Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models

Mathias Soeken, Robert Wille, and Rolf Drechsler

Institute of Computer Science, University of Bremen
Computer Architecture Group, D-28359 Bremen, Germany
{msoeken,rwille,drechsle}@informatik.uni-bremen.de

Abstract. Checking the correctness of UML/OCL models is a crucial task in the design of complex software and hardware systems. As a consequence, several approaches have been presented which address this problem. Methods based on *satisfiability* (SAT) solvers have been shown to be very promising in this domain. Here, the actual verification task is encoded as an equivalent bit-vector instance to be solved by an appropriate solving engine. However, while a bit-vector encoding for basic UML/OCL constructs has already been introduced, no encoding for non-trivial OCL data types and operations is available so far. In this paper, we close this gap and present a bit-vector encoding for more complex OCL data types, i.e. sets, bags, and their ordered counterparts. As a result, SAT-based UML/OCL verification becomes applicable for models containing these collections types. A case study illustrates the application of this encoding.

1 Introduction

The *Unified Modeling Language* (UML) [1] is a de-facto standard in the domain of software development. Besides that, in recent years UML is also being employed for the specification of hardware systems [2] as it is a promising abstraction level enabling the modeling of a complex system while hiding concrete implementation details. Within UML, the *Object Constraint Language* (OCL) enables the enrichment of object-oriented models by textual constraints which add vital information. Using OCL, it is possible to restrict valid system states by invariants or to control the applicability of operation calls by pre- and post-conditions.

However, adding too restrictive OCL constraints leads to an *inconsistent* model, i.e. a model from which no valid system state can be constructed. Consistency and other verification tasks refer to the static aspects of a UML model. Further, wrong pre- and post-conditions can cause that an operation ω is not reachable, i.e. no system state can be constructed due to calls of other operations such that the pre-conditions of ω are satisfied. This property is called *reachability* and refers along with other verification tasks to the dynamic aspects of a UML model.

Accordingly, several approaches to (semi-)automatically solve these verification tasks have been proposed in the last years. For this purpose, different solving methodologies and engines are applied ranging from (1) interactive theorem provers [3,4] which require manual interactions over (2) enumeration techniques as e.g. provided in the *UML Specification Environment* (USE) [5] to (3) automatic approaches based on *Constraint Programming* (CSP) [6,7] or specification languages such as Alloy [8]. However, these approaches suffer either from the need for manual interaction, their enumerative behavior resulting in low scalability, or complicated transformations and solving steps.

In order to tackle these drawbacks, an alternative automatic UML/OCL verification method based on *Boolean Satisfiability* (SAT) and *SAT Modulo Theories* (SMT) has recently been suggested in [9,10]. Here, the actual verification task is directly encoded as an equivalent bit-vector instance which, afterwards, is solved by an appropriate solving engine. The impressive improvements of SAT and SMT solvers achieved in the past years are exploited enabling the treatment of static verification problems for significantly larger UML/OCL instances. Furthermore, also dynamic issues (e.g. reachability of function calls) are addressed by this method.

However, while the work in [9,10] provides initial implementations and first experimental results, the description on how to encode the respective UML/OCL components into a proper bit-vector formulation was limited to basic data types and operations, respectively. In particular, non-trivial OCL constructs such as collection types and operations using them have not been introduced so far. But, this is essential in order to provide an efficient solution for UML/OCL verification tasks with full support of the modeling language.

In this paper, we cover this missing link. More precisely, we show how OCL constraints can be encoded as a bit-vector instance in order to apply them to the previously introduced SAT/SMT-based verification of UML/OCL models. Besides basic data types, we also consider more complex constructs, i.e. sets, bags, and their ordered counterparts in detail. A case study illustrates the applicability of the proposed encoding by means of a practical example.

The remainder of this paper is structured as follows. Preliminaries on OCL, bit-vector logic, and the satisfiability problem are given in the next section. Afterwards, the background on UML/OCL verification is briefly reviewed in Sect. 3 leading to the main motivation for the contribution of this paper. Section 4 eventually introduces the respective encodings of the OCL data types into an equivalent bit-vector formulation. Afterwards, the applicability of this encoding is illustrated in Sect. 5 before the paper is concluded in Sect. 6.

2 Preliminaries

To keep the paper self contained, preliminaries on OCL, bit-vector logic, and the satisfiability problem are briefly reviewed in the following.

2.1 Object Constraint Language

In UML models, the set of valid system states can be restricted by UML constraints, i.e. associations between classes. Multiplicities annotated at the

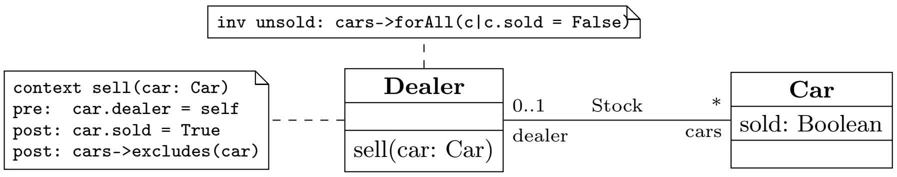


Fig. 1. Simple UML diagram enriched with OCL expressions

association ends define how the connected classes are related to each other. Further constraints, that e.g. restrict the attributes of the connected classes, cannot be expressed using UML. For this purpose, the *Object Constraint Language* (OCL) [11] has been employed.

OCL enables to extend UML diagrams with textual constraints which further restrict the set of valid system states. OCL constrains are primarily used to complete class diagrams, interaction diagrams, and state charts, but can also be applied to activity diagrams, component diagrams, and use cases. In this work, we focus on the application of OCL in UML class diagrams. However, the proposed techniques can be transferred to applications in other diagram types as well.

OCL is commonly applied to constrain different components (e.g. classes, attributes) in a model. Thus, they are modeled as an expression which evaluates to a Boolean value, i.e. *True* or *False*. In a UML class diagram, OCL expressions appear as both invariants as well as pre- and post-conditions (associated with operations). Invariants restrict the relations between classes and values of attributes, whereas pre- and post-conditions specify in which conditions an operation can be called and how the system state is specified afterwards.

Example 1. Figure 1 shows a UML class diagram for a simple car dealing model. Through the association *Stock*, a dealer can contain several cars, and a car can be assigned to a dealer or not. A car can be sold by a dealer using the *sell* method. The state whether a car is sold or not is stored in the attribute *sold* of class *Car*. One invariant *unsold* is attached to the class *Dealer* ensuring that all cars in stock must not be sold. Furthermore, the operation *sell* is annotated by pre- and post-conditions. In order to sell a car, the car must be assigned to the dealer. After a car is sold, the respective attribute must be updated and the car has to be removed from the stock.

2.2 Bit-Vector Logic

This paper aims to encode OCL expressions into equivalent bit-vector expressions. The definition of bit-vectors, their notation, and the applicable operations are briefly reviewed in the following.

Given the set of truth values $\mathbb{B} := \{0, 1\}$, the set $\mathbb{B}^n$ is referred to as *bit-vectors* of *size* or *dimension* n. Let $\boldsymbol{b} \in \mathbb{B}^n$ with $\boldsymbol{b} = (b_{n-1} \dots b_1 b_0)$ be a bit-vector. Then the i^{th} component of $\boldsymbol{b}$ is $\boldsymbol{b}[i] := b_i$. This *index* operation $\boldsymbol{b}[i]$

is a shorthand notation for $\boldsymbol{b}[i : 1]$ and the *extraction* operation is defined as $\boldsymbol{b}[i : l] := (b_{i+l-1} \dots b_i)$. Thus, $\boldsymbol{b}[i : l]$ is a bit-vector of dimension l starting from the bit b_i.

The counterpart to extraction is the *concatenation*. Given two bit-vectors $\boldsymbol{b}, \boldsymbol{c} \in \mathbb{B}^n$ with $\boldsymbol{b} = (b_{n-1} \dots b_0)$ and $\boldsymbol{c} = (c_{n-1} \dots c_0)$, the concatenation $\circ$ is defined as $\boldsymbol{b} \circ \boldsymbol{c} := (b_{n-1} \dots b_0 c_{n-1} \dots c_0)$.

The bit-vectors are big-endian to emphasize the correspondence to natural numbers, since bit-vectors represent the binary expansion of positive numbers, e.g. $1100_2 = 12_{10}$. More formally, to obtain a natural number from a bit-vector, the function $\text{nat} : \mathbb{B}^n \to [0, 2^n - 1]$ is defined as

$$\text{nat} : \boldsymbol{b} \mapsto \sum_{i=0}^{n-1} \boldsymbol{b}[i] \cdot 2^i \quad . \tag{1}$$

The inverse function of nat is $\text{bv} := \text{nat}^{-1}$ which returns the binary expansion of a natural number.

Further, there are logical and arithmetic operations which can be applied to bit-vectors. Bit-wise logical operations are amongst others $=, \wedge, \vee, \oplus$ referring to *equivalence*, *conjunction*, *disjunction*, and *exclusive disjunction (EXOR)*, respectively. Analogously, arithmetic operations are amongst others $\cdot, +, -$ referring to *multiplication*, *addition*, and *subtraction*, respectively. All arithmetic operations map into the same domain, that is e.g. a multiplication gets two n-bit bit-vectors as input and returns a n-bit bit-vector resulting value. Since obviously the result of a multiplication requires up to $2n$-bit, the operations follow an overflow arithmetic in the general case. However, in special cases a saturation arithmetic [12] can be applied, denoted by $\hat{\cdot}, \hat{+}, \hat{-}$. In this case, the maximal or minimal possible number which can be represented is taken in case of an overflow or underflow, respectively. For example, $1100_2 + 1100_2 = 1000_2$, but $1100_2 \hat{+} 1100_2 = 1111_2$.

2.3 Satisfiability Problem

The *Boolean satisfiability problem* (SAT) is defined as the task to determine a (satisfying) assignment to all inputs of a function f so that f evaluates to 1 or to prove that no such assignment exists. More formally:

Given a function $f : \mathbb{B}^n \to \mathbb{B}$, the function f is *satisfiable*, if and only if there exists an *assignment* $\alpha \in \mathbb{B}^n$ such that $f(\alpha) = 1$. In this case, α is called a *satisfying assignment*. Otherwise, f is *unsatisfiable*. Usually, the Boolean satisfiability check is conducted on a function in conjunctive normal form.

Although the SAT problem is $\mathcal{NP}$-complete [13], much research was dedicated to the investigation of SAT solvers in the recent decades (see e.g. [14,15,16]). Thus, many hard instances of practical problems can be transformed into SAT problems and, afterwards, solved quite efficiently [17].

To further enhance the efficiency of satisfiability solvers, researchers combined SAT techniques with solving strategies for higher levels of abstraction, e.g. arithmetic or bit-vector logic. This resulted in the development of solving methods for *SAT Modulo Theories* (SMT) [18,19]. Here, instead of a Boolean conjunctive normal form, instances may include more complex expressions, e.g. composed of bit-vector variables and operations as introduced above. In [20], it has been demonstrated that problems having a more complex structure tend to be solved more efficiently when retaining the level of abstraction in the solving process.

The common form of a satisfiability problem is a conjunction, in which several constraints to be satisfied are defined. SMT solvers exploit this structure by applying Boolean SAT solvers to handle the respective conjunction and specialized theory solvers to handle the single constraints provided in a higher abstraction. Besides that, also bit-blasting techniques (see e.g. [21]) are common to solve SMT instances.

The theory of quantifier-free bit-vectors (QF_BV) is an established higher abstraction which corresponds to the bit-vectors described in the previous section. The resulting problems can be formulated in the SMT-LIB file format that can be processed by off-the-shelf SMT solvers.

Example 2. Given three bit-vectors $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{B}^4$, consider the function $f : \mathbb{B}^{12} \to \mathbb{B}$ with

$$f(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}) = (\boldsymbol{a} = \boldsymbol{b} + \mathrm{bv}(2)) \wedge (\boldsymbol{b} = \boldsymbol{c} \cdot \boldsymbol{a}) \quad . \tag{2}$$

This bit-vector formula can be rewritten as an SMT instance using the bit-vectors theory as:

```
(benchmark f
  :logic QF_BV
  :extrafuns ((a BitVec[4]) (b BitVec[4]) (c BitVec[4]))

  :assumption (= a (bvadd b bv2[4]))
  :assumption (= b (bvmul c a))
)
```

When solving this instance with an SMT solver, e.g. the satisfying assignment

$$\alpha = (\boldsymbol{a} = 0011_2, \boldsymbol{b} = 0001_2, \boldsymbol{c} = 1011_2) \tag{3}$$

is returned.

3 Problem Formulation

The design of complex systems is a non-trivial task. To ease the design process, UML provides several description models that enable to explicitly specify the system to be realized, while, at the same time, hiding concrete implementation details. Properties of the design can additionally be specified using OCL. However, even on this higher abstraction, errors frequently arise leading e.g. to (1) an over-constraint model from which no valid system state can be derived or (2) to operations which can never be executed due to too restrictive pre- or post-conditions. Thus, verification approaches are applied to check the correctness

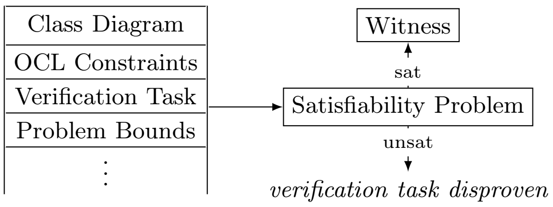


Fig. 2. General flow for solving static verification tasks

of a model. Typical verification tasks include checks for consistency (in a static view) or reachability (in a dynamic view).

In order to solve these verification problems, methods based on SAT/SMT have been shown to be quite promising [9,10]. Here, the general flow as depicted in Fig. 2 is applied. The given UML model together with the OCL constraints, the verification task, and further information (e.g. the number of objects to instantiate or the number of operation calls to consider) is taken and encoded as a satisfiability instance. Depending on the addressed verification task, the resulting instance becomes satisfiable if, e.g. in case of a consistency check, a valid system state exists. In contrast, if e.g. reachability is considered, the respective instance becomes satisfiable if a sequence diagram exists confirming that an operation can be executed. These solutions are called *witnesses* since they are witnessing the satisfiability of the considered verification task.

For this purpose, the number of objects to consider is restricted. While this is an essential requirement for the verification of UML/OCL models, this restriction is also justified by the fact that, eventually, the implemented system will be composed of a finite set of objects anyway. Further, the *small scope hypothesis* [22] supports the consideration of finite domains by stating that a large percentage of bugs can already be found by considering small state spaces.

In order to solve the created satisfiability instance, common SAT or SMT solvers are applied [16,21]. If the instance is satisfiable, the corresponding witness can be derived from the satisfying assignment to the variables. Otherwise, it has been proven that no such witness exists within the selected problem bounds.

The concrete encoding of the UML model and the OCL constraints into a proper bit-vector formulation is thereby crucial. So far, only encodings for basic data types and basic operations have been introduced. Non-trivial OCL constructs such as collection types and respective operations require more sophisticated encodings which are not available so far. Since this kind of OCL data types frequently occurs in UML/OCL models, we cover this missing link in this paper. That is, we address the following question:

How can we encode OCL data types and their respective operations in bit-vector logic so that they can be applied in SAT/SMT-based UML/OCL verification?

4 Encoding of OCL Data Types

This section briefly reviews the encoding of basic OCL data types into a bit-vector formulation. Based on that, the encodings of OCL sets are introduced and extended for further collection types.

4.1 Basic Data Types

Already for trivial OCL data types such as Boolean variables, a special bit-vector encoding is needed. This is because, although a Boolean variable can only be set to *True* and *False*, a third value, namely $\perp$ (undefined), has to be considered when checking for UML/OCL consistency. Accordingly, to encode such a variable in bit-vector logic at least two bits are required. More formally:

Encoding 1 (Boolean). An OCL Boolean variable b is encoded by a bit-vector $\boldsymbol{b} \in \mathbb{B}^2$. The truth values *True* and *False* are represented by the bit-vector values 00_2 and 01_2, respectively, whereas $\perp$ is encoded as 10_2. The remaining possible value 11_2 has to be prohibited in the satisfiability instance by adding the bit-vector constraint $(\boldsymbol{b} \neq 11_2)$.

The encoding of integer values is based on the same principle. However, there is another issue to consider: The domain of integer values in OCL is infinite. But in order to solve verification tasks using bit-vector logic, fixed sizes have to be assumed. While, at a first glance, this might look like an illegal simplification, it becomes reasonable considering that, at least for the concrete implementation of the considered UML/OCL model, finite bounds are applied nevertheless. Accordingly, integers are encoded as follows:

Encoding 2 (Integer). An OCL integer variable n is encoded by a bit-vector $\boldsymbol{n} \in \mathbb{B}^l$, where l is the precision assumed for integers when encoding the considered problem. Thus, $\boldsymbol{n}$ is suitable to encode l-bit integer values (except one). That is, if n is set to a value $v \in [0, 2^l - 2]$, then $\boldsymbol{n} = (n_{l-1} \dots n_0)$ such that $\boldsymbol{n}$ is the binary expansion of n, i.e. $\boldsymbol{n} = \text{bv}(n)$. The value $\perp$ is encoded by the remaining bit-vector value $\text{bv}(2^l - 1) = 1 \dots 1_2$.

Note that this encoding does allow the representation of all l-bit integer except $2^l - 1$. If the value $2^l - 1$ is essential in order to check an UML/OCL model, the value of l has to be increased by 1. Then, $2^l - 1$ can be represented. If necessary, all other $l + 1$-bit values can be prohibited in the satisfiability instance, e.g. by adding the bit-vector constraint $(\boldsymbol{n} = 1 \dots 1_2) \vee (\boldsymbol{n} < 10 \dots 0_2)$. Furthermore, note that negative values cannot be represented by this encoding. However, negative values can be enabled by substituting the binary expansion with the two-complement expansion.

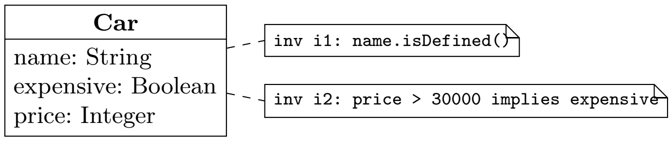
Another basic data type are strings. In a straight-forward view, each string can be seen as fixed length sequence of characters with a terminating character such as `char` arrays in the C programming language. Given an l-bit character encoding and strings of maximal length n (including the terminating character), a bit-vector of size $l \cdot n$ is required to encode the string. For ASCII strings of size 80 already $8 \cdot 80 = 640$ bits are required for each string variable. However, these bits are only required if the exact content of the string matters in the OCL constraints. This is the case, when OCL expressions such as `length` or `startsWith` are applied. If no such expressions are used, it only matters to distinguish different string values. Then, the content of the strings can be abstracted and a more efficient encoding can be applied.

Encoding 3 (String). Given a system state consisting of n string variables. Then, each string variable s is encoded by a bit-vector $\boldsymbol{s} \in \mathbb{B}^{\lceil \mathrm{ld}(n+1) \rceil}$ with $\mathrm{ld} := \log_2$. With this encoding, each variable can be set to a unique value (including $\perp$) to distinguish them. Therewith, indeed the exact content of the string cannot be determined, but operations like comparison of two string variables can be encoded.

The remaining basic data type is a real number. It has been observed that SAT instances including this kind of variables often are hard to solve [23] and, thus, should be avoided. Fortunately, the considered problems to be verified usually do not utilize real numbers since they are difficult to realize both in hardware and in software. However, if real values are needed nevertheless, abstractions such as fixed point or floating point numbers can be used. This can be encoded using bit-vectors of appropriate size.

Using these encodings of basic data types many "standard" operations of OCL constraints like logical or arithmetical expressions can already be encoded into bit-vector logic. Therefore, existing bit-vector constraints (as the one briefly sketched in Sect. 2.3) can be applied, but need to be extended to support the additional value $\perp$.

Example 3. Figure 3(a) shows a UML model with OCL invariants over basic data types, i.e. a string, a Boolean value, and a 32-bit integer. Considering a system state with three objects of class *Car*, an excerpt of the corresponding bit-vector encoding is depicted in Fig. 3(b). In Fig. 3(b), c denotes the car object being considered. Thus, these bit-vector expressions are repeated three times for each object.



(a) UML diagram consisting of basic OCL data types

$$\begin{array}{ll}
\boldsymbol{\alpha}^{c}_{\text{name}} \in \mathbb{B}^{2} & \texttt{i1} : \boldsymbol{\alpha}^{c}_{\text{name}} \neq 11_2 \\
\boldsymbol{\alpha}^{c}_{\text{expensive}} \in \mathbb{B}^{2} & \texttt{i2} : \left(\boldsymbol{\alpha}^{c}_{\text{price}} > \mathrm{bv}(30000)\right) \Rightarrow \left(\boldsymbol{\alpha}^{c}_{\text{expensive}} = 01_2\right) \\
\boldsymbol{\alpha}^{c}_{\text{price}} \in \mathbb{B}^{32} &
\end{array}$$

(b) Encoding of the OCL invariants

Fig. 3. OCL encoding of basic data types and operations

4.2 Sets

While the basic principle of encoding OCL data types into bit-vector logic has been illustrated in the previous section, the encoding of more complex OCL data types is introduced in the following by means of the *Set* container. Three scenarios of the car dealing example depicted in Fig. 1 are considered to illustrate the idea, namely

1. to address all objects of class *Car* connected to an object of class *Dealer* via the *Stock* association, i.e. by `cars`,
2. to state that all cars in the stock must not be sold (see the `forAll` operation in the invariant `unsold`), and
3. to state that a car is not included in the stock after it has been sold (see the `excludes` operation in the post-condition of the *sell* operation).

As already discussed above, the bit-vectors that encode OCL constraints must be of a fixed size in order to be suitable for SAT/SMT-based verification. However, the cardinalities of the sets within a UML model and within the respective system states, respectively, may be of dynamic size. Thus, the total number of objects in the system state is incorporated in the bit-vector encoding. More precisely:

Encoding 4 (Set). Let A be a UML class and $<$ be a total order on the objects of class A, i.e. $a_0 < \cdots < a_{|A|-1}$ where $a_0, \ldots, a_{|A|-1}$ are objects derived from class A and $|A|$ denotes the total number of these objects. Then, each OCL variable `v:Set(`A`)` is encoded by a bit-vector $\boldsymbol{v} \in \mathbb{B}^{|A|}$ with $\boldsymbol{v} = \left(v_{|A|-1} \ldots v_0\right)$, such that $v_i = 1$ if and only if `v.includes(`a_i`)`.

Example 4. Let $D = \{d_0, \ldots, d_{m-1}\}$ be the set of all objects of class *Dealer* and let $C = \{c_0, \ldots, c_{n-1}\}$ be the set of all objects of class *Car*, respectively. Using this encoding, all three scenarios mentioned above can be encoded into a bit-vector instance as follows:

1. The set `cars` of objects derived from class *Car* that are associated to class *Dealer* is represented by one bit-vector $\boldsymbol{\lambda}^d_{\text{cars}} \in \mathbb{B}^n$ for each object $d \in D$. In accordance with Enc. 4, d is linked to an object c_i, if the corresponding bit in $\boldsymbol{\lambda}^d_{\text{cars}}$ is set to 1.
2. The invariant `unsold` in Fig. 1 constrains that the `sold` attribute for each car associated to a dealer should be *False*. Although the size of the set `cars` is dynamic, the size of the corresponding bit-vector $\boldsymbol{\lambda}^d_{\text{cars}}$ is fixed. The invariant is modeled as a bit-vector expression as follows:

$$\bigwedge_{i=0}^{n} \boldsymbol{\lambda}^d_{\text{cars}}[i] \Rightarrow (\boldsymbol{\alpha}^{c_i}_{\text{sold}} = 00_2) \tag{4}$$

Thus, each bit in the bit-vector representing all possible elements in the set is considered. Only for those elements in the set, the invariant condition is forced. This is done by implication.

Table 1. OCL collection types

Type	Description	Example
Set	Each element can occur at most once	Set (b_1, b_5, b_3) = Set (b_1, b_3, b_5)
OrderedSet	As set, but ordered	OrderedSet $(b_1, b_5, b_3) \neq$ OrderedSet (b_1, b_3, b_5)
Bag	Elements may be presence more than once	Bag (b_1, b_3, b_3) = Bag (b_3, b_1, b_3)
Sequence	As bag, but ordered	Sequence $(b_1, b_3, b_3) \neq$ Sequence (b_3, b_1, b_3)

3. To state that the car being sold is not in the stock anymore (as constrained in the post-condition in Fig. 1), the corresponding bit in the bit-vector must be set to 0. Let c_i be the parameter of the operation, then $\neg\boldsymbol{\lambda}^{d}_{\text{cars}}[i]$ encodes the post-condition.

4.3 Further Collection Types

Based on the encoding of the *Set* data type, encodings for the remaining OCL collection types, namely *OrderedSet*, *Bag*, and *Sequence*, are introduced in this section. The differences of these data types are as follows: In a *Set*, each element can occur at most once, whereas in a *Bag* each element may occur more than once. For both, sets and bags, counterparts exists in which the elements follow an order, i.e. *OrderedSet* and *Sequence*, respectively. Table 1 briefly summarizes the semantics of all these data types. Note that an ordered set and a sequence are ordered, but not sorted. That is, successive elements are not greater or less than the element before (see column *Example* in Table 1). Before outlining the encoding for ordered collections, the transformation of bags into bit-vectors is described first.

Encoding of Bags. What makes a bag different from a set is the property that elements can occur more than once. The idea of encoding a bag is similar to the one of encoding a set. The difference is that the bits in the encoding of a set represent whether an element is contained or not. For bags, each bit is replaced by a cardinality number. More formally:

Encoding 5 (Bag). Let A be a UML class and $<$ be a total order on the objects of class A, i.e. $a_0 < \cdots < a_{|A|-1}$ where $\{a_0, \ldots, a_{|A|-1}\}$ are objects derived from class A. Furthermore, it is assumed that each object occurs at most 2^m times in a bag. Then, each OCL variable `v:Bag(`A`)` is encoded by a bit-vector $\boldsymbol{v} \in \mathbb{B}^{m \cdot |A|}$ with $\boldsymbol{v} = (\boldsymbol{v_{|A|-1}} \ldots \boldsymbol{v_0})$, such that $\text{nat}(\boldsymbol{v_i}) =$ `v.count(`a_i`)`.

The number of occurrences of objects in a bag (i.e. the respective cardinality) is thereby crucial. For sets, the total number of objects can be used as an upper bound. This is not possible for bags, since here an arbitrary number of equal objects may be contained. Thus, a reasonable upper bound of possible objects has to be defined. Similar to the encoding of integer values, this is a simplification which, however, becomes reasonable considering that at least for the concrete implementation finite bounds are applied nevertheless.

Encoding of Ordered Sets. To encode an ordered set in bit-vector logic, the position of the elements needs to be incorporated. This can be done as follows:

Encoding 6 (Ordered Set). Let A be a UML class with a total order $<$ and a set of derived objects $\{a_0, \ldots, a_{|A|-1}\}$. Then, an ordered set `v:OrderedSet(`A`)` is encoded by a bit-vector $\boldsymbol{v} \in \mathbb{B}^{|A| \cdot l}$ with $l = \lceil \mathrm{ld}(|A| + 1) \rceil$. For each element ($|A|$ times), l bits are devoted to encode $|A| + 1$ different values, i.e. the values $0, \ldots, |A| - 1$ specify positions of the elements and $2^l - 1 = 11\ldots1_2$ expresses that an element is not in the ordered set.

Furthermore, the following three constraints have to be added to the satisfiability instance in order to keep the semantics of the ordered set consistent:

1. There can be at most one element at each position, i.e.

$$\bigwedge_{i=0}^{|A|-1} \bigwedge_{j=0}^{|A|-1} \bigwedge_{\substack{k=0\\k\neq i}}^{|A|-1} (\boldsymbol{v}[il:l] = \mathrm{bv}(j)) \Rightarrow (\boldsymbol{v}[kl:l] \neq \mathrm{bv}(j)) \quad . \tag{5}$$

2. If an element is encoded to be at the j^{th} position (with $j > 0$), then there must be some element at position $j - 1$, i.e.

$$\bigwedge_{i=0}^{|A|-1} \bigwedge_{j=1}^{|A|-1} (\boldsymbol{v}[i:il] = \mathrm{bv}(j)) \Rightarrow \bigvee_{\substack{k=0\\k\neq i}}^{|A|-1} \boldsymbol{v}[kl:l] = \mathrm{bv}(j-1) \quad . \tag{6}$$

3. Since l bits can possibly encode more than $|A|+1$ values, illegal assignments must be prohibited, i.e.

$$\bigwedge_{i=0}^{|A|-1} \big(\boldsymbol{v}[il:l] < \mathrm{bv}(|A|) \vee \boldsymbol{v}[il:l] = \mathrm{bv}(2^l - 1)\big) \quad . \tag{7}$$

Encoding of Sequences. Sequences are the most expensive data type to encode. Using the same argumentation used within the encoding of bags, the number of elements appearing in a sequence is not limited by the system state. Thus, again a reasonable upper bound has to be determined before encoding the satisfiability instance.

Encoding 7 (Sequence). Let A be a UML class with a total order $<$ and a set of derived objects $\{a_0, \ldots, a_{|A|-1}\}$. Then, a sequence `v:Sequence(`$A$`)` is encoded by a bit-vector $\boldsymbol{v} \in \mathbb{B}^{(2^m \cdot |A| \cdot l)}$ with $l = \lceil \mathrm{ld}(2^m \cdot |A| + 1) \rceil$. Otherwise, the same semantics as for ordered sets apply, however, for sequences $2^m \cdot |A|$ possible positions have to be encoded and not just $|A|$, since each element can occur up to 2^m times (cf. Enc. 5).

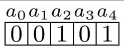
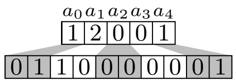
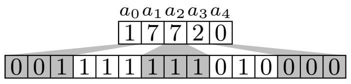
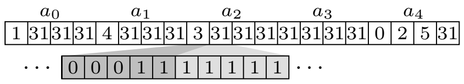
OCL Collection	**Bit-Vector**
`v1=Set{`a_2`,`a_4`}` $\boldsymbol{v_1} \in \mathbb{B}^{\lvert A\rvert} \rightsquigarrow \mathbb{B}^5$	
`v2=Bag{`a_0`,`a_1`,`a_1`,`a_4`}` $\boldsymbol{v_2} \in \mathbb{B}^{m\cdot\lvert A\rvert} \overset{m=2}{\rightsquigarrow} \mathbb{B}^{10}$	 $a_0\,a_1\,a_2\,a_3\,a_4$ 1 2 0 0 1 0 1 1 0 0 0 0 0 0 1
`v3=OrderedSet{`a_4`,`a_0`,`a_3`}` $\boldsymbol{v_3} \in \mathbb{B}^{\lvert A\rvert\cdot l} \overset{l=3}{\rightsquigarrow} \mathbb{B}^{15}$	 $a_0\,a_1\,a_2\,a_3\,a_4$ 1 7 7 2 0 0 0 1 1 1 1 1 1 1 0 1 0 0 0 0
`v4=Sequence{`a_4`,`a_0`,`a_4`,`a_2`,`a_1`,`a_4`}` $\boldsymbol{v_4} \in \mathbb{B}^{(2^m\cdot\lvert A\rvert\cdot l)} \overset{l=5}{\rightsquigarrow} \mathbb{B}^{100}$	 a_0 a_1 a_2 a_3 a_4 1 31 31 31 4 31 31 31 3 31 31 31 31 31 31 31 0 2 5 31 ··· 0 0 0 1 1 1 1 1 1 1 ···

Fig. 4. Overview of encodings for OCL collection data types

Example 5. Figure 4 illustrates all encodings applied to a base collection $A = \{a_0, a_1, a_2, a_3, a_4\}$. For the bag and sequence, the cardinality of elements is set to 4, i.e. $m = 2$. The value of l is determined according to the maximal number of elements in the respective collection. Thus, for an ordered set this is $l = \lceil \mathrm{ld}(|A|+1)\rceil = \lceil \mathrm{ld}\,6\rceil = 3$, and for a sequence it is $l = \lceil \mathrm{ld}(2^m \cdot |A| + 1)\rceil = \lceil \mathrm{ld}\,20\rceil = 5$, respectively.

In case of the bag, a_0 is contained once and a_1 is contained twice. Thus, the respective fields in the bit-vector are 01_2 for a_0 and 10_2 for a_1, respectively.

4.4 Operations on Collection Types

Having the encodings of the collection data types available, they can be used to encode the respective operations on them. Example 4 already illustrated the encoding of the `excludes` operation. In a similar way, this can be done for the remaining operations as well.

In fact, many of the OCL operations can be mapped to a corresponding bit-vector counterpart. To illustrate this, consider the encoding of a set. The elements in both, the set as well as the corresponding bit-vector encoding, are supposed to follow a total order. That is, each element in the set corresponds to a fixed bit in the bit-vector. Because of this, the set-operations `union` and `intersection` can be mapped to the bit-wise disjunction and bit-wise conjunction, respectively.

Analogously, this can be done for the remaining set-operations. This is summarized in detail in Table 2 which lists all set-operations together with the respective encoding for a class A with objects $\{a_0, \ldots, a_{n-1}\}$ and sets `v1:Set(`A`)` as well as `v2:Set(`A`)`[1]. Note that the operations `asBag`, `asOrderedSet` and

[1] For simplicity we omitted exceptional cases in the encodings such as the treatment of undefined collections. However, they can easily be supported by adding case differentiation to the bit-vector expressions.

Table 2. Encoding of set operations into bit-vector operations

Operation	Encoding
`v1 = v2`	$\boldsymbol{v_1} = \boldsymbol{v_2}$
`v1 <> v2`	$\boldsymbol{v_1} \neq \boldsymbol{v_2}$
`v3 = v1->asBag()`	$\boldsymbol{v_3} \in \mathbb{B}^{m \cdot n}$ s.t. $\boldsymbol{v_3}[j] = \begin{cases} \boldsymbol{v_1}[i] & \text{if } j = im, \\ 0 & \text{otherwise.} \end{cases}$
`v3 = v1->asOrderedSet()`	$\boldsymbol{v_3} \in \mathbb{B}^{n \cdot l}$ s.t. $\boldsymbol{v_3}[il : l] = \begin{cases} \mathrm{bv}\left(\sum_{j=0}^{i-1} \boldsymbol{v}_1[j]\right) & \text{if } \boldsymbol{v}_1[i] = 1, \\ \mathrm{bv}(2^l - 1) & \text{otherwise.} \end{cases}$ with $l = \lceil \mathrm{ld}(n+1) \rceil$
`v3 = v1->asSequence()`	see `v1->asBag()->asSequence()`
`v1->count(`a_i`)`	$\boldsymbol{v_1}[i]$
`v1->excludes(`a_i`)`	$\neg \boldsymbol{v_1}[i]$
`v1->excludesAll(v2)`	$\neg \boldsymbol{v_1} \wedge \boldsymbol{v_2} = \boldsymbol{v_2}$
`v1->excluding(`a_i`)`	$\boldsymbol{v_1} \wedge \neg\, \mathrm{bv}(2^i)$
`v1->includes(`a_i`)`	$\boldsymbol{v_1}[i]$
`v1->includesAll(v2)`	$\boldsymbol{v_1} \wedge \boldsymbol{v_2} = \boldsymbol{v_2}$
`v1->including(`a_i`)`	$\boldsymbol{v_1} \vee \mathrm{bv}(2^i)$
`v1->intersection(v2)`	$\boldsymbol{v_1} \wedge \boldsymbol{v_2}$
`v1->isEmpty()`	$\boldsymbol{v_1} = \mathrm{bv}(0)$
`v1->notEmpty()`	$\boldsymbol{v_1} \neq \mathrm{bv}(0)$
`v1->size()`	$\sum_{i=0}^{n-1} \boldsymbol{v_1}[i]$
`v1->symmetricDifference(v2)`	$\boldsymbol{v_1} \oplus \boldsymbol{v_2}$
`v1->union(v2)`	$\boldsymbol{v_1} \vee \boldsymbol{v_2}$

`asSequence` require thereby auxiliary variables since the operation results in a different bit-vector domain.

Example 6. Consider the operation `v1->including(`a_i`)` which results in a set containing all elements of `v1` and the element a_i. This can be rewritten as `v1->union(Set{`a_i`})`. A set containing only the element a_i can be expressed as a bit-vector with only one bit set at position i, which corresponds to the natural number 2^i. Using the bit-wise disjunction to express the union of two sets, the operation results in $v_1 \vee \mathrm{bv}(2^i)$.

Accordingly, bit-vector expressions to model operations on bags are outlined in Table 3.

Example 7. Consider e.g. the `including` transformation applied to a bag. Instead of *activating* the i^{th} bit, first all bits are erased at position i, i.e.

$$\boldsymbol{v_1} \wedge \neg\, \mathrm{bv}\left(\sum_{k=il}^{im+m-1} 2^i \right), \tag{8}$$

before to the result of that expression

$$\mathrm{bv}\left((\mathrm{nat}(\boldsymbol{v_1}[im : m]) \hat{+} 1) \cdot 2^{im} \right) \tag{9}$$

is *added* by disjunction. That is, to the current amount of a_i, i.e. $\mathrm{nat}(\boldsymbol{v_1}[im : m])$, first 1 is added before shifting by im bits to the left so that they replace the current cardinality of a_i. Further, consider the expressions for `intersection` and `union`. Both bags are element-wise concatenated, whereby for the intersection the respective minimal amount of elements and for the union the sum of both amounts is used, respectively.

Table 3. Mappings of bag operations into bit-vector operations

Operation	Mapping
`v1 = v2`	$\boldsymbol{v_1} = \boldsymbol{v_2}$
`v1 <> v2`	$\boldsymbol{v_1} \neq \boldsymbol{v_2}$
`v3 = v1->asOrderedSet()`	see `v1->asSet()->asOrderedSet()`
`v3 = v1->asSequence()`	$\boldsymbol{v_3} \in \mathbb{B}^{2^m nl}$ s.t. $\forall_{i=0}^{n-1} \forall_{j=0}^{2^m-1}$: $\boldsymbol{v_3}[i2^m l + jl : l] = \begin{cases} \mathrm{bv}(j) + \sum_{k=0}^{i-1} \boldsymbol{v_1}[km : m] & \text{if } j < \mathrm{nat}(\boldsymbol{v_1}[im : m]), \\ \mathrm{bv}(2^l - 1) & \text{otherwise.} \end{cases}$
`v3 = v1->asSet()`	$\boldsymbol{v_3} \in \mathbb{B}^n$ s.t. $\boldsymbol{v_3}[i] = \begin{cases} 1 \text{ if } \boldsymbol{v_1}[im : m] \neq \mathrm{bv}(0), \\ 0 \text{ otherwise.} \end{cases}$
`v1->count(`a_i`)`	$\mathrm{nat}(\boldsymbol{v_1}[im : m])$
`v1->excludes(`a_i`)`	$\boldsymbol{v_1}[im : m] = \mathrm{bv}(0)$
`v1->excludesAll(v2)`	$\bigwedge_{i=0}^{n-1} (\boldsymbol{v_2}[im : m] \neq \mathrm{bv}(0)) \Rightarrow (\boldsymbol{v_1}[im : m] = \mathrm{bv}(0))$
`v1->excluding(`a_i`)`	$\left(\boldsymbol{v_1} \wedge \neg\, \mathrm{bv}\left(\sum_{k=im}^{im+m-1} 2^i\right)\right) \vee \mathrm{bv}\left(\mathrm{nat}(\boldsymbol{v_1}[im : m]\hat{-}1) \cdot 2^{im}\right)$
`v1->includes(`a_i`)`	$\boldsymbol{v_1}[im : m] \neq \mathrm{bv}(0)$
`v1->includesAll(v2)`	$\bigwedge_{i=0}^{n-1} (\boldsymbol{v_2}[im : m] \neq \mathrm{bv}(0)) \Rightarrow (\boldsymbol{v_1}[im : m] \neq \mathrm{bv}(0))$
`v1->including(`a_i`)`	$\left(\boldsymbol{v_1} \wedge \neg\, \mathrm{bv}\left(\sum_{k=im}^{im+m-1} 2^i\right)\right) \vee \mathrm{bv}\left((\mathrm{nat}(\boldsymbol{v_1}[im : m])\hat{+}1) \cdot 2^{im}\right)$
`v1->intersection(v2)`	$\bigcirc_{i=0}^{n-1} \min\{\mathrm{nat}(\boldsymbol{v_1}[im : m]), \mathrm{nat}(\boldsymbol{v_2}[im : m])\}$ ($\bigcirc$ *is concatenation)*
`v1->isEmpty()`	$\boldsymbol{v_1} = \mathrm{bv}(0)$
`v1->notEmpty()`	$\boldsymbol{v_1} \neq \mathrm{bv}(0)$
`v1->size()`	$\sum_{i=0}^{n-1} \mathrm{nat}(\boldsymbol{v_1}[im : m])$
`v1->union(v2)`	$\bigcirc_{i=0}^{n-1} \boldsymbol{v_1}[im : m] \hat{+} \boldsymbol{v_2}[im : m]$

The mappings for operations on ordered sets are given in Table 4 considering ordered sets with at most n elements and l as described in Enc. 6. The function maxpos is used in some operations and returns the largest index in the ordered set. The function is defined as

$$\mathrm{maxpos}(\boldsymbol{v}) := \max_{k=0}^{n-1} \left\{ \mathrm{nat}(\boldsymbol{v}[kl : l]) \mid \boldsymbol{v}[kl : l] \neq \mathrm{bv}(2^l - 1) \right\} \quad . \tag{10}$$

Note that in OCL, the first element in an ordered set has the index 1, while in the encoding the first index is 0 due to advantages in the implementation. The bit-vector expressions for the OCL operations on ordered sets is described by the means of two examples.

Example 8. Consider the operation `v1->at(`k`)` in Table 4. According to the encoding defined in Enc. 6, the bit-vector is subdivided into several fields, where each field corresponds to one item of all available items. The field contains an index describing either the position of that item in the ordered set or the value $\mathrm{bv}(2^l - 1)$ if the item is not contained in the ordered set. Thus, the field containing the required position k has to be found: For each position in the encoding, the content is compared to the index with $\boldsymbol{v_1}[il : l] = \mathrm{bv}(k - 1)$. This either evaluates to 0 or, in one case, to 1 assuming that `v1` contains at most k items. Multiplying the result with $\mathrm{bv}(2^l - 1)$, i.e. a bit-vector containing l ones, results in either a bit-vector containing only zeros or ones. This bit-vector is used as a bit-mask for the considered position, i.e. $\mathrm{bv}(i)$, and all these bit-vectors are added. Since only one bit-vector does not contain of all zeros, which is the bit-vector containing the item, the result is a bit-vector encoding the item at position k.

Table 4. Mappings of ordered set operations into bit-vector operations

Operation	Mapping
`v1 = v2`	$\boldsymbol{v_1} = \boldsymbol{v_2}$
`v1 <> v2`	$\boldsymbol{v_1} \neq \boldsymbol{v_2}$
`v1->append(`a_i`)`	$\boldsymbol{v_1}[jl:l] = \begin{cases} \text{bv}\,(\text{maxpos}(v_1)+1) & \text{if } j = i \wedge \boldsymbol{v_1}[jl:l] = \text{bv}(2^l-1), \\ \boldsymbol{v_1}[jl:l] & \text{otherwise.} \end{cases}$
`v3 = v1->asBag()`	see `v1->asSet()->asBag()`
`v3 = v1->asSequence()`	see `v1->asSet()->asBag()->asSequence()`
`v3 = v1->asSet()`	$\boldsymbol{v_3} \in \mathbb{B}^n$ s.t. $\boldsymbol{v_3}[i] = \begin{cases} 1 \text{ if } \boldsymbol{v_1}[il:l] \neq \text{bv}(2^l-1), \\ 0 \text{ otherwise.} \end{cases}$
`v1->at(`k`)`	$\sum_{i=0}^{n-1}\Big((\boldsymbol{v_1}[il:l] = \text{bv}(k-1)) \cdot \text{bv}(2^l-1)\Big) \wedge \text{bv}(i)$
`v1->count(`a_i`)`	$\boldsymbol{v_1}[il:l] \neq \text{bv}(2^l-1)$
`v1->excludes(`a_i`)`	$\boldsymbol{v_1}[il:l] = \text{bv}(2^l-1)$
`v1->excludesAll(v2)`	$\bigwedge_{i=0}^{n-1}\Big(\boldsymbol{v_2}[il:l] \neq \text{bv}(2^l-1)\Big) \Rightarrow \Big(\boldsymbol{v_1}[il:l] = \text{bv}(2^l-1)\Big)$
`v1->excluding(`a_i`)`	$\boldsymbol{v_1} \vee \text{bv}(2^l-1) \cdot 2^{il}$
`v1->first()`	$\bigwedge_{j=0}^{n-1}\Big((\boldsymbol{v_1}[jl:l] = \text{bv}(0)) \cdot \text{bv}(2^l-1)\Big) \wedge \text{bv}(j)$
`v1->includes(`a_i`)`	$\boldsymbol{v_1}[il:l] \neq \text{bv}(2^l-1)$
`v1->includesAll(v2)`	$\bigwedge_{i=0}^{n-1}\Big(\boldsymbol{v_2}[il:l] \neq \text{bv}(2^l-1)\Big) \Rightarrow \Big(\boldsymbol{v_1}[il:l] \neq \text{bv}(2^l-1)\Big)$
`v1->including(`a_i`)`	see `v1->append(`a_i`)`
`v1->indexOf(`a_i`)`	$\text{nat}(\boldsymbol{v_1}[il:l]) + 1$
`v1->insertAt(`k, a_i`)`	$\boldsymbol{v_1}[jl:l] = \begin{cases} \text{bv}(k-1) & \text{if } j = i \wedge \boldsymbol{v_1}[jl:l] = \text{bv}(2^l-1), \\ \boldsymbol{v_1}[jl:l] & \text{if } \boldsymbol{v_1}[jl:l] < \text{bv}(k) \vee \boldsymbol{v_1}[jl:l] = \text{bv}(2^l-1), \\ \boldsymbol{v_1}[jl:l] + \text{bv}(1) & \text{otherwise.} \end{cases}$
`v1->isEmpty()`	$\boldsymbol{v_1} = \text{bv}(2^{ln}-1)$
`v1->last()`	$\bigwedge_{j=0}^{n-1}\Big((\boldsymbol{v_1}[jl:l] = \text{bv}(\text{maxpos}(\boldsymbol{v_1})) \cdot \text{bv}(2^l-1)\Big) \wedge \text{bv}(j)$
`v1->notEmpty()`	$\boldsymbol{v_1} \neq \text{bv}(2^{ln}-1)$
`v1->prepend()`	$\boldsymbol{v_1}[jl:l] = \begin{cases} \text{bv}(0) & \text{if } j = i \wedge \boldsymbol{v_1}[jl:l] = \text{bv}(2^l-1), \\ \text{bv}(2^l-1) & \text{if } \boldsymbol{v_1}[jl:l] = \text{bv}(2^l-1), \\ \boldsymbol{v_1}[jl:l] + \text{bv}(1) & \text{otherwise.} \end{cases}$
`v1->size()`	$\text{bv}(\text{maxpos}(\boldsymbol{v_1})) + 1$

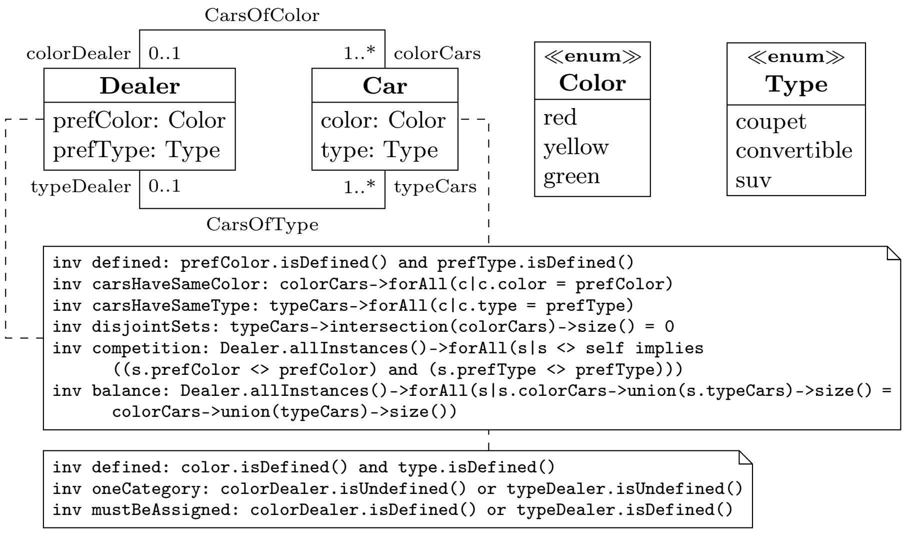
On the other hand, the operation `v1->indexOf(`a_i`)` is encoded straight forward. Since the field corresponding to a_i can be determined directly by $\boldsymbol{v_1}[il:l]$, the result is its natural representation incremented by 1.

We omitted the detailed table of bit-vector expressions for operations on sequences due to page limitations. However, they can be derived by *combining* the bit-vector expressions for the respective operations on bags and ordered sets.

5 Case Study

In this section, we illustrate the application of the proposed encoding by means of a case study. Therefore, the UML/OCL model depicted in Fig. 5 and representing a car dealing scenario is considered. A car dealer (*Dealer*) offers cars (*Car*) according to a preferred color and preferred type. The associations *CarsOfColor* and *CarsOfType* are used to model which cars belong to the dealer (distinguished with respect to the color and the type, respectively). A car dealer has at least one car by color and by type, and each car can only be assigned to one dealer.

In the following, selected OCL invariants for this model along with the resulting bit-vector encoding are outlined. The respective verification task is to generate a valid system state composed of three dealers, i.e. three objects $D =$



```
inv defined: prefColor.isDefined() and prefType.isDefined()
inv carsHaveSameColor: colorCars->forAll(c|c.color = prefColor)
inv carsHaveSameType: typeCars->forAll(c|c.type = prefType)
inv disjointSets: typeCars->intersection(colorCars)->size() = 0
inv competition: Dealer.allInstances()->forAll(s|s <> self implies
      ((s.prefColor <> prefColor) and (s.prefType <> prefType)))
inv balance: Dealer.allInstances()->forAll(s|s.colorCars->union(s.typeCars)->size() =
      colorCars->union(typeCars)->size())
```

```
inv defined: color.isDefined() and type.isDefined()
inv oneCategory: colorDealer.isUndefined() or typeDealer.isUndefined()
inv mustBeAssigned: colorDealer.isDefined() or typeDealer.isDefined()
```

Fig. 5. Car dealing example

$\{d_0, d_1, d_2\}$ derived from class *Dealer*, and 15 cars, i.e. objects $C = \{c_0, \ldots, c_{14}\}$ derived from class *Car*. To encode the attributes, we introduce the bit-vector variables $\boldsymbol{\alpha}^d_{\text{prefColor}} \in \mathbb{B}^2$ for each $d \in D$. Other bit-vectors are created accordingly for the other attributes of the class diagram. In the same manner as in Example 4, bit-vectors are created for the associations, i.e. $\boldsymbol{\lambda}^d_{\text{colorCars}} \in \mathbb{B}^{15}$ for each $d \in D$.

The `defined` invariants for both classes can be encoded according to Fig. 3. Next, the invariants `carsHaveSameColor` and `carsHaveSameType` ensure that cars who are in the stock of a dealer must meet the preferred color or type respective to the association. The respective bit-vector encoding for the invariant `carsHaveSameColor` reads as follows:

$$\forall d \in D \, : \, \bigwedge_{i=0}^{|C|-1} \left(\boldsymbol{\lambda}^d_{\text{colorCars}}[i] \Rightarrow \left(\boldsymbol{\alpha}^{c_i}_{\text{color}} = \boldsymbol{\alpha}^d_{\text{prefColor}}\right)\right) \tag{11}$$

The bit-vector encoding for the invariant `carsHaveSameType` is formulated analogously.

The invariant `disjointSets` assures that cars are either connected by their color or by their type, i.e. the intersection of *colorCars* and *typeCars* must be empty for each dealer. Using the encodings suggested in Table 2, the following bit-vector expression results:

$$\forall d \in D \, : \, \sum_{i=0}^{|C|-1} \left(\boldsymbol{\lambda}^d_{\text{typeCars}} \wedge \boldsymbol{\lambda}^d_{\text{colorCars}}\right)[i] = 0 \tag{12}$$

In this expression, the number of bits of the intersection (bit-wise conjunction) are counted and forced to be 0.

To ensure a variety of car dealers, the `competition` invariant is added to ensure that there are no dealers with the same preferred color or type. This is encoded as:

$$\forall d \in D : \bigwedge_{d' \in D} (d = d') \Rightarrow \left(\left(\boldsymbol{\alpha}^{d}_{\text{prefColor}} \neq \boldsymbol{\alpha}^{d'}_{\text{prefColor}}\right) \wedge \left(\boldsymbol{\alpha}^{d}_{\text{prefType}} \neq \boldsymbol{\alpha}^{d'}_{\text{prefType}}\right)\right) \tag{13}$$

The invariant `balance` ensures that all dealers have the same number of cars, regardless of whether by color or by type. Thus, the size of the unions of both sets are compared:

$$\forall d \in D : \bigwedge_{d' \in D} \left(\sum_{i=0}^{|C|-1} \left(\boldsymbol{\lambda}^{d}_{\text{colorCars}} \vee \boldsymbol{\lambda}^{d}_{\text{typeCars}}\right)[i] = \sum_{i=0}^{|C|-1} \left(\boldsymbol{\lambda}^{d'}_{\text{colorCars}} \vee \boldsymbol{\lambda}^{d'}_{\text{typeCars}}\right)[i] \right) \tag{14}$$

The invariant `disjointSets` assures that one car cannot be used both by color and by type for one dealer. However, using the invariants introduced so far, a car can still be assigned by color to one dealer and by type to another one. To prevent this, the invariant `oneCategory` is added to the *Car* class, stating that one of the association ends has to be undefined:

$$\forall c \in C \, : \, \boldsymbol{\lambda}^{c}_{\text{colorDealer}} = 11_2 \vee \boldsymbol{\lambda}^{c}_{\text{typeDealer}} = 11_2 \tag{15}$$

Note that in this case the $\boldsymbol{\lambda}$ variables are not interpreted as bit-vectors but as natural numbers directly pointing to the dealer object. This is done, since a car can only be assigned to at most one dealer. In this sense, the $\boldsymbol{\lambda}$ vectors can be interpreted analogously to the $\boldsymbol{\alpha}$ attribute vectors.

Finally, we want to assign each car to a dealer by car or by color. Analogously to the previous invariant, in this case it has to be assured that at least one association end is defined resulting in the `mustBeAssigned` invariant, encoded as:

$$\forall c \in C \, : \, \boldsymbol{\lambda}^{c}_{\text{colorDealer}} \neq 11_2 \vee \boldsymbol{\lambda}^{c}_{\text{typeDealer}} \neq 11_2 \tag{16}$$

Solving the resulting satisfiability instance with an SMT solver such as Boolector [21], a satisfying assignment is returned, amongst others assigning $\boldsymbol{\alpha}^{c_8}_{\text{color}} = 00_2$, $\boldsymbol{\alpha}^{c_{11}}_{\text{type}} = 10_2$, and $\boldsymbol{\lambda}^{d_0}_{\text{colorCars}} = 00010\,01000\,00000_2$. From the assignments, a system state can be constructed, e.g. *Car8* is assigned the color *red*, *Car11* is assigned the type *suv*, and *Dealer0* is connected via *CarsOfColor* to *Car8* and *Car11*. Together with other assignments, an object diagram representing the system state can be obtained. This is partially depicted in Fig. 6. Here, one dealer together with its connected cars and all attribute assignments is shown.

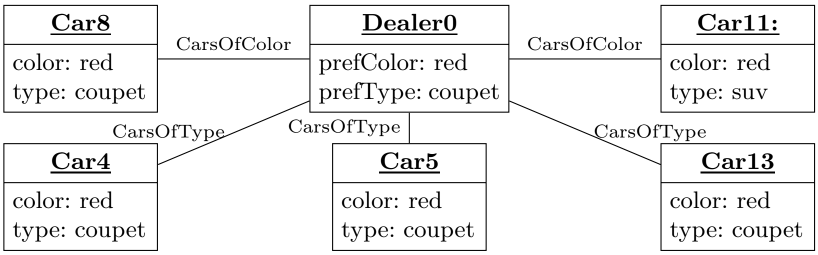


Fig. 6. Derived system state

Solving this particular problem with 15 cars the solver Boolector [21] requires less than 0.1 seconds to determine the solution on an Intel 2.26 GHz Core 2 Duo processor with 3 GB main memory. Scaling the example to determine a solution with 150 cars, the solver takes 6.9 seconds. Further experiments with run-times can be found in [9,10].

6 Conclusion

In this work, encodings for both OCL basic and collection data types have been presented. The encoding of these data types and their operations into bit-vector expressions enables their application in satisfiability based verification techniques proposed in the past. OCL and satisfiability instances follow different design paradigms. One example is the number of variables and their size, which is dynamic in OCL, whereas it must be defined initially with a static bit-width in a satisfiability instance. This leads to non trivial encodings for both, the data types and their operations. The applicability of the encodings has been demonstrated in a case study by means of a practical example.

References

1. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language reference manual. Addison-Wesley Longman, Essex (1999)
2. Vanderperren, Y., Müller, W., Dehaene, W.: UML for electronic systems design: a comprehensive overview. Design Automation for Embedded Systems 12(4), 261–292 (2008)
3. Kyas, M., Fecher, H., de Boer, F.S., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., Kugler, H.: Formalizing UML Models and OCL Constraints in PVS. Electronic Notes in Theoretical Computer Science 115, 39–47 (2005)
4. Beckert, B., Hähnle, R., Schmitt, P.: Verification of Object-Oriented Software: The KeY Approach. Springer, Secaucus (2007)
5. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In: Tests and Proof, pp. 90–104. Springer, Heidelberg (2009)

6. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL Class Diagrams using Constraint Programming. In: IEEE Int. Conf. on Software Testing Verification and Validation Workshop, pp. 73–80 (April 2008)
7. Cabot, J., Clarisó, R., Riera, D.: Verifying UML/OCL Operation Contracts. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 40–55. Springer, Heidelberg (2009)
8. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A Challenging Model Transformation. In: Int. Conf. on Model Driven Engineering Languages and Systems, pp. 436–450. Springer, Heidelberg (2007)
9. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: Design, Automation and Test in Europe, pp. 1341–1344. IEEE Computer Society, Los Alamitos (2010)
10. Soeken, M., Wille, R., Drechsler, R.: Verifying Dynamic Aspects of UML Models. In: Design, Automation and Test in Europe. IEEE Computer Society, Los Alamitos (2011)
11. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise modeling with UML. Addison-Wesley Longman, Boston (1999)
12. Constantinides, G.A., Cheung, P.Y.K., Luk, W.: Synthesis of Saturation Arithmetic Architectures. ACM Trans. Design Autom. Electr. Syst. 8(3), 334–354 (2003)
13. Cook, S.A.: The complexity of theorem-proving procedures. In: ACM Symp. on Theory of Computing, pp. 151–158. ACM, New York (1971)
14. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Design Automation Conference, pp. 530–535. ACM, New York (2001)
15. Goldberg, E.I., Novikov, Y.: BerkMin: A Fast and Robust Sat-Solver. In: Design, Automation and Test in Europe, pp. 142–149. IEEE Computer Society, Los Alamitos (2002)
16. Eén, N., Sörensson, N.: An Extensible SAT-solver. Theory and Applications of Satisfiability Testing, 502–518 (May 2003)
17. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, February 2009. IOS Press, Amsterdam, NL (February 2009)
18. Armando, A., Castellini, C., Giunchiglia, E.: SAT-Based Procedures for Temporal Reasoning. In: Biundo, S., Fox, M. (eds.) ECP 1999. LNCS, vol. 1809, pp. 97–108. Springer, Heidelberg (2000)
19. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)
20. Wille, R., Große, D., Soeken, M., Drechsler, R.: Using Higher Levels of Abstraction for Solving Optimization Problems by Boolean Satisfiability. In: IEEE Symp. on VLSI, pp. 411–416. IEEE Computer Society, Los Alamitos (2008)
21. Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: Tools and Algorithms for Construction and Analysis of Systems, pp. 174–177. Springer, Heidelberg (2009)
22. Jackson, D., Damon, C.: Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. IEEE Trans. on Software Engineering 22(7), 484–495 (1996)
23. Davenport, J.H., Heintz, J.: Real Quantifier Elimination is Doubly Exponential. Journal of Symbolic Computation 5(1-2), 29–35 (1988)