Encoding Object-oriented Models in MiniZinc

Gottfried Schenner[1] and Richard Taupe[1,2(✉)]

[1] Siemens AG Österreich, Corporate Technology, Vienna, Austria
firstname.lastname@siemens.com
[2] Alpen-Adria-Universität, Klagenfurt, Austria

Abstract. Object-oriented models are omnipresent in software engineering. To use a CSP solver with an object-oriented model, the mapping between the object-oriented data model and the CSP is typically implemented manually and specific to the problem to solve. This paper discusses different generic encodings to reason about object-oriented models using the MiniZinc constraint modelling language. We focus mainly on finding instantiations of object-oriented models under domain-specific constraints. Possible applications are found in product configuration and software engineering.

Keywords: Object-oriented programming · Constraint programming · MiniZinc

1 Introduction

Object-oriented models are widely used in software engineering. Object-oriented models facilitate the development of large software projects and provides a mechanism to communicate the domain of concern visually to all project members. UML class diagrams are often used for this purpose.

The object-oriented model is not only visually appealing, but does also contain valuable information about the cardinalities of the classes involved. This information is given in the form of association multiplicities, singleton patterns, etc. Used to a lesser extent in practice, a specification language like OCL can be used to fully specify the constraints a valid instantiation must satisfy.

An important reasoning task for object-oriented models is to find an instantiation that satisfies some additional constraints, e.g. an instantiation containing X objects of type Y, but not more than Z objects altogether. Unfortunately there are few tools (e.g. [12]) available that provide this functionality.

Constraint programming is a powerful approach to declarative problem solving. Although modern constraint solvers are often implemented in object-oriented programming languages, most solvers do not support reasoning about object-oriented models. Therefore some special encoding is necessary in order to use a constraint solver with an object-oriented model. Usually this mapping is done ad hoc by the knowledge engineer, especially if the domain of concern involves just a few classes and attributes.

MiniZinc [14] is the de facto standard to express constraint problems. In this paper we show how to generically encode the instantiation of an object-oriented model in MiniZinc. In doing so we explore different strategies to encode associations. The resulting encoding is a MiniZinc program that generates all instantiations of the model up to a given maximum number of objects. By adding additional problem-specific constraints, we can accomplish various reasoning tasks. Examples for this are checking an instantiation, completing an instantiation, and checking the validity of assertions.

The remainder of this article is structured as follows: In Section 1.1 we relate our approach to previous work. In Section 1.2 we provide a motivation for the topic, which is accompanied by a motivating example in Section 1.3. Afterwards, our approach to encode object-oriented models in MiniZinc is presented in Section 2. The performance of various encodings is evaluated in Section 3, followed by our conclusions in Section 4.

1.1 Related Work

Our aim for this paper is to integrate of object-oriented programming with constraint programming. Therefore we use the term *OOCSP* (object-oriented constraint satisfaction programming). This term was introduced by M. Paltrinieri [15, 16], who promotes it as a framework to design CSPs in traditional constraint programming languages through a visual methodology. The term *OOCSP* is used again by [11], where an object-oriented constraint language of that name is presented. Specifications in this language are not translated into the input format of an off-the-shelf constraint solver, but to first-order logic. OOCSP programs are then solved by a dedicated solver that is based on a first-order resolution-style theorem prover.

The authors of [4] present a translation of UML class diagrams including general class hierarchies, but excluding attributes and OCL constraints, to OPL [18]. They split the process of finding a model into two stages: First, *finite model reasoning* is employed to find admissible class cardinalities. Then, a finite model is constructed using exactly those numbers of objects. Boolean arrays whose dimensions correspond to the total number of objects are used to encode class memberships and association links.

ConfSolve [9, 10] is an object-oriented language to specify system configurations. Specifications written in it are translated to MiniZinc and then solved by Gecode. Although this approach addresses a similar problem as we do, its focus is rather restricted. This is mainly because the language supports only fixed numbers of objects, both globally and in associations. In other words, the number of objects of each class and the number of association links are always predetermined.

In [3], a translation of UML class diagrams into the language of ECL^{i}PS^e [1] is presented. The authors' main motivation is to verify quality criteria of the object-oriented models, e.g. weak or strong satisfiability. They also support the translation of generalization sets, association classes, and OCL constraints. The

method has been implemented in the tools UMLtoCSP[3] and EMFtoCSP[4]. Although the approach is not limited to ECL^{i}PS^e but also realizable by other constraint programming languages, it is not directly transferable to MiniZinc. This is due to MiniZinc's lack of lists of struct types, i.e. one cannot express association links by lists of object-ID tuples in MiniZinc.

Endeavours similar to OOCSP are also made in the area of Answer Set Programming, there being designated as *OOASP* [7].

The applicability of OOASP and CSP to reconfiguration problems has been studied in [8], where an encoding for associations has been introduced that is also used in the present work. Generative CSP (GCSP) is a variant of CSP that is tailored towards product configuration problems [17]. Some authors also use this formalism in an object-oriented manner [6].

1.2 Motivation

The motivation for OOCSP is manifold. First of all, in our experience object-oriented models are a powerful way to represent complex domains. Most software engineers are well-trained in working with object-oriented models using object-oriented programming languages. On the downside, formal methods for instantiating models are seldom used in the context of object-oriented programming. Constraint programming on the other hand is a powerful declarative paradigm, but the flat data model of constraint variables is hard to maintain if the domain of concern is highly structured and consists of a lot of different interrelated entities.

With OOCSP we have two goals:

1. To show how to use an existing object-oriented model with a constraint language like MiniZinc.
2. To "objectify" existing constraint problems by using an object-oriented model to specify the CSP.

In this paper we will concentrate on the first task. The main reasoning task we want to support is finding an *instantiation* of the object model that satisfies some additional constraints. Other reasoning tasks of interest are *checking* and *completing* an instantiation.

Instantiation: finding a valid instantiation of an object-oriented model.
Checking: checking if a given instantiation is a valid instantiation.
Completing: checking if a partial instantiation can be extended to become a valid one.

As we will see, checking and completing can be achieved by adding constraints describing the given (partial) instantiation to the MiniZinc encoding.

[3] http://gres.uoc.edu/UMLtoCSP/
[4] https://github.com/SOM-Research/EMFtoCSP

1.3 Motivating Example

Our running example is the abstract version of a hardware configuration problem from the domain of railway interlocking systems[5]. Figure 1 shows the UML class diagram of this domain.

Every element of the domain requires a certain number of (control) modules. The modules must be placed inside a frame. A frame must be placed inside a rack. Aside from the constraints implied by the UML class diagram, the following additional constraints hold:

- An *ElementA* requires one *ModuleI*, an *ElementB* requires two *ModuleII*, an *ElementC* requires three *ModuleIII* and an *ElementD* requires four *ModuleIV*.
- A *ModuleV* cannot have an element, all other modules must have an element assigned.
- If a frame contains a *ModuleII*, it must also contain a *ModuleV*.
- All modules of an element must reside in the same frame.
- A *RackSingle* must have exactly 4 frames, a *RackDouble* must have exactly 8 frames.

Instantiations of the class diagram will range from the empty configuration up to a configuration containing one hundred racks and elements, which is enforced by the associations in the class diagram. In a typical configuration scenario, a user asks for a valid configuration containing four *ElementA*. The minimal configuration in terms of number of objects is then a configuration containing four *ElementA*, each assigned to a *ModuleI*, each of which is again assigned to one of the four frames of a *RackSingle*.

Another use case is to find out if there is a configuration with a frame containing six *ModuleII*. In this case the answer is "no", because the constraint that there must be a *ModuleV* in every frame containing a *ModuleII* enforces that the upper bound for *ModuleII* in a frame is 4. This kind of reasoning with cardinalities is very natural and easy to do for human experts. However, as we will see, it is still challenging for a constraint solver.

Of course, real-world problems are much more complex than the simple example described above. Railway systems, for example, often consist of hundreds of different part types. Instantiations of such models contain tens of thousands of configured parts for hardware, software, user interfaces, and communication equipment.

2 Encoding object-oriented models

The most challenging aspect of encoding object-oriented models in a CSP is their dynamic nature. Usually one does not know how many objects will be contained in a solution to the problem[6], whereas in a classical CSP the number of variables and their domains are fixed.

[5] A similar example was also presented in [8].

[6] Generative CSPs [17] capture this dynamic nature of CSPs.

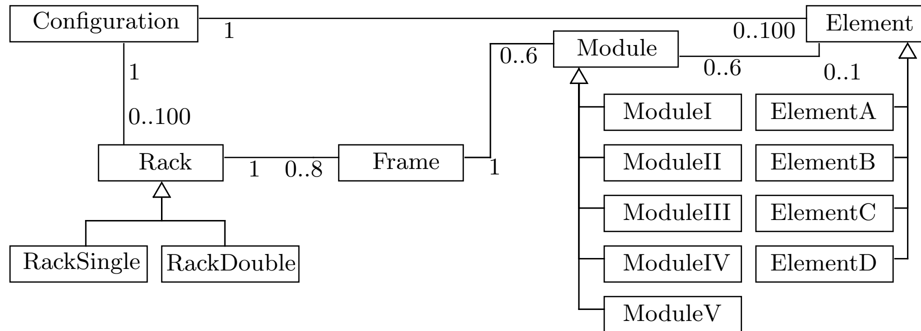

Fig. 1. Running example: Assigning modules to racks

In a manual encoding of a CSP, a knowledge engineer would use special "null" values, optional values, or booleans to deal with the dynamic aspects of the problem.

Listing 1. Warehouse Location Problem[7]

```
1  int: n_suppliers;
2  int: n_stores;
3  int: building_cost;
4  array[1..n_suppliers] of int: capacity;
5  array[1..n_stores,1..n_suppliers] of int: cost_matrix;
6
7  int: MaxCost =
8            max(i in 1..n_stores, j in 1..n_suppliers)(cost_matrix[i,j]);
9  int: MaxTotal = (n_suppliers * building_cost)
10         + sum(i in 1..n_stores, j in 1..n_suppliers)(cost_matrix[i,j]);
11
12 array[1..n_stores] of var 1..n_suppliers: supplier;
13 array[1..n_suppliers] of var bool: open;
14 array[1..n_stores] of var 1..MaxCost: cost;
15 var 1..MaxTotal: tot;
```

A typical example is the warehouse allocation problem[7] in listing 1. From an object-oriented perspective the problem consists of two classes (*Warehouse*, *Store*) and one association (*Store.supplier*). Integers are used to identify different instances (1..*n_stores* and 1..*n_suppliers*). Arrays encode associations (*supplier*) and attributes (*cost*). The number of stores (*n_stores*) is static and given as a parameter. The maximum number of warehouses is also given as parameter (*n_suppliers*), but not every warehouse is necessarily used. A boolean array (*open*) indicates which warehouse is used in the solution.

For our generic encoding we decided to use a global set of object IDs, which ranges from 1 to the maximum number of objects (*MAXNROFOBJECTS*). All arrays encoding the types of the objects, associations and attributes are also defined over this range. This way we can define a generic translation of an object-oriented class hierarchy to a CSP.

[7] The given encoding © Guido Tack, 2007, `http://www.csplib.org/Problems/prob034/models/warehouses.mzn.html`

2.1 File Organization

We partition encodings for object-oriented models into several files as follows:

oocsp.mzn contains the domain-independent OOCSP encoding. This includes the definition of various constants (but not their values), domain-independent constraints, and predicates that can be used in other files to describe a model.

cd.mzn contains the description of a class diagram. This includes assignments to the constants for the number of classes, the class names, and constraints to describe associations. It may also define parameters (constants without values, e.g. for the minimum and maximum class cardinalities). This file is automatically generated from the object-oriented model.

constraints.mzn contains additional constraints that are not expressed in the class diagram. In our running example, this may include the constraint that every *ElementA* is associated to exactly one *Module* which has to be a *ModuleI*.

cc.mzn contains cardinality constraints that can be derived from the constraints of the domain. For instance from the constraint that every *RackSingle* has four objects of type *Frame* associated, we can derive the cardinality constraint `nrofobjects[CLASS_Frame] >= 4 * nrofobjects[CLASS_SingleRack]`.

instance.dzn is a data file which contains arguments for the parameters specified by the model files. This includes the maximum number of objects which is required by `oocsp.mzn`.

2.2 Encoding classes and objects

Classes are the most basic feature of an object-oriented model. These classes are organized in generalization hierarchies. An instantiation of an object-oriented model contains a number of instances of each class, which are called *objects*. Each object shall have a unique ID which we will use later to associate objects to each other.

This leads to the initial MiniZinc encoding in listing 2. The number of classes is given as a constant *NROFCLASSES*. Since we cannot deal with infinite instantiations, an upper bound for the number of objects is given as *MAXNROFOBJECTS*. Every integer in the set 1..*NROFCLASSES* corresponds to a class in the model. To be able to identify these classes, additional constants like `int: CLASS_Element = 2;` etc. (in `cd.mzn`) can be useful. The identifier *CLASS_UNUSED* corresponds to an additional pseudo-class containing unused objects. The sets *CLASS* and *OBJECT* are also present in a second version that includes a special value indicating absence[8].

Listing 2. Classes and objects in MiniZinc (`oocsp.mzn`)

```
1 % how many classes are there
2 int: NROFCLASSES;
```

[8] Absent values could be represented more naturally using *null* values [2] or option types [13].

```
3  set of int : Classes = 1..NROFCLASSES;
4  % unused objects are of type CLASS_UNUSED
5  int : CLASS_UNUSED = 0;
6  set of int : OptClasses = CLASS_UNUSED..NROFCLASSES;
7  % how many objects can be instantiated
8  int : MAXNROFOBJECTS;
9  set of int : Objects = 1..MAXNROFOBJECTS;
10 set of int : OptObjects = 0..MAXNROFOBJECTS;
```

In a first approach to encode class hierarchies, we define for each class its (unique) superclass[9] in the array `superclass`. For our running example, this definition is given in listing 3.

Listing 3. The `superclass` array for our running example (`cd.mzn`)

```
1  array[CLASS] of OPT_CLASS: superclass = [ CLASS_UNUSED, CLASS_UNUSED,
     ↪ CLASS_Element, CLASS_Element, CLASS_Element, CLASS_Element,
     ↪ CLASS_UNUSED, CLASS_UNUSED, CLASS_Module, CLASS_Module,
     ↪ CLASS_Module, CLASS_Module, CLASS_Module, CLASS_UNUSED,
     ↪ CLASS_Rack, CLASS_Rack ];
```

This information is used to construct the two-dimensional boolean `isA` array which specifies for each pair of classes whether the first is a (direct or indirect) descendant of the second one (or if they are just the same). Directly assigning this definition to a constant array (cf. listing 4) fails with a *MiniZinc type error: circular definition of isA*. The same happens if we make the array an array of variables instead of constants.

Listing 4. Approach 1 to encode a class hierarchy (`oocsp.mzn`) (does not compile!)

```
1  array[CLASS,CLASS] of bool: ISA = [ (c1 = c2) \/ SUPERCLASS[c1] = c2 \/
     ↪ exists(c in CLASS)(ISA[c1,c] /\ SUPERCLASS[c] = c2) | c1, c2 in
     ↪ CLASS ];
```

When using a variable array and a universally quantified constraint instead of assigning the array expression directly, it works (cf. listing 5). In this variant, the constraint solver propagates the valid instantiations throughout the array and we obtain the unique correct solution for the problem.

Listing 5. Approach 2 to encode a class hierarchy (`oocsp.mzn`)

```
1  % the generalization hierarchy:
2  array[Classes,Classes] of var bool: isA;
3  array[Classes] of 0..NROFCLASSES: superclass;
4  constraint forall(c1,c2 in Classes)
5    (isA[c1,c2] = ((c1 = c2) \/ superclass[c1] = c2 \/ exists(c in Classes
       ↪ )(isA[c1,c] /\ superclass[c] = c2)));
```

Based on this, we can encode an array assigning types to objects[10], and an `instanceof` predicate, as shown in listing 6.

---

[9] This encoding does not support multiple inheritance.

[10] This encoding does not support non-disjunctive class hierarchies, i.e. an object can not be an instance of two different subclasses of a class. In general, every class can be instantiated in this encoding. If desired, classes can be made abstract by additional constraints.

Listing 6. An array of object types, and an `instanceof` predicate (`oocsp.mzn`)

```
1 array[Objects] of var OptClasses: otype;
2 predicate instanceof(var OptObjects: o, Classes: c) = if (o != 0 /\
    ↪ otype[o] != CLASS_UNUSED) then isA[otype[o],c] else false endif;
```

The solver will decide for each object to which class it belongs, i.e. it will assign elements of `CLASS` to each variable in the `otype` array. In order to make the number of objects variable, the auxiliary class `CLASS_UNUSED` can be assigned to the redundant objects. To know how many instances of each class exist in a solution, the array `nrofobjects` in listing 7 is used.

Listing 7. More information about class instances (`oocsp.mzn`)

```
1 % nr of instances of class
2 array[OptClasses] of var 0..MAXNROFOBJECTS : nrofobjects;
3
4 constraint forall(c in Classes)
5   (nrofobjects[c] = sum(o in Objects) (bool2int(instanceof(o,c))));
```

Symmetry breaking constraints, e.g. the one in listing 8 can also be introduced.

Listing 8. Symmetry breaking for otype (`oocsp.mzn`)

```
1 % symmetry breaking:
2 bool : OTYPEINCREASING;
3 constraint
4   (if (OTYPEINCREASING) then increasing(otype) else true endif);
```

2.3 Encoding associations

Let us turn to binary associations between classes. We investigate three possibilities to encode *one side* of an association:

link For each instance of the first class, there is one variable of type `Objects`. It contains the object ID at the other end of the association link.

set For each instance of the first class, there is one `set of Objects` variable. It contains the object ID(s) at the other end of the association link(s).

ports For each instance of the first class, there are n variables of type `Objects`, where n is the upper bound on the multiplicity of the first class in the association. Each contains one object ID at the other end of an association link.

Each variable can be seen as a *pointer* variable that either points to another object or nowhere. In the latter case, it contains the special value 0 (the least element of the `OptObjects` set).

The three possible encodings can be combined arbitrarily between two classes. That means that one class can use either of the three and the other can use either of the three or none at all. Each combination leads to different implications on the rest of the encoding, and possibly also on solving performance. Table 1 gives an overview over the expressive power of the encodings: some are able to express

Table 1. Possible encodings of a binary association

1st class	2nd class	Expressive power
link		1:n
set		m:n
ports		m:n
link	link	1:n
link	set	1:n
link	ports	1:n
set	set	m:n
set	ports	m:n
ports	ports	m:n

general m:n associations, while some are restricted to the 1:n case. The table can, of course, be read from left to right as well as from right to left. The encoding with *links* on the n side of the association is also used in [8].

Our goal is to encode domain-independent predicates in `oocsp.mzn` that can be used to specify the associations in a class diagram in `cd.mzn`. For example, listing 9 describes the association between *Rack* and *Frame*.

Listing 9. The association between *Rack* and *Frame* (`cd.mzn`)

```
1 % Association Frame_rack(1,1) <-> Rack_frames(0,8)
2 array[OBJECT] of var OPT_OBJECT: Frame_rack;
3 array[OBJECT] of var 0..MAXNROFOBJECTS: Rack_frames_count;
4 constraint hasAssocLink(CLASS_Frame,1,1,Frame_rack,CLASS_Rack,0,8,
     ↪ Rack_frames_count);
```

One rack has between 0 and 8 frames, and each frame belongs to exactly one rack. This is a 1:n association, which can be modelled by all proposed encodings. In listing 9, the variant using *link* on one side is encoded by the `Frame_rack` array. Every element of this array contains a pointer from *Frame* to *Rack*. Additionally, each element of `Rack_frames_count` contains the number of frames associated to a rack. This is done to ease the formulation of cardinality constraints. The implementation of the `hasAssocLink` predicate is given in listing 10.

Listing 10. The `hasAssocLink` predicate (`oocsp.mzn`)

```
1  % define N-1 assoc with link
2  predicate hasAssocLink(
3    Classes: CLASS1, 0..1: MIN1, 1..1: MAX1,
4    array[Objects] of var OptObjects: role1,
5    Classes: CLASS2, int: MIN2, int: MAX2,
6    array[Objects] of var 0..MAXNROFOBJECTS: count_role2,
7  ) = (if MIN1 = 0 then hasAssocLink01(CLASS1,role1,CLASS2) else
       ↪ hasAssocLink1(CLASS1,role1,CLASS2) endif)
8    /\ hasAssocCardinality(role1,count_role2,CLASS2,MIN2,MAX2);
9
10 % defines N-0/1 assoc for class <CLASS> to class <LINKCLASS>
11 predicate hasAssocLink01(Classes: CLASS, array[Objects] of var
       ↪ OptObjects: linkarray, Classes: LINKCLASSES) =
12   forall(o in Objects)
13     (if instanceof(o,CLASS) then linkarray[o] = 0 \/ instanceof(
           ↪ linkarray[o],LINKCLASSES) else linkarray[o] = 0 endif);
14
15 % defines N-1 assoc for class <CLASS> to class <LINKCLASS>
```

```
16 predicate hasAssocLink1(Classes: CLASS, array[Objects] of var OptObjects
     ↪ : linkarray, Classes: LINKCLASSES) =
17   forall(o in Objects)
18     (if instanceof(o,CLASS) then instanceof(linkarray[o],LINKCLASSES)
         ↪ else linkarray[o] = 0 endif);
```

The other encodings can be implemented similarly.

2.4 Encoding attributes

We provide the predicate `hasAttribute`, which allows to define an integer attribute for a class whose value must lie between certain bounds. Listing 11 shows the implementation of this predicate.

Listing 11. The `hasAttribute` predicate (`oocsp.mzn`)

```
1 % defines attribute for class <CLASS>
2 predicate hasAttribute(Classes: CLASS, array[Objects] of var int:
     ↪ attributearray, int: min, int: max) =
3   forall(o in Objects)
4     (if instanceof(o,CLASS) then attributearray[o] >= min /\
         ↪ attributearray[o] <= max else attributearray[o] = 0 endif);
```

2.5 Encoding constraints

Additional constraints can be modelled quite naturally using the standard MiniZinc features and additional predicates offered by *OOCSP*. Listing 12 shows a selection of what is possible.

Listing 12. Selected constraints (`constraints.mzn`)

```
1 constraint forall (o in OBJECT where instanceof(o,CLASS_ElementA)) (
     ↪ hasNObjects(o,Module_element,CLASS_Module,1,1));
2 constraint forall (o in OBJECT where instanceof(o,CLASS_ElementA)) (isA(
     ↪ o,Module_element,CLASS_ModuleI));
3 constraint forall(o1 in OBJECT where instanceof(o1,CLASS_ModuleII))(
     ↪ exists (o2 in OBJECT) (instanceof(o2,CLASS_ModuleV) /\
     ↪ Module_frame[o1]=Module_frame[o2]));
```

The first constraint in listing 12 enforces that every *ElementA* is associated to exactly one *Module*, while the second one enforces this *Module* to be a *ModuleI*. Because of the third constraint, for every *ModuleII* there must be a *ModuleV* in the same frame.

2.6 Reasoning

The most important reasoning task from a practical perspective is to find a valid instantiation under domain-specific constraints.

Finding Instantiations If no additional constraints are defined an *OOCSP* program for a domain will produce all instantiations of the domain within the given scope. The scope is limited by the maximum number of objects that can be instantiated (*MAXNROFOBJECTS*) and by minimum and maximum cardinality restrictions for each class *CNAME* (*CLASS_CNAME_MIN* / ... *MAX*).

Listing 13 shows a solution with `MAXNROFOBJECTS = 10`, `CLASS_ElementA_MIN = 1`, and `CLASS_ElementA_MAX = 1`. Every solution to this problem contains exactly one *ElementA*.

If the program is unsatisfiable, then there is no solution within the given scope or there is an inconsistency in the model. According to the small scope hypothesis [12] most bugs or modelling errors can be found in small scopes, therefore *OOCSP* should also be usable for detecting modelling errors.

Once a solution has been found it is easy to create an object-oriented instantiation from it. For every entry in the *otype* array that differs from 0 (i.e. *UNUSED*) create an instance of the given type and use the index of the entry as temporary object ID. After all instances have been created set the attributes and associations according to the arrays of the MiniZinc solution using the temporary object IDs as keys. For example in listing 13 for `otype[5]=7` a Frame object with temporary ID 5 will be created and associated with a *RackSingle* because of `Frame_rack[5]=10` and `otype[10]=16`.

Listing 13. Sample output of *OOCSP* for the racks example (`solution.txt`)

```
1  int : CLASS_Configuration = 1;
2  int : CLASS_ElementA = 3;
3  int : CLASS_Frame = 7;
4  int : CLASS_ModuleI = 9;
5  int : CLASS_RackSingle = 16;
6
7  otype = [0, 0, 1, 3, 7, 7, 7, 7, 9, 16]
8  nrofobjects = [0, 1, 1, 1, 0, 0, 0, 4, 1, 1, 0, 0, 0, 0, 1, 0, 1]
9  Configuration_configtype = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
10 Element_configuration = [0, 0, 0, 3, 0, 0, 0, 0, 0, 0]
11 Frame_rack = [0, 0, 0, 0, 10, 10, 10, 10, 0, 0]
12 Module_frame = [0, 0, 0, 0, 0, 0, 0, 0, 5, 0]
13 Module_element = [0, 0, 0, 0, 0, 0, 0, 0, 4, 0]
14 Rack_configuration = [0, 0, 0, 0, 0, 0, 0, 0, 0, 3]
```

Checking Instantiations If we want to check an object-oriented instantiation, we must first create an *OOCSP* representation of the object-oriented instantiation i.e. create an *otype* array with an entry for every object in the instantiation and generate the arrays for attributes and association accordingly. By adding the *OOCSP* representation as constraint to the MiniZinc program the consistency of the instantiation can then be checked.

Completing Instantiations For completing a partial instantiation the partial configuration is expressed as additional constraints. The solution of the MiniZinc program will then be a completion of the partial configuration. For example if the partial configuration consists of a *ModuleI* that is assigned to a *Frame* in

a *RackSingle* this can be expressed as the constraint `otype[1]=9` ∧ `otype[2]=7` ∧ `otype[3]=16` ∧ `Module_frame[1]=2` ∧ `Frame_rack[2]=3`.

For this to work some symmetry breaking constraints might have to be deactivated. For example, the constraint in listing 8 that forces the object types to be increasing cannot be used in this example because it would eliminate valid solutions.

3 Evaluation

To evaluate the different encodings in terms of practical utility, we conducted a number of experiments using instances from different problem domains[11].

All tests where run on a Windows PC with an Intel Core i5 CPU @ 2.70GHz and 16 GB RAM. In one experiment we used the running example and created 100 test cases, each fixing the number of instances of one class and leaving the other classes unrestricted. Listing 14 shows a typical example of a test case with three Elements.

Listing 14. Racks example: up to 30 objects, exactly three elements (`oocsp_racks_030_element_003.dzn`)

```
1 MAXNROFOBJECTS = 30;
2 CLASS_Configuration_MIN = 0;
3 CLASS_Configuration_MAX = MAXNROFOBJECTS;
4 CLASS_Element_MIN = 3;
5 CLASS_Element_MAX = 3;
6 CLASS_ElementA_MIN = 0;
7 CLASS_ElementA_MAX = MAXNROFOBJECTS;
```

We used the Gecode solver to run the test cases with the standard heuristic of the Gecode solver and with a domain specific heuristic tuned to find a configuration with a minimum number of objects. Three different association encodings were used: *link*, *set*, and *port*, i.e. the first three entries in Table 1. The figures show the number of failures as reported by Gecode, when solving the particular testcase[12]. For easier comparison data points are only shown for testcases that could be solved by all three encodings within the ten-minute time frame. The number of failures are also an indicator for the quality of the constraint model. As the domain is rather simple, most work should be done by constraint propagation and not by search.

Overall the link encoding could solve the largest percentage of test cases within a ten-minute time frame. Fig. 2 compares the number of failures for test cases that where solvable by all three encodings. The port encoding was the most stable one, whereas the link encoding had some outliers. Interestingly under the domain-specific heuristic, shown in Fig. 3, the picture is almost reversed, because

[11] The encodings that were used for the experiments can be obtained from the authors via e-mail.

[12] The failures can be roughly seen as backtracks required until a solution is found and correlate with the runtime. In contrast to runtime it is not dependent on the computer the experiments were conducted on.

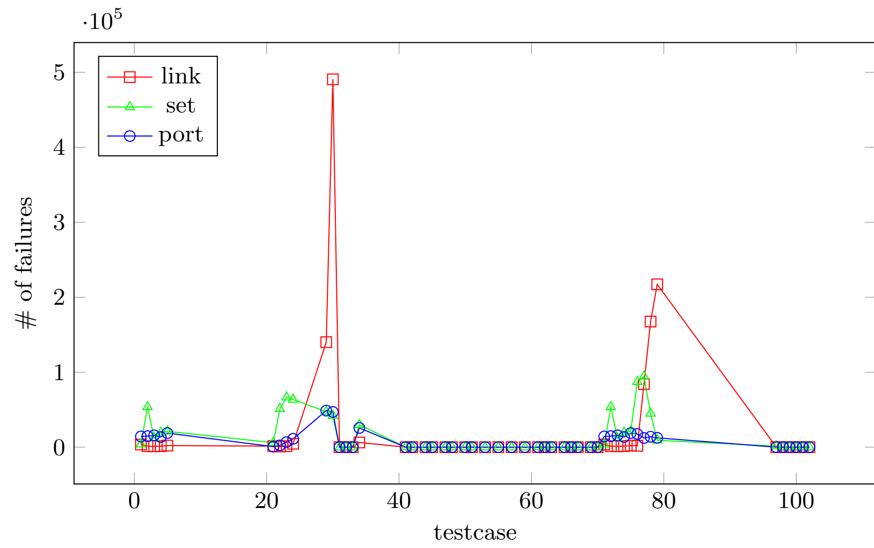

Fig. 2. Number of failures in different association encodings with standard heuristic

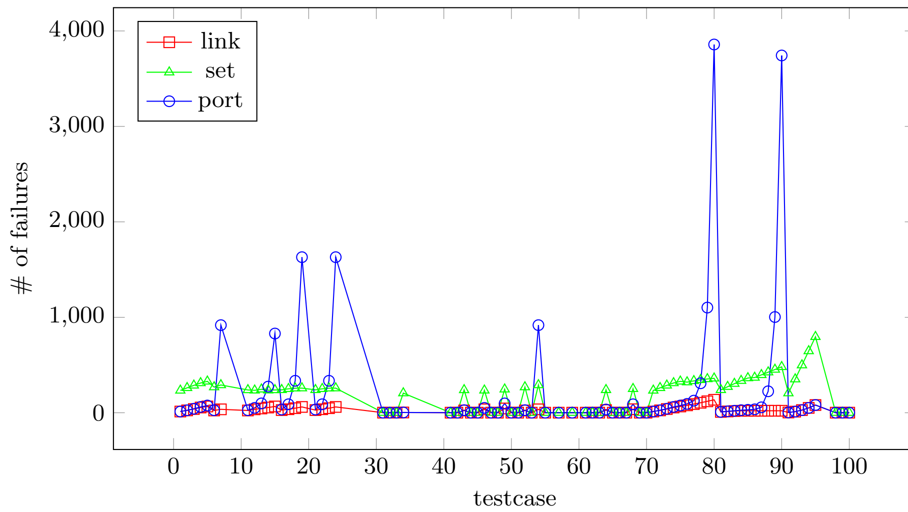

Fig. 3. Number of failures in different association encodings with domain-specific heuristics

the link encoding is the most stable. The set encoding behaves similarly in both cases. With the domain specific heuristic more instances could be solved overall.

The same data is shown in a different incarnation in Fig. 4. For each type of association encoding we show two box plots: in the left part using the default search strategy and in the right part using the custom one. Each box plot shows, from bottom to top: the minimum, the first quartile, the median[13], the third quartile, and the maximum. Please note the different scales in the two graphs. Again, it can clearly be seen that the link encoding leads to the most extreme numbers of failures in the case where the default heuristics are used. In the case of custom search strategies, the highest numbers of failures occur in the port encodings.

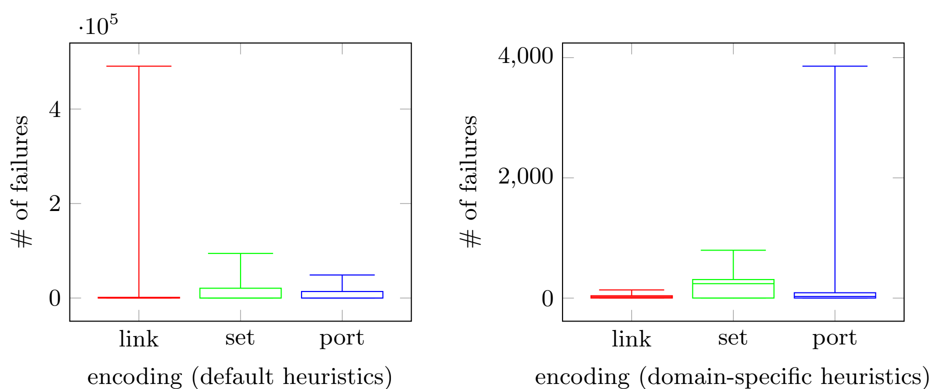

Fig. 4. Box blots for numbers of failures in different encodings

Looking at each test case individually, we found that often a human could come up with a better search strategy for that particular problem instance. This is not that surprising as a human expert can solve these test cases in a "backtrack-free" manner after some deliberation.

Fig. 5 shows the size of the generated flatzinc files. Here the link encoding is the most efficient one, but by looking at the sizes of the flatzinc files it is clear that the encoding can only be used for small to medium scale problem sizes (up to one hundred objects). Even for the link encoding solving the flatzinc file with `MAXNROFOBJECTS = 70` uses more than 1GB main memory.

Similar problems occur in domains with a lot of classes, attributes and associations. We experienced that a real-world configurator domain with 173 classes could be solved, but surprisingly the solver ran out of memory while processing the output statement.

[13] Where the median cannot be seen, it coincides with one of the other quartiles.

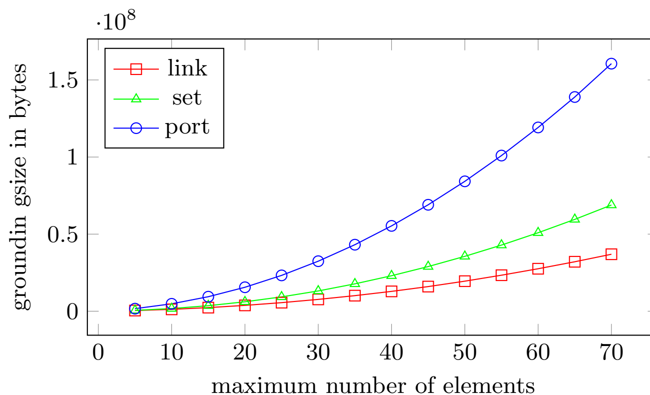

Fig. 5. Size of generated flatzinc file

4 Conclusion

We showed how to encode object-oriented models in MiniZinc in a generic manner. This encodings can be used for medium sized problems (up to one hundred instances), depending on the complexity of the domain. Paradoxically, it seems easier to solve difficult domains with few objects than to solve very simple domains with many objects.

It is clear that these encoding can not compete with special encodings done by a skilled knowledge engineer. One additional problem is the need to use different heuristics based on the nature of the input parameters.

4.1 Future Work

The greatest problem for applying this approach in practice is the grounding size. Therefore we need to explore some form of lazy grounding. For a thorough discussion of the topic see [5]. Besides that we'll have to investigate how to derive as much cardinality information as possible from the constraints of the domain.

For the future we plan to "objectify" existing standard CSP problems and compare the performance of the OOCSP version to the hand-coded problems to gain insight how to improve MiniZinc code generated for OOCSP. Maybe it is possible to define new global constraints for the special struture of object-oriented models.

Finally and perhaps most challenging is to find a way to automatically derive heuristics or solver parameters based on the current problem, as we don't want to devise a new heuristic every time a constraint is added.

Acknowledgements. This work was funded by the Austrian Research Promotion Agency within the project HINT (Heuristic Intelligence) under grant number 840242.

References

1. Apt, K.R., Wallace, M.: Constraint Logic Programming Using ECL^{i}PS^e. Cambridge University Press, Cambridge (2006), http://araku.ac.ir/~d_comp_engineering/88892/1323163/Eclipse.pdf
2. Caballero, R., Stuckey, P.J., Tenorio-Fornés, A.: Two type extensions for the constraint modeling language MiniZinc. Science of Computer Programming 111, 156–189 (2015)
3. Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. Journal of Systems and Software 93, 1–23 (2014)
4. Cadoli, M., Calvanese, D., de Giacomo, G., Mancini, T.: Finite Satisfiability of UML class diagrams by Constraint Programming. In: Barták, R., Junker, U., Silaghi, M.C., Zanker, M. (eds.) Proceedings oft the CP 2004 Workshop on CSP Techniques with Immediate Application. pp. 8–23 (2004)
5. de Cat, B., Denecker, M., Stuckey, P., Bruynooghe, M.: Lazy Model Expansion: Interleaving Grounding with Search. Journal of Artificial Intelligence Research 52, 235–286 (2015)
6. Falkner, A.A., Haselböck, A.: Challenges of Knowledge Evolution in Practice. AI Communications 26(1), 3–14 (2013)
7. Falkner, A.A., Ryabokon, A., Schenner, G., Shchekotykhin, K.: OOASP: Connecting Object-Oriented and Logic Programming. In: Calimeri, F., Ianni, G., Truszczynski, M. (eds.) Logic Programming and Nonmonotonic Reasoning. Lecture Notes in Computer Science, vol. 9345, pp. 332–345. Springer International Publishing, Cham (2015)
8. Haselböck, A., Schenner, G.: A Heuristic, Replay-based Approach for Reconfiguration. In: Tiihonen, J., Falkner, A.A., Axling, T. (eds.) Configuration Workshop. CEUR Workshop Proceedings, vol. 1453, pp. 73–80 (2015)
9. Hewson, J.A., Anderson, P.: Modelling System Administration Problems with CSPs. In: Rendl, A., Beck, J. (eds.) Constraint Modelling and Reformulation (ModRef'11) (2011)
10. Hewson, J.A., Anderson, P., Gordon, A.D.: A Declarative Approach to Automated Configuration. In: Rowland, C. (ed.) Proceedings of LISA '12: 26th Large Installation System Administration Conference. USENIX Association, Berkeley, Calif. (2012)
11. Hinrichs, T., Love, N., Petrie, C., Ramshaw, L., Sahai, A., Singhal, S.: Using Object-Oriented Constraint Satisfaction for Automated Configuration Generation. In: Sahai, A., Wu, F. (eds.) Utility Computing. Lecture Notes in Computer Science, vol. 3278, pp. 159–170. Springer, Berlin Heidelberg (2004)
12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge, Mass., revised edn. (2012)
13. Mears, C., Schutt, A., Stuckey, P.J., Tack, G., Marriott, K., Wallace, M.: Modelling with Option Types in MiniZinc. In: Simonis, H. (ed.) Integration of AI and OR Techniques in Constraint Programming. Lecture Notes in Computer Science, vol. 8451, pp. 88–103. Springer International Publishing, Cham (2014)
14. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a Standard CP Modelling Language. In: Bessière, C. (ed.) Principles and Practice of Constraint Programming – CP 2007. Lecture Notes in Computer Science, vol. 4741, pp. 529–543. Springer, Berlin, Heidelberg (2007)
15. Paltrinieri, M.: Integrating objects with constraint-programming languages. In: Bertino, E., Urban, S. (eds.) Object-Oriented Methodologies and Systems. Lecture

Notes in Computer Science, vol. 858, pp. 248–265. Springer, Berlin Heidelberg (1994)

16. Paltrinieri, M.: Some Remarks on the Design of Constraint Satisfaction Problems. In: Borning, A. (ed.) Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 874, pp. 190–195. Springer, Berlin, Heidelberg (1994)
17. Stumptner, M., Friedrich, G.E., Haselböck, A.: Generative Constraint-Based Configuration of Large Technical Systems. AI EDAM (Artificial Intelligence for Engineering Design, Analysis and Manufacturing) 12(4), 1–27 (1998)
18. van Hentenryck, P.: The OPL optimization programming language. MIT Press, Cambridge, Mass. and London (1999)