created which significantly harms the efficiency of model finding. Moreover, further investigations show that the vast majority of the resulting SMT formulations are often not necessary and, in fact, are ignored during the actual model finding. Vice versa, just skipping the SMT formulation prior to model finding yields an SMT instance which does not represent the model in an adequate fashion.

Hence, we propose a compromise by introducing the concept of ground setting properties. They allow the designer to additionally provide some obvious information which, in many cases, is easily available prior to model finding. Afterwards, this information is utilized in order to prune huge parts of SMT formulations which are not necessary anymore considering the additional information. By this, SMT formulations can be generated which have only a fraction of the size of the originally created ones. Using an initial implementation, this yields reductions of up to 4 orders of magnitude of the used memory and allows to solve problems that could not be handled before.

The remainder of this work is structured as follows. The following section provides the basics and notations used in this work and reviews the main idea as well as concepts of SMT-based model finding. In Section 3 the running example is introduced. Afterwards, in Section 4 the resulting model

is used to investigate the translation of invariants defined in OCL to SMT constraints and to show how this may yield unnecessarily huge SMT formulations. Motivated by this, we introduce the concept of ground setting properties and discuss their relations to related work in Section 5. Finally, Section 6 deals with the implementation and evaluation, before Section 7 concludes the paper.

2. PRELIMINARIES

In order to keep the paper self-contained, this section provides the notation for UML/OCL models used in this paper. Furthermore, the basics on SMT-based Model Finding are briefly reviewed. For a more detailed treatment of the respective issues, we refer to the related work cited in this section.

2.1 UML/OCL Models

Whenever *UML/OCL models* are mentioned in this paper, class diagrams enriched with OCL description means are meant. A *class diagram* consists of classes C and references R (also called associations). A class is a 3-tuple composed of sets of attributes, invariants, and operations and is formally denoted by $c = (A, I, O) \in C$. A reference $r = (c_1, c_2, (l_1, u_1), (l_2, u_2)) \in R$ is a 4-tuple, where $c_1, c_2 \in C$ are the two connected classes and the latter two entries of the 4-tuple are multiplicity constraints. They restrict how often the reference is allowed to be instantiated for each object of the class c_1 or c_2, respectively. In the remainder of this work, we state with the term *model elements*, denoted by μ, the union of all attributes (of all classes) and all references. The invariants I of each class are (additional) OCL constraints [2] which restrict the possible system states that can be instantiated from the UML/OCL model.

Note that we restrict ourselves to binary associations. This restriction does not decrease expressiveness, since it has been shown that models containing n-ary associations can be mapped into a semantically equivalent model solely composed of binary associations by adding a helping class and some invariants to the affected classes [20]. Furthermore, modeling languages such as EMF [21] do not support n-ary associations at all without loading the UML metamodels.

EXAMPLE 1. *Figure 1 shows a class digram, i. e., in our terms a UML/OCL model, composed of the classes* System *and* Counter. *Both classes are connected with a reference, indicated by a line. The multiplicities as well as the reference end names (used, e. g., as identifier for navigation through OCL constraints) are annotated above and below the line, respectively. The class* Counter *has an integer attribute for its* `id` *and another one for its* `value`. *Additionally, there are two invariants for the class* Counter. *The invariant* `greaterEqualZero` *enforces the attribute* `value` *to be greater or equal than* `0`. *The second invariant* `uniqueId` *requires that the* `id` *of the counter is unique within all instances of the counters of the connected system.*

For a given UML/OCL model m, a *system state* $\sigma = (\Upsilon, \Lambda)$ is a tuple composed of a set of object instances of the classes of m and a set of links, i. e., instances of the references of m. A system state is called *valid*, if the multiplicities of the references and also all invariants are satisfied. Otherwise, the system state is called *invalid*. If for a given model m at least one valid system state exists, m is called *consistent*. Otherwise, m is called *inconsistent*.

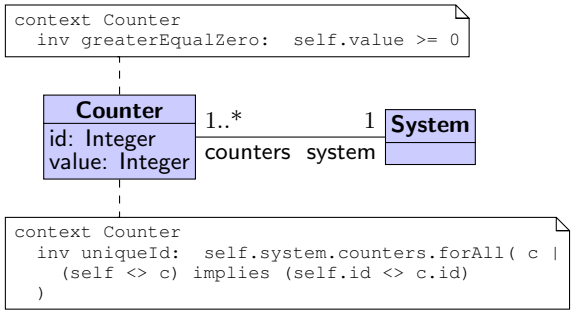

Figure 1: A model of a simple counter

2.2 SMT-based Model Finding

In order to check whether a given UML/OCL model is consistent or whether it satisfies, e. g., certain behavioral properties, several approaches for *model finding* have been introduced recently (see, e. g., [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 14, 18, 19]).

They take a given UML/OCL model and aim for determining a valid system state or a sequence of system states witnessing the consistency or behavioral property, respectively. In this work, we are focusing on model finders which are based on solvers for *Satisfiability Modulo Theory* (SMT).

For this work, SMT can be seen as an extension of the *Boolean satisfiability problem* (SAT problem), which 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 such that f evaluates to `true` or to prove that no such assignment exists. SMT extends this concept by additionally allowing to formulate the problem in bit vector logic rather than pure Boolean logic. In the remainder of this work, SMT formulations are provided in SMT-LIB syntax specified in [22]. Here, each constraint is provided following the Polish notation, i. e., each operation is encapsulated by parenthesis and the operator is provided before the list of (ordered) operands. For a detailed list of all operators and the logic descriptions the reader is referred to [22].

For SMT problems, very efficient solving engines are available which are capable of solving rather large formulations (see `http://www.smtcomp.org`). Motivated by that, researchers investigated their potential for the task of model finding – yielding *SMT-based model finders* as, e. g., introduced in [16, 17, 18, 19]. To this end, the question "Does a consistent system state or a sequence of system states showing the considered behavior exist?" has to be formulated in terms of a satisfiability problem. This requires the provision of problem bounds, i. e., an interval for the number of instantiations for each class. Then, a symbolic formulation representing all possible system states (or all sequences of system states) is created and formulated using the above-mentioned SMT-LIB syntax.

More precisely, for each attribute in all possible object instances of all classes, an SMT variable is created. The size, i. e., the bit vector length, of these variables depends on the domain of the attribute. For example, for an integer 8 bits may be used, while a set of integers consequently may require 256 bits ($2^8 = 256$). A Boolean attribute can be represented by one bit.[1] References are symbolically encoded by creating a bit vector variable for each object which can be used within a link. The corresponding sizes depend

[1]Note that, for sake of clarity, we are not considering `null` and `invalid` values, but only the *regular* domain of an attribute.

```
((_ card_ge 1) |System@0::counters|)
((_ card_ge 1) |System@1::counters|)
((_ card_eq 1) |Counter@0::system|)
((_ card_eq 1) |Counter@1::system|)
((_ card_eq 1) |Counter@2::system|)
(bvsgt |Counter@0::value| #x00)
(bvsgt |Counter@1::value| #x00)
(bvsgt |Counter@2::value| #x00)
(and (=> (= ((_ extract 0 0) |Counter@0::system|) #b1)
         (and (=> (= ((_ extract 0 0) |System@0::counters|) #b1)
                  (=> (not (= #b001 #b001))
                      (not (= |Counter@0::id| |Counter@0::id|))))
              (=> (= ((_ extract 1 1) |System@0::counters|) #b1)
                  (=> (not (= #b001 #b010))
                      (not (= |Counter@0::id| |Counter@1::id|))))
              (=> (= ((_ extract 2 2) |System@0::counters|) #b1)
                  (=> (not (= #b001 #b100))
                      (not (= |Counter@0::id| |Counter@2::id|))))))
     (=> (= ((_ extract 1 1) |Counter@0::system|) #b1)
...))
```

Listing 1: SMT constraints for the system-counter model

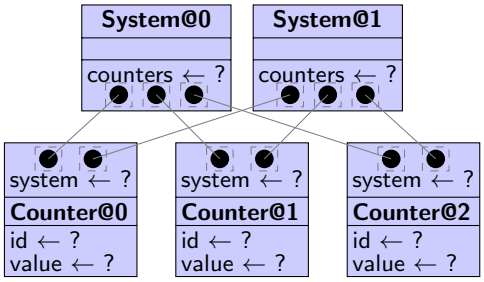

Figure 2: A system state for the simple counter model

on the possible number of opposite ends and, hence, on the problem bounds. The following example illustrates the main ideas.

Example 2. *Consider again the system-counter model from Figure 1 and assume we want to check whether this model is consistent for exactly two instances of class* System *and exactly three instances of class* Counter. *Figure 2 sketches the variables created for the corresponding SMT formulation. For all three* Counter *instances, there is one variable for the attribute* `id` *and another one for the attribute* `value`. *The variables for the reference ends are placed in boxes below (above) the* System *(*Counter*) object instances. Here, each dot within a gray dashed rectangle represents a single bit of the bit vector variable indicating whether there is a link between the objects (highlighted by a gray line) or not.*

Passing just this list of variables to an SMT solver obviously leads to an arbitrary assignment of all variables and consequently to an arbitrary system state. Hence, SMT constraints must be added in order to restrict the set of possible assignments such that a valid assignment is determined which represents a consistent system state. This particularly requires constraints enforcing the multiplicities as well as the OCL invariants. Basic concepts for that have been introduced in [16] and more detailed mappings, e. g., from OCL to SMT are available in [17]. The following example only illustrates the main ideas.

Example 3. *Consider again the example from Figure 1 and the corresponding SMT variables as shown in Figure 2. In order to determine* valid *assignments only, SMT constraints as sketched in Listing 1 are added. The first five lines enforce the multiplicities of the reference between the two classes, while the next three lines enforce the invariant* `greaterEqualZero`. *Invariant* `uniqueId` *is enforced by the constraints in the remaining lines (partially illustrated for* Counter@0 *only).*

Passing the resulting SMT formulation to an SMT solver either yields a satisfying assignment to all SMT variables or a proof that no such assignment exists. In the former case, a consistent system state or a sequence of system states can easily be derived from the assignment to the SMT variables. In the latter case, it has been proven that no such system state or sequence of system states exists (within the defined problem bounds).

3. CONSIDERED RUNNING EXAMPLE

The contributions of this work will be illustrated and evaluated by means of a running example, namely a model of the *One Tile Game* – a variant of the well-known *Wythoff's Game* which is a so-called *Nim* game. This section briefly reviews the game and its rules as well as its formulation in terms of a UML/OCL model. Based on that, the considered problem as well as the proposed solution are explained in the remainder of this work.

3.1 Considered Scenario

The *One Tile Game* is played with a single tile on a board with x columns and y rows. At the beginning, the tile is placed on the north-east field of the board. Then, two players A and B alternatingly move the tile either one or two fields towards the south, the south-west, or the west. Figure 4 sketches the allowed moves at the beginning on a 6×6 board. Player A starts the game. A player wins the game, if he/she is capable of performing the last valid move, i. e., if the other player is not able anymore to move the tile on the board.

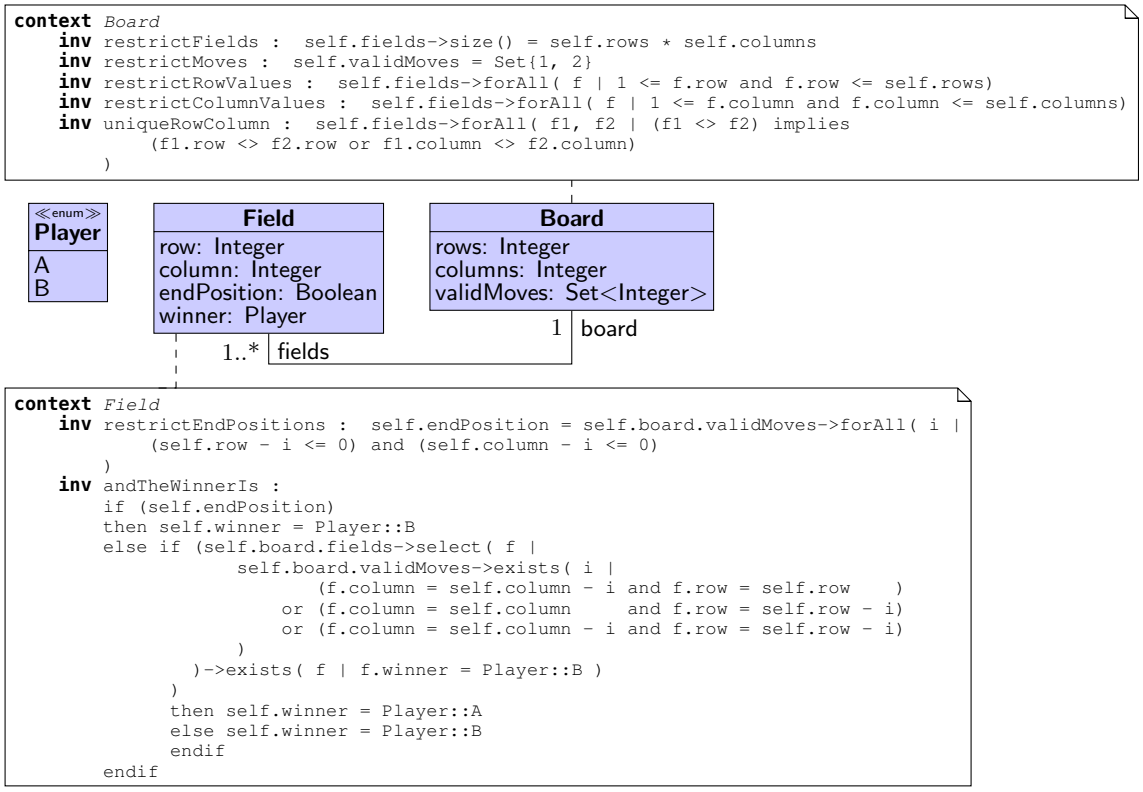

Figure 3: The one-tile-game as UML/OCL model

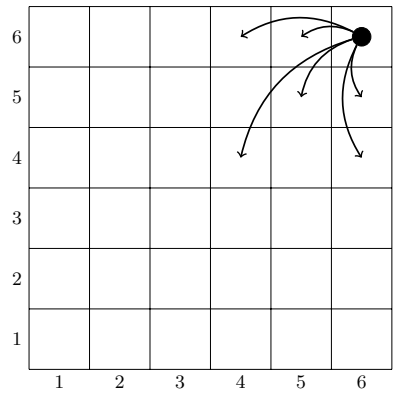

Figure 4: One Tile Game – a Nim variant

If we assume that both players are playing with a perfect strategy – and they can do so because the *One-Tile-Game* has perfect information, i. e., all information about decisions and/or possible moves is available at any time –, it is obvious that the winner of the game can be determined before starting the game.[2] Because of this, it is possible to mark each field of the board with the player that is going to win the game if the game starts on this field. Now, the question which player wins the game when it is started on an arbitrary field can be tackled as a model finding instance. To this end, a UML/OCL model (here, in terms of a class diagram) has to be available.

[2]Note that Player B wins, if the game starts on field (1,1), since Player A has to make the first move but no valid options to do so.

3.2 Corresponding UML/OCL Model

To model the *One Tile Game* with UML/OCL, i. e., as a class diagram, we introduce a class representing the entire board as well as a class representing the single fields of the board as shown in Figure 3. Further, an enum *Player* is defined, which is used as data-type to mark what player is going to win when the game starts on a particular field.

The class *Board* additionally provides attributes to define the number of rows and columns. Another attribute, namely `validMoves`, stores all the possible numbers of fields which the tile can be moved. More precisely, this means, that for the *One Tile Game* 1 and 2 should be the only elements in `validMoves`, since the tile can be moved either 1 or 2 fields. This attribute allows for a quick consideration of further variants of the game in which you can move the tile by more or less fields.

Since all fields in the actual game have a unique position defined by its row and column, the class *Field* is equipped with corresponding attributes. Besides this, an attribute `endPosition` indicates whether a valid move is still possible from this field or not (i. e., `endPosition` defines whether the field is the terminal/end field). Additionally, an attribute `winner` is used to mark the winner for this field as described above – if, e. g., `winner` is set to A, Player A would win if the game is started on this field and an optimal strategy is used. Finally, each field is connected through a reference with exactly one board, while the board must be connected to at least one field.

Up to this point, a nearly arbitrary number of objects per class as well as arbitrary values for the attributes would constitute a valid system state for this class diagram. However, since we are still aiming only for configurations of object instances which correctly represent different versions of boards

as well as the respectively correct winners for each field, the attribute values have to be restricted by OCL invariants as shown at the top of Figure 3. More precisely, for a board with x rows and y columns the number of connected fields should be restricted by the product of `rows` and `columns`. This is done by means of the invariant `restrictFields`. Since in the *One Tile Game*, the tile can only be moved by one or two fields, `validMoves` is restricted by the invariant `restrictMoves`. Because the values of `row` as well as `column` of each field are restricted and unique within a board, three additional invariants for *Board* have been added. Altogether, the five invariants ensure that a valid system state represents a board where all corresponding fields are properly representing a board as required for the *One Tile Game*.

Next, invariants further restricting the set of valid system states are added. These invariants enforce correct markings of the respective winners for each field and are provided at the bottom of Figure 3. More precisely, `restrictEndPositions` is added enforcing that, for each field, `endPosition` is set to `true` if no valid moves can be performed from this field anymore or to `false` otherwise. The invariant `andTheWinnerIs` first checks if the fields constitutes an end position; then the winner must be Player B. Otherwise, it is checked whether there is at least one reachable field with `winner` set to Player B; then the winner must be Player A. If not, Player B wins.

Passing the resulting class diagram together with its invariants to a model finder eventually yields a valid system state representing various possible instances of the *One Tile Game* and their respective marking of winners. However, existing SMT-based model finders significantly suffer from OCL descriptions as depicted in Figure 3 and, hence, are hardly applicable particularly for large instances. In order to illustrate the problem (and provide a solution), a more detailed look on how OCL invariants are handled by state-of-the-art model finders is necessary.

4. TRANSLATION OF OCL INVARIANTS AND RESULTING PROBLEM

In this section, the translation of OCL invariants into an equivalent SMT formulation (to be used for model finding as reviewed in Section 2.2) is investigated in more detail. Based on this, we afterwards show how dedicated OCL constraints may cause a "blow-up" of the resulting SMT formulation and, hence, pose a serious obstacle to the efficiency of SMT-based model finding. By this we are illustrating the main problem which is considered in this work. After that, Section 5 introduces our solution with which we are addressing this problem.

4.1 Translation of OCL Invariants

As reviewed in Section 2.2, SMT-based model finding first creates variables in order to represent all possible system states symbolically. Considering the running example introduced in the previous section, all variables of a corresponding symbolic formulation for a 3×3 board are illustrated in Figure 5. Since the assignment of all variables is unknown prior to model finding, all values are denoted by a "?". Note that the reference ends are handled by variables inside the objects, namely `fields` or `board`, respectively.

Next, SMT constraints enforcing the invariants of the UML/OCL model to the symbolic formulation are created. For the five invariants of class *Board*, this can be conducted using existing solutions such as proposed in [16, 17].

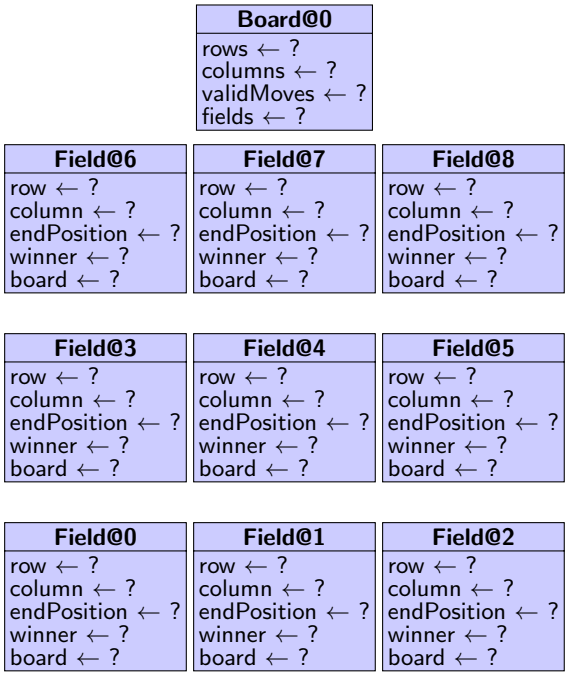

Figure 5: Symbolic formulation for a 3×3 board

More precisely, the `forAll` iterations in the invariants `restrictRowValues` and `restrictColumnValues` are unfolded for all fields, i. e., the internal constraints are translated nine times, while the `forAll` with two iterators in the invariant `uniqueRowColumn` is unfolded $9 \cdot 9 = 81$ times. The `size` operation is internally translated into $|Field| + 1$ constraints – each of them checks if for an $i \in \{0, \ldots, |Field|\}$ the board is connected to exactly i fields. Listing 2 sketches the resulting SMT constraints for the invariant `restrictRowValues`.

However, existing solutions such as proposed in [16, 17] will reach their limits when translating invariants of the class *Field*. In order to illustrate that, let us first consider the invariant `restrictEndPositions`: Here, `self.board` represents a reference of the connected board which is unknown prior to model finding (since the precise connections are to be determined by a model finder). Accordingly, a corresponding SMT formulation has to consider all possibilities. Assuming that only one object of class *Board* is instantiated, this does not cause a problem (since this would result in only one possible reference). But as soon as *Board* would be instantiated multiple times, the number of possible references quickly multiplies.

The consideration of all possibilities really becomes a problem when the access of the attribute `validMoves` is translated into SMT. Again, the precise value of this attribute is unknown prior to model finding. In principle, this set can contain integers between 0 and 255 (assuming an integer is encoded with 8 bits). For each of these possible integers, the internal OCL constraint must be translated into an equivalent SMT constraint. Afterwards, all these SMT constraints are joined by an `and` expression and the result must be equal to `self.endPosition`. For *Field@0*, this yields an SMT constraint as sketched in Listing 3.[3] Since such an SMT constraint enforcing the invari-

[3]Note that, in Listing 3, unsigned bit vectors have been used and, thus, the comparison looks a bit different as in the corresponding OCL constraints.

```
(and (=> (= ((_ extract 0 0) |Board@0::fields|) #b1)
         (and (bvule #x01 |Field@0::row|)
              (bvule |Field@0::row| |Board@0::rows|)))
     (=> (= ((_ extract 1 1) |Board@0::fields|) #b1)
         (and (bvule #x01 |Field@1::row|)
              (bvule |Field@1::row| |Board@0::rows|)))
     (=> (= ((_ extract 2 2) |Board@0::fields|) #b1)
         ...)
     ...
     (=> (= ((_ extract 8 8) |Board@0::fields|) #b1)
         (and (bvule #x01 |Field@8::row|)
              (bvule |Field@8::row| |Board@0::rows|))))
```

Listing 2: SMT constraints for the invariant `restrictRowValues`

```
(= |Field@0::endPosition|
   (and (ite (= ((_ extract 0 0) |Board@0::validMoves|) ; check if i = 0 is in validMoves
                #b1)
             (and (or (bvult (bvsub |Field@0::row| #x00) ; #x00 represents i = 0
                             |Field@0::row|)
                      (not (= (bvsub |Field@0::row| #x00)
                              #x00)))
                  (or (bvult (bvsub |Field@0::column| #x00)
                             |Field@0::column|)
                      (not ((= (bvsub |Field@0::column| #x00)
                               #x00)))))
             true)
        (ite (= ((_ extract 1 1) |Board@0::validMoves|)
                #b1) ...)
        ...
        (ite (= ((_ extract 255 255) |Board@0::validMoves|)
                #b1) ...) ) )
```

Listing 3: SMT constraints for the invariant `restrictEndPositions`

ant `restrictEndPositions` has to be created for all nine field instances (with the 256 internal checks caused by the unknown set), this easily "blows up" the resulting SMT instance – a serious problem which significantly degrades the performance of SMT-based model finding.

The second invariant of *Field*, namely `andTheWinnerIs`, causes even more problems. Consider here the condition of the inner `if-then-else` expression. First, the correct reference of the board must be identified – this is similar to the first invariant. On top of that, the `select`-expression additionally has to consider all possible fields. Within the scenario of the running example, this iteration should only consider fields which are reachable by a valid move. In order to decide if a field is reachable by a valid move or not, an iteration over `self.board.validMoves` and its up to 256 different values is required. Moreover, a constraint checking whether player B wins the game for at least one of these selected fields has to be added. Since what fields are selected here is unknown prior to model finding, the constraints have to be generated for all fields.

4.2 Consequences and Resulting Problem

In the cases discussed above, a huge overhead in the SMT formulation is created. In fact, already the running example considered here with a 3×3 board requires a total of approx. 2 MB and 500 MB of memory to store the SMT formulations just for the invariants `restrictEndPositions` and `andTheWinnerIs`, respectively.[4] Here, the consideration of all possible cases, particularly for (nested) OCL navigation and iterator expressions, poses a significant obstacle to SMT-based model finding. Similar observations can be made for other UML/OCL models relying on navigation or iterator expressions on unknown large collections.

In contrast to that, most of the SMT constraints are never used by the applied model finder. For example, as soon as the model finder concludes that the iteration of `validMoves` only defines two fields because of the corresponding invariant (rather than all 256 ones which are possible in general), a huge amount of the generated SMT constraints is rendered to be irrelevant for further consideration.

The same happens as soon as the references between the fields and board as well as the positions of the fields are assigned by the model finder – then, it becomes clear which fields indeed are reachable from another field. Hence, although a complete and general SMT formulation is required in general, most of the constraints are not required as soon as partial assignments, e. g., on `validMoves` and `row`/`column` for the position, are applied.

Unfortunately, information on valid moves or reachable fields is not directly available in the model as given in Figure 3. However, the designer can easily assume a dedicated structure which explicitly provides such information without harming the generality of the model finding task. In fact, the order of the fields in the running example is practically irrelevant as long as they are organized in a grid-fashion. Employing such information could prevent the generation of a general SMT formulation which generically considers all possible cases. Instead, it would help to create only those

[4] Files of the resulting SMT formulations are online available: http://informatik.uni-bremen.de/agra/divers/gsp/

SMT constraints which are actually needed in the model finding process. To this end, the user has to provide additional information prior to the automatic translation from OCL to SMT.

On a first glance, a partial system state might be a good solution to incorporate those additional information in the translation flow. Unfortunately, this does not satisfactorily address the problem. Using partial system states, the solving engine can immediately satisfy huge parts of the resulting formulation (e. g., providing a fix structure of the board in the One-Tile-Game immediately satisfies the 5 invariants of class *Board* and huge parts of the 2 invariants of class *Field*). However, although they are immediately satisfied, simply not generating these constraints in the first place is not easily possible, since the generation procedure of the SMT constraints for the OCL expressions does not have knowledge about what constraints are affected by a partial system state. Hence, another solution is required.

5. GROUND SETTING PROPERTIES FOR EFFICIENT TRANSLATION OF OCL

In this section, we propose the concept of ground setting properties which enables the designer to explicitly specify and provide the structure of a given model to be considered during model finding. This additional information allows for omitting huge parts of the SMT formulation and, hence, yields a significantly more efficient model finding. In the following, ground setting properties themselves and how to specify them is described. Afterwards, we outline how the generation of an SMT formulation profits from ground setting properties. Finally, the difference of this concept compared to similar and related work is discussed.

5.1 Ground Setting Properties

As discussed above, the significant overhead of the SMT formulation can be avoided if additional information (e. g., about the structure) of the considered model is explicitly given. In order to provide the model finder with this information, ground setting properties are introduced in this section.

In addition to other existing properties which a model element μ might already have (such as, e. g., Changable, Name, and Unsettable in EMF), we are proposing to add a further property called *ground setting* by an annotation. Each *ground setting property* of a model element μ has one of the two Boolean values: If the values of all instances of μ can be assumed to have a fixed value, then the ground setting property is set to `true` (stating that additional information can be exploited when generating the SMT formulation). If at least one value of an instance of μ can not be assumed to have a fixed value, then the ground setting property is set to `false` (stating that the generic SMT formulation to be created has to consider all possibilities).[5] In the former case (the ground setting property of μ is set to `true`), the additional information to be exploited has to be provided by corresponding values for each instance of μ.

Applying this concept to the running example, ground setting properties are set as summarized in Table 1, where ✓ and ✗ represents `true` and `false`, respectively. More precisely, the ground setting properties of the attributes `rows`, `columns`, and `validMoves` of class *Board* can be set to `true` (because the designer usually knows the sizes of the board to be considered as well as the value of the attribute `validMoves`). Furthermore, the ground setting properties of the attributes `row` and `column` of class *Field* can be set to `true` (because, without loss of generality, the positions of the fields can be set). Finally, the ground setting property of the reference between *Board* and *Field* is also set to `true`, because the designer knows which fields are connected to which board. For all remaining model elements, the ground setting properties are set to `false` – those are the attributes from which we would like the values to be determined by the model finder.

[5]Note that `false` is assumed as default value.

Table 1: Ground setting properties for the running example

Class	Model element	Ground Setting Property
Board	`rows`	✓
	`columns`	✓
	`validMoves`	✓
Field	`row`	✓
	`column`	✓
	`endPosition`	✗
	`winner`	✗
Reference fields-board		✓

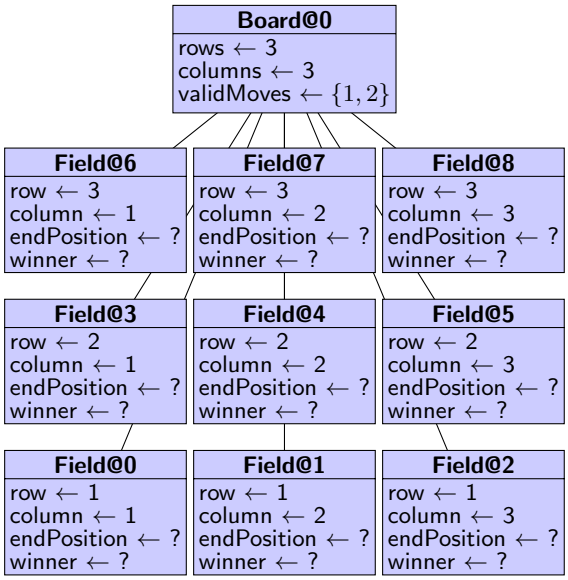

Figure 6: Additional values for the running example

Besides that, additional information on the values for all model elements μ whose ground setting property is set to `true` has to be provided. Therefore, a partial system state is used. Thanks to the ground setting annotation at the model level, it is now possible to access the information of the partial system state during the translation from OCL to SMT; without the annotation for ground setting properties, this would be not possible. For the running example, these values are summarized in Figure 6 (the links between objects are indicated by a connecting line in the background). Overall, this provides more information which does not significantly harm the generality of the model finding task (as discussed above), but, as described next, allows for a much more efficient SMT formulation.

5.2 Efficient Translation of OCL

Given the information from the subsection before, it is now possible to identify a model element where the ground setting property is set to `true`. For those model elements, the SMT formulation can directly use the provided value

```
(= |Field@0::endPosition|
   (and ( // constraints for 1 )
        ( // constraints for 2 ) )
// constraints for 1:
(and (or (bvult (bvsub |Field@0::row|
                       #x01)
                |Field@0::row|)
         (not (= (bvsub |Field@0::row|
                        #x01)
                 #x00)))
     (or (bvult (bvsub |Field@0::column|
                       #x01)
                |Field@0::column|)
         (not ((= (bvsub |Field@0::column|
                         #x01)
                  #x00)))))
```

Listing 4: Opt. SMT cnstrs. for `restrictEndPositions`

```
// constraints for 1:
(and (or (bvult (bvsub #x01
                       #x01)
                #x01)
         (not (= (bvsub #x01
                        #x01)
                 #x00)))
     (or (bvult (bvsub #x01
                       #x01)
                #x01)
         (not ((= (bvsub #x01
                         #x01)
                  #x00)))))
```

Listing 5: Additionally applying values

```
(= |σ_o::Field@0::endPosition|
   true)
```

Listing 6: Completely optimized SMT constraints

instead of unfolding it completely and causing the problems described in Section 4. More precisely, it is now possible to directly use the respective value in the SMT formulation (of course it must be mapped to a bit string before) rather than relying on an SMT variable. In particular for (nested) OCL navigation and iterator expressions, this avoids the huge enumeration of all possibilities.

Applying this to the running example, the values of the `validMoves` set are given and, thus, the SMT formulation for the invariant `restrictEndPositions` can now be shortened as sketched in Listing 4. Because of the same reason, also the variables in the constraints, i. e., `Field@0::row` and `Field@0:column`, for the two elements can be replaced with its precise values. This is illustrated in Listing 5.

Already this yields substantial reductions. However, further optimizations can be achieved since many arguments of SMT constraints become constant due to the ground setting properties. This allows for pre-calculating the result of constraints and the direct application in the SMT formulation. Iteratively applying these pre-calculations to the constraint in Listing 5 finally yields the value `true`. Analogously, the second constraint in Listing 4 can be optimized to `true`. Putting both results together, the original SMT constraint from Listing 2 deflates to the one shown in Listing 6.

5.3 Discussion and Related Work

Ground setting properties sufficiently address the problem of generating huge SMT formulations for standard SMT-based model finding. However, other previously proposed solutions may provide an alternative solution as well. In this section, we review corresponding related work and discuss why they address the problem considered here in an inadequate fashion only.

An obvious solution to the considered problem might be the utilization of partial system states – without annotations for ground setting properties – which are also used to provide the values for model elements where the ground setting property is set to `true`. Instead of leaving all attributes and references unassigned, the designer could, e. g., provide a partial system state in which relevant model elements (such as `validMoves`, `rows`, and `columns` of the class *Board* as well as the attributes `row` and `column` of the class *Field* in the running example) are already pre-defined as done in Figure 6. Then, e. g., for each attribute, a further constraint is added which enforces the corresponding SMT variable to assume the respective value. By this, information on the structure of the board would be known, but only for the SMT solver and not for the translation process from OCL to SMT – the size of the SMT formulation would remain the same (in fact, the size would even slightly increase). Moreover, even if additional information from the partial system state would be exploited to improve the SMT formulation, many tools provide default values. Those default values are a problem because for the OCL to SMT translation an internal system state of the model is generate and whenever one of the attributes or references is used within OCL it is not possible differentiate between a default value (mostly `null`) and a real pre-assignment provided by the designer by a partial system state or not. Ground setting properties as proposed in this work provide a solution which entirely avoids problems like that.

The *UML-based specification environment* (USE) [3] offers a so-called *model validator* [14] which uses relation logic, Kodkod, Alloy, and SAT solvers. The main flow is thereby similar to SMT-based model finding, i. e., the USE model validator uses Kodkod [15] to transform the model, which itself uses Alloy [12] to eventually generate an equivalent SAT formulation to be solved. This model validator allows for a restriction of the domain of each attribute before starting with the complete translation chain among other configuration possibilities. By this, the designer could, e. g., add useful information which restricts the search space of the problem – similar to the ground setting properties proposed here. However, so far it is not possible to pre-assign a specific attribute of a respective class instance. Hence, the internal SAT constraints generated, e. g., for the nested `select-exists-exists` in the inner condition of the invariant `andTheWinnerIs` would cause a similar "blow-up" when increasing the number of *Field* instances as in the SMT-based solution (this is also confirmed by experimental evaluations later in Section 6).

Finally, Kodkod [15] internally supports symmetry breaking in order to avoid permutations within the search space to be considered by the solver. This is a great advantage compared to the approaches proposed in [16, 23] where no similar techniques have been used. However, to the best of our knowledge the "blow-ups" as described earlier will not be avoided in the last transformation step (to SAT). Hence, a similar problem as discussed in this work remains.

6. IMPLEMENTATION AND EVALUATION

The concept of ground setting properties has been implemented on top of an SMT-based model finder which has been implemented within the Eclipse framework. Here, UML/OCL models are represented in terms of the *Eclipse Modeling Framework* (EMF). In this section, we briefly review this environment and describe how the proposed concept has been realized and can be used. Afterwards, we evaluate the improvement of the performance which can be observed by exploiting ground settings.

6.1 Implementation

In order to apply the proposed solution, we have re-implemented the general concepts of SMT-based model finding as presented in [16, 23]. This resulted in an implementation as an Eclipse plugin using both, Java and Xtend. As the SMT solving engine, Z3 [24] is utilized. This setting has been enriched by the proposed concept of ground setting properties as described in Section 5. Eventually, this resulted in an extended framework which enables designers to easily add and configure ground setting properties as well as provide corresponding values.

Figure 7 exemplary illustrates this for the considered model of the One-Tile-Game. In Figure 7a the `emf` file of the One-Tile-Game is shown for the general model finding approach. This means that no ground settings properties are present. However, in order to add and configure a ground setting property, the designer must only add the following line in the text editor before a model element:

```
@ModelFinder(groundSettingProperty="true")
```

The result for the One-Tile-Game model is depicted in Figure 7b. If the designer prefers to work on the `ecore` files of the EMF, the same results can be achieved with *Sample Ecore Model Editor* provided by the EMF itself. There, the annotations can be added and configured with some clicks only.

The corresponding values can afterwards be provided using the XMI format which enables designers to declare a (partial) system state. The provided system state implicitly includes problems bounds for the number of objects to be instantiated for each class. Such an XMI file can be created using the editors provided by the EMF – the precise details can be analyzed and changed using, e. g., the *Sample Reflective Ecore Model Editor* as shown in Figure 8.

6.2 Evaluation

In this section, we compare the performance of SMT-based model finding with ground setting properties to the performance of the original approach. In addition to that, we also conducted comparisons to the USE model validator with restricted domains as discussed in Section 5.3. For SMT-based model finding, our implementations have been applied within *Eclipse* in version *Mars.2*. For USE, we applied version 4.2.0-465 with version 5588 of the model validator. All experiments have been carried out on an Intel i5 with 2.6 GHz cores and 8 GB memory using a Windows 7 64 bit and a Java 64 bit installation.

In a first evaluation, we considered the sizes of the obtained SMT formulations.[6] Table 2 provides the obtained numbers for the running example with several board sizes.

[6]Since the USE model validator does not provide any information on the size of the resulting SAT instance, only SMT-based model finding is considered in this part of the evaluation.

OneTileGame.emf

```
@namespace(uri="OneTileGame", prefix="OneTileGame")
package OneTileGame;

class Board {
  !ordered ref Field[+]#board fields;
  !ordered attr int[+] validMoves;
  !ordered attr int[1] rows;
  !ordered attr int[1] columns;
}

class Field {
  !ordered ref Board[1]#fields board;
  !ordered attr int[1] row;
  !ordered attr int[1] column;
  !ordered attr boolean[1] endPosition;
  !ordered attr Player[1] winner;
}

enum Player {
  A = 0;
  B = 1;
}
```

(a) without ground setting properties

OneTileGame.emf

```
@namespace(uri="OneTileGame", prefix="OneTileGame")
package OneTileGame;

class Board {
  @ModelFinder(groundSettingProperty="true")
  !ordered ref Field[+]#board fields;
  @ModelFinder(groundSettingProperty="true")
  !ordered attr int[+] validMoves;
  @ModelFinder(groundSettingProperty="true")
  !ordered attr int[1] rows;
  @ModelFinder(groundSettingProperty="true")
  !ordered attr int[1] columns;
}

class Field {
  @ModelFinder(groundSettingProperty="true")
  !ordered ref Board[1]#fields board;
  @ModelFinder(groundSettingProperty="true")
  !ordered attr int[1] row;
  @ModelFinder(groundSettingProperty="true")
  !ordered attr int[1] column;
  !ordered attr boolean[1] endPosition;
  !ordered attr Player[1] winner;
}

enum Player {
  A = 0;
  B = 1;
}
```

(b) with ground setting properties

Figure 7: One-Tile-Game model in the EMF format

Here, the problem investigated in Section 4 can clearly be seen. Since a general formulation is created which considers all possibilities, a "blow-up" in the SMT formulation results. Already for rather tiny boards gigabytes of memory are consumed by the formulation; boards sizes of just 5×5 could not even be handled because of memory limitations. In contrast to that, just some kilobyte of memory are required once ground setting properties are applied and exploited.[7]

In a second experiment, we compared the run times of the considered approaches. Figure 9 provides the resulting numbers in terms of three plots: the brown one shows the run times when applying the USE model validator while the red and blue one shows the run times when applying the SMT model finder without and with ground setting properties, respectively. The x-axis provides the size of the board, while the y-axis provides the run time in CPU seconds. Also

[7]Note that, without calculating the size of the SMT instance, the general SMT-based model finder was capable of determining solutions for 5×5 and 6×6 as well (but approached its limits for problems sizes of 7×7 or larger).

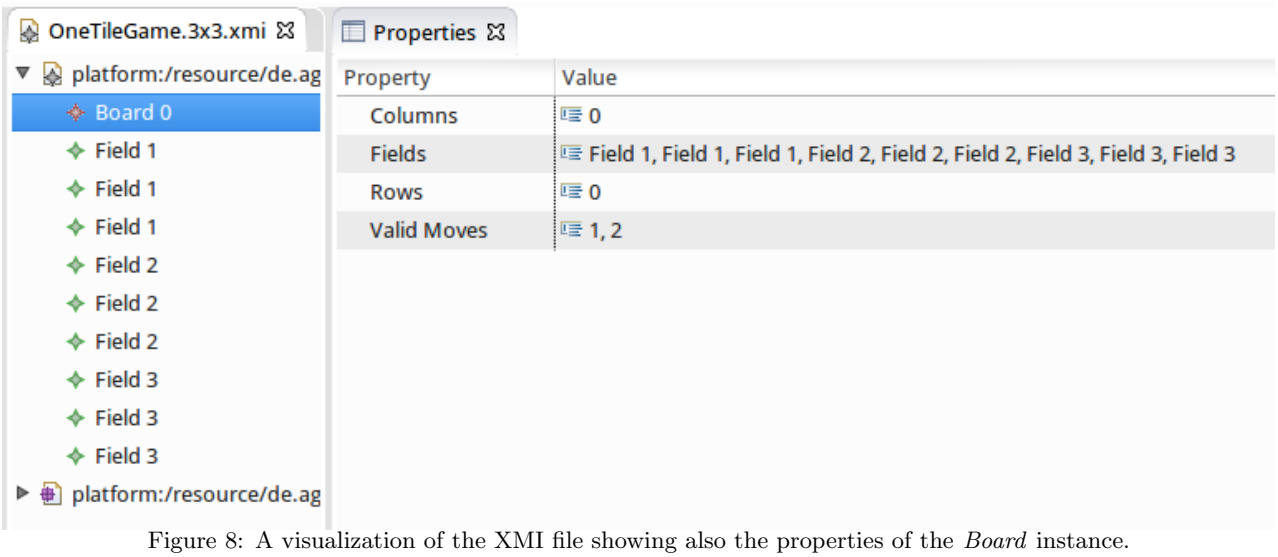

Figure 8: A visualization of the XMI file showing also the properties of the *Board* instance.

Table 2: Size of resulting SMT formulations

Field size	3 × 3	4 × 4	5 × 5
without gr. set. prop.	491.0 MB	3.4 GB	MO
with gr. set. prop.	7.4 kB	13.8 kB	22.4 kB
Field size	**10 × 10**	**20 × 20**	**30 × 30**
without gr. set. prop.	MO	MO	MO
with gr. set. prop.	96.1 kB	403,9 kB	936.5 kB

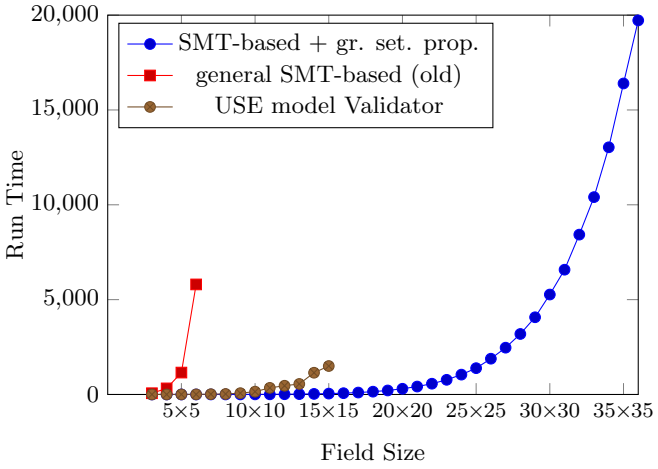

Figure 9: Run times for different board sizes

this evaluation clearly shows the superiority of the proposed approach. While the SMT-based model finder hardly can be applied even for rather tiny instances (which is not surprising considering the size of the SMT formulation as covered above), the USE model validator is, after all, capable of handling board sizes up to 15 × 15. However, both approaches are clearly outperformed by the proposed approach with ground setting properties which can even handle a 36 × 36 board in less than 20,000 seconds.

7. CONCLUSION

In this work, the translation from OCL to SMT for UML model finding has been investigated. Examples as well as experiments have shown that, in particular, nested navigation and iteration expressions of the OCL cause a severe "blow-up" of the SMT formulation. In contrast, a significant part of the SMT formulation is never used and can be omitted as soon as some additional information, e. g., of the structure of the model at hand is known prior to model finding. Motivated by that, we introduced the concept of ground setting properties – an additional property for model elements – which enables the designer to provide this additional information. Exploiting these properties and information, huge parts of the SMT-formulation can be pruned – eventually allowing for a significantly more efficient SMT-based model finding.

Our initial investigations showed that, exploiting ground setting properties, reductions of up to four orders of magnitude are possible and that instances which could not be handled before can now be tackled in negligible runtime. We expect that such improvements are not only possible for the considered One Tile Game setting, but also for other models which are generically described but usually assume a regular structure. This particularly holds, e. g., for models from domains such as graph problems (including their vast applications, e. g., in route finding, resource allocation, etc.) or hardware design for which, thus far, many SMT constraints have to be created in a general case, but most of them are not needed once a fix structure is instantiated for a dedicated model finding task. Since a detailed evaluation would have been beyond the scope of this work, we left corresponding cases studies and investigations for future work.

Acknowledgements

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

8. 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] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 27–34, 2007.

[4] M. Kyas, H. Fecher, F. S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons, and H. Kugler, "Formalizing UML Models and OCL Constraints in PVS," *Electronic Notes in Theoretical Computer Science*, vol. 115, pp. 39–47, 2005.

[5] A. D. Brucker and B. Wolff, "A Proposal for a Formal OCL Semantics in Isabelle/HOL," in *Theorem Proving in Higher Order Logics*, 2002, pp. 99–114.

[6] B. Beckert, R. Hähnle, and P. H. Schmitt, *Verification of Object-Oriented Software: The KeY Approach*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[7] J. Cabot, R. Clarisó, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," in *ICST Workshops*, 2008, pp. 73–80.

[8] T. Mancini, "Finite satisfiability of UML class diagrams by constraint programming," in *Description Logics*, 2004.

[9] H. Malgouyres and G. Motet, "A UML model consistency verification approach based on meta-modeling formalization," in *Proceedings of the ACM Symposium on Applied computing*, 2006, pp. 1804–1809.

[10] D. Berardi, D. Calvanese, and G. De Giacomo, "Reasoning on UML class diagrams," *Artif. Intell.*, vol. 168, no. 1, pp. 70–118, 2005.

[11] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers, "Using description logic to maintain consistency between UML models," in *UML*, 2003, pp. 326–340.

[12] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[13] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A Challenging Model Transformation," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, 2007, pp. 436–450.

[14] 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.

[15] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for Construction and Analysis of Systems*, 2007, pp. 632–647.

[16] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using Boolean satisfiability," in *Design, Automation and Test in Europe*, 2010, pp. 1341–1344.

[17] M. Soeken, R. Wille, and R. Drechsler, "Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models," in *Tests and Proof*, 2011, pp. 152–170.

[18] 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.

[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.

[20] M. Gogolla and M. Richters, "Expressing UML Class Diagrams Properties with OCL," in *Object Modeling with the OCL*, 2002, pp. 85–114.

[21] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2009.

[22] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," 2016. [Online]. Available: http://www.SMT-LIB.org

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

[24] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for Construction and Analysis of Systems*, 2008, pp. 337–340.