Strengthening SAT-Based Validation of UML/OCL Models by Representing Collections as Relations

Mirco Kuhlmann and Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen
{mk,gogolla}@informatik.uni-bremen.de

Abstract. Collections, i. e., sets, bags, ordered sets and sequences, play a central role in UML and OCL models. Essential OCL operations like role navigation, object selection by stating properties and the first order logic universal and existential quantifiers base upon or result in collections. In this paper, we show a uniform representation of flat and nested, but typed OCL collections as well as strings in form of flat, untyped relations, i. e., sets of tuples, respecting the OCL particularities for nesting, undefinedness and emptiness. Transforming collections and strings into relations is particularly needed in the context of automatic model validation on the basis of a UML and OCL model transformation into relational logic.

1 Introduction

Models are a central means to master complex systems. Thus, for developing systems, building precise models is a main concern. Naturally, the examination of the validity of complex systems must be supported via tracing and checking model properties.

We employ the Unified Modeling Language (UML) and its accompanying textual constraint and query language OCL (Object Constraint Language) for the description of models. For automatically analyzing and validating models, we utilize *relational logic*. Relational logic is efficiently implemented in Alloy [11] and its interface Kodkod [19] which transforms relational models into boolean satisfiability (SAT) problems. As a consequence, our task consists in transforming our source languages UML and OCL as well as the considered model properties into structures and formulas of the target language relational logic. This way, we enable SAT-based validation of UML/OCL models. We have started to implement the transformation from UML/OCL to relational logic in a so-called model validator [13] which has been integrated into our UML-based Specification Environment (USE) [8].

In this paper, we focus on a vital aspect of UML/OCL models, namely the handling of OCL collection kinds (set, bag, ordered set, and sequence)[1] and

[1] One collection kind (e. g., set) can be manifested in different concrete collection types (e. g., Set(Integer) and Set(Bag(String))).

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 32–48, 2012.

strings. OCL collections and collection operations play a central role in the language. They are crucial for building precise UML/OCL models which can be successfully analyzed and checked. For instance, the evaluation of existentially and universally quantified formulas is based upon collections of values like in `Person.allInstances->exists(p|p.age<18)`. Another naturally used operation is role navigation which results in collection values, e. g., when the allowed states of a structural model given in form of a class diagram have to be restricted and the restriction involves two classes and an association navigation path between the classes, the association path will be evaluated in OCL through a collection expression. The following example ensures a minimum salary by collecting the employees of all companies using navigation: `Company.allInstances.employee->forAll(e| e.salary>3000)`.

The example model shown as a UML class diagram in Fig. 1 emphasizes the use of strings and different types of collections. A university is located in a specific state encoded by a two character string, e. g., 'DK' or 'US'. A person may have several postal and e-mail addresses and may be enrolled in a university. While postal addresses are always unordered (Set(String)), the e-mail addresses of a person can be prioritized by using an ordered set of addresses, or be retained without any prioritization by using a set. The abstract type Collection(String) allows for determining the concrete type (OrderedSet(String) or Set(String)) at runtime.

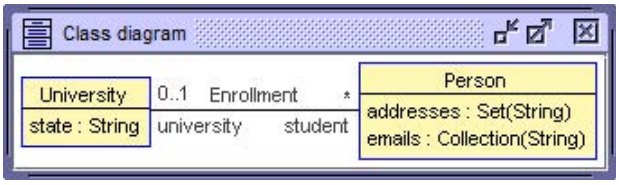

Fig. 1. Example UML Class Diagram with String and Collection Type Attributes

The following example OCL invariant further constrains the model. It requires each person who is a student at a university to have a postal address in the same state the university is located. For this purpose, it checks whether the string representing the university's location also occurs at any position in at least one postal address string of the student.

```
context Person inv AccessibleStudents:
  self.university.isDefined implies
    self.addresses->exists(a|
      Set{1..a.size}->exists(i|a.substring(i,i+1)=self.university.state))
```

However, when comparing UML and OCL, our source languages for describing models, and relational logic, our direct target language utilized for automatic model validation, we observe an impedance mismatch: (a) OCL offers four collection kinds whereas relational logic and its implementation Alloy directly support only relations, i. e., sets of flat tuples; *"other structures (such as lists and*

sequences) are not built into Alloy the way sets and relations are" (see p. 158 in [11]) (b) OCL is a typed language whereas plain relational logic is untyped. This means that the OCL type system has to be represented in relational logic and the missing collection kinds have to be encoded as sets. (c) The lack of *"higher-order relations"* implies that *"collections of collections"* which often occur in UML/OCL models are not directly supported in Alloy [1]. Consequently, the challenge is to respect all involved OCL particularities in the translation which means that nested collections as well as OCL type rules, the undefined value, and empty collections deserve special attention.

We present a uniform transformation which respects all language inherent differences between UML/OCL collections and flat relations of relational logic. Furthermore, we enable the representation of structured string values in relational logic. A comprehensive representation of UML/OCL collections and strings in relational logic is the premise for the translation of collection and string operations and, hence, a comprehensive approach to automatic UML/OCL model validation utilizing Kodkod and SAT solving. However, there is currently no other SAT-based approach which supports models like the example depicted in Fig. 1 or the related OCL constraint.

The rest of this paper is structured as follows. Section 2 associates the content of this paper with the SAT-based model validation context. The central Sect. 3 will show how OCL collections and strings are represented as relations. First, we introduce the approach considering exemplary transformations. Then, we illustrate the underlying transformation algorithms. After a discussion of performance implications in Sect. 4 and related work Sect. 5, we conclude with Sect. 6.

2 Model Validation via SAT Solving: Context

Our validation approach bases upon checking model properties by inspecting the properties of model instances (snapshots), e. g., the existence or non-existence of specific snapshots allows conclusions about the model itself. As shown in Fig. 2, the USE *model validator* allows developers to automatically analyze properties of their UML/OCL models by translating them into relational structures, i. e., bounded relations and relational formulas, which can be handled by the model finder Kodkod. In addition to a model, the properties under consideration, usually given in form of OCL expressions, as well as user-configurations with respect to the search space are transformed and handed over to Kodkod.

Kodkod in turn employs SAT solvers to find a solution, i. e., proper instantiations of specified relations, fulfilling the given formulas. Found SAT instances are therefore translated back into instances of the specified relations. In the end, the model validator transforms the relational instances into instances of the UML/OCL model and visually presents the found solution in form of an object diagram to the developer.

Since UML/OCL collections and strings values play a central role in precisely specified models, corresponding validation approaches must support the four collection kinds and their peculiarities as well as strings in order to provide a

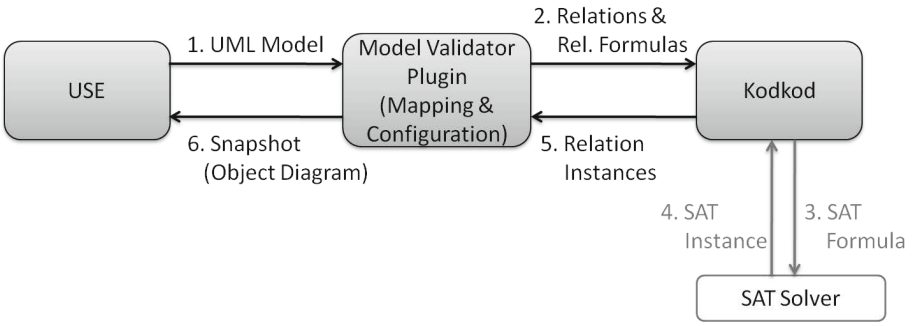

Fig. 2. Transformation process involving the USE model validator

comprehensive validation platform. The transformation algorithm discussed in this paper has been implemented in the model validator enabling the definition of meaningful OCL constraints on the one hand, and user-defined properties which are to be inspected on the other hand.

3 Transforming Collections and Strings into Relations

Relational logic describes formulas whose evaluation is based on flat relations with different arities, i. e., sets of tuples with atomic components, since relational logic forbids nested relations. Beside boolean and integer operations, relational logic naturally supports set operations like union and set comprehension. A central operation is the relational join for accessing specific components (i. e., columns) of tuples and for connecting tuples of different relations.

Relations generally have the same properties as OCL sets which are unordered and do not allow duplicate elements.[2] Thus, there is a straightforward way to translating non-nested sets into unary relations, e. g., `Set{2,1,3}` can be represented by the relation `[[3],[1],[2]]`. On the other hand, the following characteristics of OCL collections must be respected:

- Bags and sequences require the support of duplicate elements.
- Ordered sets and sequences require the support of ordered elements.
- All collection kinds require the support of nested collections.

A universally applicable transformation must cover all of these properties.

3.1 The Basic Idea

In this subsection we consider the first two named properties (support of duplicate and ordered elements), nested collections, and strings as well as the handling of undefined and empty values in greater detail. The comparability of collection and string values must be preserved by their relational representation. This essential aspect is discussed at the end of this subsection.

[2] Henceforth, the term 'set' refers to an OCL set.

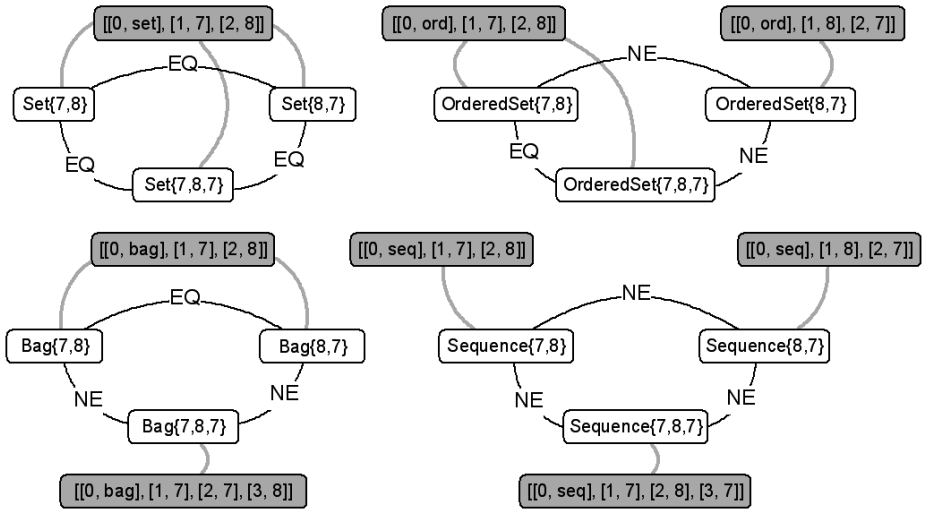

Fig. 3. Distinction of OCL collections and the corresponding translation into relations

Handling of Flat Collections. In Fig. 3 we illustrate the properties of the four OCL collection kinds based on three concrete literals, in each case. The OCL literals – depicted in white boxes – involve duplicate elements and elements in a particular order. Two literals are equal (EQ), i. e., they represent the same value, or are not equal (NE). For instance, while `Bag{7,8}` equals `Bag{8,7}`, the collection value `OrderedSet{7,8}` does not equal `OrderedSet{8,7}`.

Collection literals describing the same value should naturally yield the same (identifying) relational representation which are shown in grey boxes and by grey connecting lines. The translation assigns an index $1 \leq i \leq n$ to each element of a collection with n elements, determining an explicit element position. See, for example, the relational representation of the value `Sequence{7,8,7}`. The corresponding relational representation relates the index 1 to the first element (7), index 2 to the second element (8) and index 3 to the last element (7).

The depicted, grey relations reveal four distinctive features of the transformation which do not directly result from the collection properties, but from explicit design decisions:

- The elements of a set or bag are indexed in the respective relational representation, although sets and bags are intrinsically unordered and sets do not include duplicate elements.
- m duplicates of element e in a bag occur m times in the respective relational representation.[3]
- The elements of sets and bags are sorted in the resulting relations based on the natural order of integer values. For example, considering the literals `Set{7,8}` and `Set{8,7}`, the integer value 7 always precedes the value 8 in the relational representation of both sets.

[3] An alternative would be tuples counting element occurrences (e. g., $[e,m]$).

– The special index 0 indicates a typing tuple that determines the collection kind a relation represents (set, bag, ord, seq).

The first two features (an order for sets resp. bags and retention of explicit duplicates) directly follow from our intention to define a uniform transformation with no exceptional cases and resulting case distinctions, clearly simplifying (a) the representation of nested collections, and (b) the translation of OCL collection operations into relational logic. The last two aspects (sorting elements and explicit typing) allow us to compare OCL collection values in relational logic through an explicit sorting of their elements, as we will discuss at the end of this subsection.

Handling of Nested Collections and Strings. In the case of nested collections, the elements of a collection are in turn collections. In order to encapsulate the individual collections, i. e., to determine which value belongs to which collection, there must be an additional indicator. Following our uniform translation, a natural way to representing collection type elements is the use of a new index column, as shown in the following example:

```
Set{7}                --> [[  0,set  ],   the relation represents a set
                           [  1,7    ]]   the first element is 7
Set{8,9}              --> [[  0,set  ],   the relation represents a set
                           [  1,8    ],   the first element is 8
                           [  2,9    ]]   the second element is 9
Sequence{Set{7},
         Set{8,9}}    --> [[0,seq,seq],   the relation represents a sequence[4]
                           [1,0,set  ],   the first element is a set
                           [1,1,7    ],   the first element of the set is 7
                           [2,0,set  ],   the second element is a set
                           [2,1,8    ],   the first element of the set is 8
                           [2,2,9    ]]   the second element of the set is 9
```

The support of OCL collections also allows for representing string values. While strings may be seen as atomic values (e. g., `'Ada' --> [[Ada]]`), it is often necessary to consider a string as a value with an inner structure. Thus, because there is a need in OCL for manipulating and querying strings, they are treated like sequences of characters and are identified by a respective string typing tuple (e. g., `'Ada' --> [[0,str],[1,A],[2,d],[3,a]]`). A set of strings can thus be seen as sequences of values nested in a set:

```
Set{'Ada','Bob'} --> [[0,set,set],
                      [1,0,str], [1,1,A], [1,2,d], [1,3,a],
                      [2,0,str], [2,1,B], [2,2,o], [2,3,b]]
```

[4] Since all tuples of a relation must have the same arity, we use the collection kind indicator (e. g., seq) to *fill* typing tuples until they yield the required number of components. Multiple indicators in one typing tuple, thus, have no special meaning.

In OCL, sets, bags, sequences and ordered sets are specializations of *Collection*, and all basic types are subtypes of *OclAny*. Thus, for example, we can create collections including elements of type Collection(OclAny):

```
Set{Sequence{5,6,5},Set{'Ada',7,'Bob',8}} =
Set{Set{7,8,'Ada','Bob'},Sequence{5,6,5}} -->
[[0,set,set,set],
 [1,0,set,set],
 [1,1,1,7],
 [1,2,1,8],
 [1,3,0,str],[1,3,1,A],[1,3,2,d],[1,3,3,a],
 [1,4,0,str],[1,4,1,B],[1,4,2,o],[1,4,3,b],
 [2,0,seq,seq],
 [2,1,1,5],
 [2,2,1,6],
 [2,3,1,5]]
```

If string and non-string basic types are mixed, the non-string basic type values are brought into the complex string representation by handling them as if they were strings of length one with an absent typing tuple, e. g., the integer value `7` is represented as `[1,7]` instead of `[7]`.

The translation result of the previous example is a relation that represents a set including collections of sequences. Each additional nesting level adds a further index column to the relation. The fourth column determines the character or integer value. The third column determines the position of the characters in a string. The second column determines the position of a string within a collection. The first column determines the position of the collection in the outer set.

For instance, the tuple `[1,3,2,d]` determines `'d'` to be the second character of the third element (`'Ada'`) in the first element (`Set{7,8,'Ada','Bob'}`) of `Set{Set{7,8,'Ada','Bob'},Sequence{5,6,5}}`.

Undefined and Empty Collections. Empty collections are naturally represented by the absence of further tuples besides the typing tuple. Undefined (`un`) collections on the other hand yield a characteristic relational representation. This representation allows us to identify at which nesting level an undefined value occurs (c. f. the three different levels in the following example). Furthermore, undefined values are not accompanied by typing tuples, since the information which concrete type an undefined value represents is irrelevant.

```
Set{Undefined, Set{}, Set{Undefined, Set{}, Set{Undefined}}} -->
[[0,set,set,set],    the relation represents a set
 [1,un,un,un],       the first element is an undefined collection
 [2,0,set,set],      the second element is an empty set
 [3,0,set,set],      the third element is a set
 [3,1,un,un],        its first element is an undefined collection
 [3,2,0,set],        its second element is an empty set
 [3,3,0,set],        its third element is a set
 [3,3,1,un]]         which includes an undefined value
```

Making Ordered Relations Comparable. In Fig. 3 we depicted the equality and inequality of specific collection literals. Since sets and bags are intrinsically unordered, we obtain the properties: `Set{7,8}=Set{8,7}` and `Bag{7,8}=Bag{8,7}`. Accordingly, an equality check regarding the relational representation of both sets and both bags, respectively, must evaluate to true. We can achieve a general valid comparability at the relational level (a) by sorting the elements of the relational representations of sets and bags on demand (e. g., in the case of an equality check, or the casting operation `Set::asSequence()`), or (b) by sorting the elements already during the creation process. We applied the latter strategy which results in a unique representation of equal collection values through direct sorting:

`Set{7,8} --> [[0,set],[1,7],[2,8]] <-- Set{8,7}` and
`Bag{7,8} --> [[0,bag],[1,7],[2,8]] <-- Bag{8,7}`

SAT-based validation implies bounded search spaces, i. e., at the UML level, a limited set of covered model instances. Hence, the set of participating values (boolean, integer, enumeration, character, or object type)[5] is finite. This allows us to create a total order on all available values and to define a sorting algorithm with respect to the precedence of these values. Within the previous example of nested collections with mixed basic type values, the literals `Set{7,8,'Ada','Bob'}` and `Set{'Ada',7,'Bob',8}` yield the same relational representation after sorting the elements (integer precedes string). The example shows three further properties of the sorting algorithm:

- The sorting of sets and bags has a recursive nature, i. e., sorting is applied at each nesting level. Before an outer set or bag can be sorted, its elements must have been sorted.
- Sequences, ordered sets and strings are never sorted, since the order of their elements (or characters, respectively) is significant (e. g., `Sequence{5,6,5}<> Sequence{5,5,6}`). If they, however, include set or bag valued elements, these sets and bags have to be sorted (e. g., `Sequence{Set{8,7},Set{2,1}}= Sequence{Set{7,8},Set{1,2}}`).
- Beside basic values, also strings and collections obtain an explicit precedence, based on the collection kind, number of elements (or characters, respectively), and precedence of their elements (or characters), e. g., `'Bo' < 'Ada'`, `'Ada' < 'Bob'`, `Set{7} < Bag{7}`, `Set{7} < Set{1,2}`, and `Set{1,2} < Set{7,8}`.

The need for typing tuples directly follows from the need for comparability. Since the values `Bag{7,8} --> [[0,bag],[1,7],[2,8]]` and `Set{7,8} --> [[0,set],[1,7],[2,8]]` are not equal, a relational representation without typing tuples would lead to an invalid conclusion for equality:

`Set{7,8} --> [[1,7],[2,8]] <-- Bag{7,8}` $\lightning$

[5] The character type includes the alphabetic characters which are needed to create string values. The basic predefined type Real is currently not supported.

3.2 Realization of the Transformation Algorithms

In this section, we explain the details of translating UML/OCL collections into relations by considering the relevant transformation algorithms. Since the algorithm for constructing strings at the relational level is a special case of the creation of flat sequences, we focus on the general handling of collections.

First, we consider the core algorithm describing the creation of collection values. Then, we go into details of sorting collections which is needed in the context of sets and bags.

The Collection Creation Algorithm. The algorithm for creating UML/OCL collections in their relational representation includes two main aspects: (a) Each given element which should be included into the collection and is already available in a relational representation is incrementally indexed and added to the resulting relation. Duplicate elements are discarded if the resulting relation should represent a set or ordered set. (b) Relations representing sets or bags are sorted in the end. In this case, the following sorting algorithms become relevant. For details see Algorithm 1 in the appendix.

The Collection Sorting Algorithms. The central sorting algorithm includes the main activities for sorting relations representing sets or bags. It takes all possible pairs of elements existing in the given relation and determines which element precedes the other. The number of predecessors an element possesses then determines its new position in the sorted relation. The element without predecessors obtains the first position (index 1), and an element with x predecessors becomes the $x + 1$th element in the sorted relation. For details see Algorithm 2 in the appendix.

The precedence of two complex, collection-valued elements is determined by a further algorithm which respects the following precedence rules: undefined collections precede sets, sets precede sequences, sequences precede bags, bags precede ordered sets; in the case of collections of the same kind, the number of elements within these collections becomes relevant; if the numbers are identical, the precedence of the elements within the two considered collections must be recursively determined. For details see Algorithm 3 in the appendix.

The recursive calculation of element precedences may end at different levels of nested values. For example, consider the following pairs of collections:

A: `Set{Set{Set{7}}}`, `Bag{Set{Set{7}}}`
B: `Set{Set{Set{7}}}`, `Set{Set{Bag{7}}}`
C: `Set{Set{Set{7}}}`, `Set{Set{Set{8}}}`

In each case the left side precedes the right side. While in pair A the precedence can be directly determined (sets precede bags), pair B demands a nested (recursive) comparison until the elements `Set{7}` and `Bag{7}` at the second nesting level are reached. In the case of pair C, the final level of recursion is reached, i. e., the level at which only simple values occur (`7` and `8`).

As mentioned before, each validation task given to the USE model validator describes a *finite* user-defined *universe* of simple values (i. e., boolean, integer, enumeration, character, or object type). As a consequence, the precedence of these simple values can always be specified via a total order relation.[6] Given such a total order relation and two simple values, the precedence of both values can directly be calculated. For details see Algorithm 4 in the appendix.

4 Discussion

A bounded search space of the model validator (resp. Kodkod) requires bounded relations and thus bounded collection representations. Kodkod considers the set of all available (user-defined) simple values as a universe of atoms. Relations are bounded to a set of possible tuples by determining a set of possible atoms (a domain) for each column of the relation tuples. For instance, a relation that represents the type Set(Set(Boolean)) yields tuples of the form:

[$index_1$, $index_2$, $value$], with
$index_1 \in Domain_1 = \{0, 1, ..., x, \texttt{un}\}$, where x is a user defined maximum number,
$index_2 \in Domain_2 = \{0, 1, ..., x, \texttt{un}, \texttt{set}\}$, and
$value \in Domain_3 = \{\texttt{true}, \texttt{false}, \texttt{un}, \texttt{set}\}$.

There are $|Domain_1| * |Domain_2| * |Domain_3|$ possible tuples which can be included by an instance of the considered relation. As we have explained before, each nesting level of collections adds one additional column to the respective relation, increasing its arity by one. A nesting depth of n implies a relation of arity $n+2$ (or $n+3$ if strings are involved). Consequently, each additional nesting level considerably increases the search space and correspondingly reduces the SAT solving performance. Furthermore, Kodkod limits the maximum arity of involved relations und thus the maximum nesting depth: $|universe|^{max\_arity} < 2^{31} - 1$. Future work will comprise the optimization of the search space by bounding the possible tuples to OCL collection specific patterns.

There is also potential for optimization with respect our representation of OCL collections as flat relations. However, our aim is to present a universally defined and applicable approach in this paper. A concrete implementation can naturally realize several optimizations like discarding typing tuples, if the collection types can be statically determined, i. e., *Collection* is not involved, or using a simple representation of OCL sets and strings in form of unary relations, if only non-nested collections and no complex string values are needed. That is, while our approach supports all OCL collections structures, it can be thinned out as required.

In order to inspect the performance implications in the context of a complete implementation of our approach, let us consider the class diagram shown in Fig. 4

[6] In the case of values which do no yield a natural order, the model validator explicitly induces one, e. g., the order of objects is determined by the order the corresponding object identifiers are declared within the model validator, independent of the classes they instantiate.

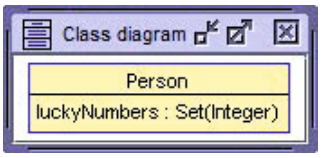

Fig. 4. UML Class with a Set-valued Attribute

which models persons with a set of lucky numbers, as well as the following OCL invariant which demands that people have unique sets of lucky numbers. Please note that in our approach for transforming UML and OCL models into relational models, classes are translated into unary relations, holding atoms which represent object identifiers. Attributes are translated into relations connecting object identifiers with attribute values, plus relational constraints ensuring attribute values of the specified type. In the case of collection-valued attributes, an object is related to each individual tuple of the corresponding collection value. An example instance of the attribute relation `Person_luckyNumbers` is shown at the end of this section.

```
context p:Person
  inv uniqueLuckyNumbersSets:
    Person.allInstances->forAll(p1,p2|
      p1.luckyNumbers=p2.luckyNumbers implies p1=p2)
```

`-->` (sketch of a translation into relational logic)

```
(all p1:Person, p2:Person |
  p1.Person_luckyNumbers=p2.Person_luckyNumbers => p1=p2)
```

First, we use the model validator to automatically translate this UML and OCL model into a relational model, and initiate a search for valid instances in the context of 4, 8, and 12 person objects. Then, we repeat this procedure for nested attribute types. Table 1 reveals the corresponding search times. The second column yields the results for a simple set representation using unary relations, e. g., `Set{7,8} --> [[7],[8]]` instead of the complex representation discussed in this paper, e. g., `Set{7,8} --> [[0,set],[1,7],[2,8]]`.

Table 1. Comparison of SAT Solving Performance regarding different Nesting Levels

#Persons	Set(Int) (simple)	Set(Int)	Set(Set(Int))	Set(Set(Set(Int)))
4	62 ms	437 ms	2200 ms	14955 ms
8	109 ms	764 ms	5132 ms	62540 ms
12	140 ms	1326 ms	16497 ms	140522 ms

In the context of type Set(Set(Integer)) and 4 required person objects, we, for example, obtain the following class and attribute relation instances as a result which are automatically transformed by the model validator into the object diagram shown in Fig. 5.

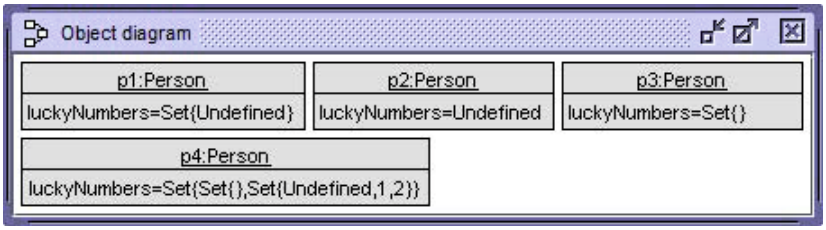

Fig. 5. Solution in the context of 4 Objects and Type Set(Set(Integer))

```
Person=[[p1],[p2],[p3],[p4]]
Person_luckyNumbers=[
  [p1,0,set,set,set],[p1,1,un,un,un],
  [p2,un,un,un,un],
  [p3,0,set,set,set],
  [p4,0,set,set,set],
    [p4,1,0,set,set],
    [p4,2,0,set,set],[p4,2,1,un,un],[p4,2,2,1,1],[p4,2,3,1,2]]
```

5 Related Work

Our paper has connections to many related works. The collection kinds set, bag and list (sequence) are considered in the context of functional programming in [10,23] whereas our approach is designed for object-oriented design and modeling. The Object Query Language OQL [7] uses three (set, bag, list) of the four OCL collections in the same way as they are employed in OCL but without defining a formal semantics. [16] makes a proposal to complete the OCL collections in a lattice-like style leading to union and intersection types. The work concentrates on the OCL collection kind set.

General type and container constructors similar to sets or bags are considered for database design using ER modeling in [9]. [4] represents the OCL standard collections with an extended OCL metamodel allowing for practical tool support with the aim of code generation. [6] studies fundamental properties of OCL collections in order to establish a new generalization hierarchy and focusses of the relationship between sets and ordered sets. [22] proposes a unified description of OCL collection types and OCL basic data types. [14] translates OCL into Maude and represents OCL collections by introducing new algebraic sorts without considering the complete OCL type system. A mapping of non-nested OCL collections and strings into bit-vector logic is done in [17]. In [5] the authors describe a staged encoding of OCL strings that performs reasoning on string equalities and string lengths before fully instantiating the string.

Our approach is based on relational logic which is implemented in the powerful Alloy system described in [11]. Alloy supports non-nested sets and sequences modeled as functions mapping integer (indices) to the sequence elements. The

UML2Alloy approach presented in [1] tackles the translation of UML and OCL concepts into Alloy. The authors sketch the possibility to describe sequences, bags and ordered sets via Alloy structures, but do not discuss further details like the preservation of collection comparability. While the representation of nested collections in Alloy is not possible, because of the lack of higher-order relations and restrictions with respect to available Alloy structures, the Alloy interface Kodkod [19] which we utilize in our approach allows users to handle plain relations with arbitrary contents.

Alloy and Kodkod are used for many purposes. [3] translates conceptual models described in OntoUML for validation purposes into Alloy. In [12] modeling languages and their formal semantics, in [21] enterprise architecture models based on ontologies are specified and analyzed with Alloy. Kodkod has been utilized for executing declarative specifications in case of runtime exceptions in Java programs [15], reasoning about memory models [20], or generating counterexamples for Isabelle/HOL a proof assistant for higher-order logic (Nitpick) [2]. [18] use Kodkod for checking the consistency of models described with basic UML concepts.

6 Conclusion

We have discussed a uniform representation of strings and nested, typed collections in form of flat, untyped sets respecting the OCL particularities for nesting, undefinedness and emptiness. Collections are a central modeling feature in UML and OCL for model inspection and model validation and verification. We have successfully implemented this approach in our model validator and applied it in several middle-sized examples.

As future work, we want to check the approach with larger case studies. In particular, we have to check whether efficiency improvement may be made by factoring out type information from the collection instances. A small benchmark for checking collection and string values could be developed. With respect to OCL, one might propose an OCL simplification based on the experience with the difficult handling of undefinedness in order to shorten the gap between the source and target languages.

Furthermore, the concepts for sorting collections at the relational level which we have discussed in this paper can be reused for standardizing corresponding OCL *sort* operations. Such operations could deterministically lead from unordered collections to sorted collections, e. g.,

```
Set{1,2,3}->sort = OrderedSet{1,2,3} = Set{2,3,1}->sort,
Bag{1,2,2,3}->sort = Sequence{1,2,2,3} = Bag{2,1,2,3}->sort, and
Set{OrderedSet{2,1}, OrderedSet{1,2}}->sort =
OrderedSet{OrderedSet{1,2}, OrderedSet{2,1}}.
```

However, also ordered collections (which are not necessarily sorted) need sometimes to be sorted, so that we propose using this sort operation for all four collection kinds.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. SoSyM 9(1), 69–86 (2010)
2. Blanchette, J.C., Nipkow, T.: Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010)
3. Braga, B.F.B., Almeida, J.P.A., Guizzardi, G., Benevides, A.B.: Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. ISSE 6(1-2), 55–63 (2010)
4. Bräuer, M., Demuth, B.: Model-Level Integration of the OCL Standard Library Using a Pivot Model with Generics Support. ECEASST 9 (2008)
5. Büttner, F., Cabot, J.: Lightweight String Reasoning for OCL. In: Vallecillo, A., et al. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 240–254. Springer, Heidelberg (2012)
6. Büttner, F., Gogolla, M., Hamann, L., Kuhlmann, M., Lindow, A.: On Better Understanding OCL Collections *or* An OCL Ordered Set Is Not an OCL Set. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 276–290. Springer, Heidelberg (2010)
7. Cattell, R.G.G., Barry, D.K.: The Object Data Standard: ODMG 3.0. Morgan Kaufmann (2000)
8. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
9. Hartmann, S., Link, S.: Collection Type Constructors in Entity-Relationship Modeling. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 307–322. Springer, Heidelberg (2007)
10. Hoogendijk, P.F., Backhouse, R.C.: Relational Programming Laws in the Tree, List, Bag, Set Hierarchy. Sci. Comput. Program. 22(1-2), 67–105 (1994)
11. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
12. Kelsen, P., Ma, Q.: A Lightweight Approach for Defining the Formal Semantics of a Modeling Language. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 690–704. Springer, Heidelberg (2008)
13. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 290–306. Springer, Heidelberg (2011)
14. Roldan, M., Duran, F.: Dynamic Validation of OCL Constraints with mOdCL. ECEASST 44 (2011)
15. Samimi, H., Aung, E.D., Millstein, T.: Falling Back on Executable Specifications. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 552–576. Springer, Heidelberg (2010)
16. Schürr, A.: A New Type Checking Approach for OCL Version 2.0? In: Clark, A., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 21–41. Springer, Heidelberg (2002)
17. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 152–170. Springer, Heidelberg (2011)
18. Van Der Straeten, R., Pinna Puissant, J., Mens, T.: Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 69–84. Springer, Heidelberg (2011)

19. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
20. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: checking axiomatic specifications of memory models. SIGPLAN Not. 45, 341–350 (2010), http://doi.acm.org/10.1145/1809028.1806635
21. Wegmann, A., Le, L.-S., Hussami, L., Beyer, D.: A Tool for Verified Design using Alloy for Specification and CrocoPat for Verification. In: Jackson, D., Zave, P. (eds.) Proc. First Alloy Workshop (2006)
22. Willink, E.D.: Modeling the OCL Standard Library. ECEASST 44 (2011)
23. Wong, L.: Polymorphic Queries Across Sets, Bags, and Lists. SIGPLAN Notices 30(4), 39–44 (1995)

A Collection Creation and Sorting Algorithms

The presented algorithms abstract from complex language characteristics of relational logic and are designed to clarify the overall activities for creating and sorting UML/OCL collections at the relational level.

```
collectionCreation(colKind, ... elements)
input: the required collection kind (colKind ∈ {set,bag,ord,seq}),
       a list of elements already available in relational representation
output: a collection of kind colKind including the properly ordered elements
  newCol ← []
  index ← 1
  for each e in elements do
    if colKind is bag or sequence or e does not already exist in newCol then
      indexed_e ← add index as first component to each tuple of e
      newCol ← newCol ∪ indexed_e
      index ← index + 1
    end
  end
  newCol ← newCol ∪ typing tuple
  if newCol is set or bag then
    return complexSort(newCol)
  else
    return newCol
  end
```

Algorithm 1: Creating relations for representing UML/OCL collections

```
complexSort(col)
input: an unsorted relation col representing a set or bag
output: a relation with sorted elements
  predecessorMap ← empty map
  for each e1, e2 in col do
    if complexPredecessor(e1,e2) = e1
      or (complexPredecessor(e1,e2) = Undefined
          and original e1 position < original e2 position) then
      predecessorMap.add(e1,e2)
    end
  end
  positionMap ← empty map
  for each e in col do
    positionMap.add(e, |predecessorMap(e)|+1)
    modified_e ← e with topmost index replaced by positionMap(e)
    col ← col with e replaced by modified_e
  end
  return col
```

Algorithm 2: Sorting relations representing sets or bags

```
complexPredecessor(e1, e2)
input: two complex elements
output: the preceding element
  if e1 is a singleton, i. e., a relation including just a simple value then
    return simplePredecessor(e1, e2)
  else
    if e1 is undefined and e2 is undefined then return Undefined end
    if e1 is undefined and e2 is not undefined then return e1 end
    if e1 is not undefined and e2 is undefined then return e2 end

    if e1 is a set and e2 is not a set then return e1 end
    if e1 is not a set and e2 is a set then return e2 end

    if e1 is a sequence and e2 is not a sequence then return e1 end
    if e1 is not a sequence and e2 is a sequence then return e2 end

    if e1 is a bag and e2 is not a bag then return e1 end
    if e1 is not a bag and e2 is a bag then return e2 end

    if e1 has less elements than e2 then return e1 end
    if e2 has less elements than e1 then return e2 end
    relevantElement ← null
    for each position i in e1 and e2 do
      e1_elem ← element at position i of e1
      e2_elem ← element at position i of e2
      if complexPredecessor(e1_elem, e2_elem) = Undefined then
        continue
      else
        if complexPredecessor(e1_elem, e2_elem) = e1_elem then
          return e1
        else
          return e2
  end end end end
  return Undefined
```

Algorithm 3: Determining the precedence of elements

```
simplePredecessor(TO, e1, e2)
input: a binary relation TO specifying a total order for all simple values
       so that if [x,y] ∈ TO, x is the direct predecessor of y,
       two simple elements
output: the preceding element
  if e1 = e2 then
    return Undefined
  else if [e1,e2] ∈ closure(TO) then
    return e1
  else
    return e2
  end
```

Algorithm 4: Determining the precedence of simple values