# **Testing Object-Oriented Configurators With ASP**<sup>1</sup>

Andreas A. Falkner and Gottfried Schenner<sup>2</sup>, Gerhard Friedrich and Anna Ryabokon<sup>3</sup>

**Abstract.** Testing is an important aspect of every software project. For configurator applications it is equally important but often neglected. This paper shows how to support testing object-oriented and constraint-based configurators by automatically generating positive and negative test cases using answer set programming (ASP). The object-model of the configurator is mapped to ASP code; the constraints to be tested are coded redundantly in ASP. Based on that, the ASP solver generates appropriate test cases, which are then used for unit testing in the object-oriented configurator. There are different strategies to improve this basic process, e.g. reduction of the number of test cases with symmetry breaking.

## 1 Introduction

Testing is an important but often neglected aspect of every software development project. Especially for object-oriented (OO) languages, unit testing with a testing framework like JUnit [3] is well established and an integral part of development methods like Extreme Programming [4]. Unit testing frameworks are also gaining acceptance outside of object-oriented programming [9].

A configurator is a software system that enables the user to configure complex systems or services using predefined components. In a constraint-based configurator, constraints describe the conditions which the configured system must satisfy. In order to test the correctness of each individual constraint, the tester must provide positive and negative test cases for it. A positive (negative) test case is a partial configuration where the constraint is satisfied (violated). Obviously, the test cases cannot be created by the solver of the configurator because one cannot use the possible faulty constraint to generate the test case. Therefore, the test cases currently must be created manually.

There are different testing strategies such as black-box and white box testing ([16]. In black-box testing, the internal structure of the test object must not be known to the tester and the tests are devised according to the specification of the software system. In white-box testing, the internal structure is known and the tester designs the tests to achieve a high test coverage. In practise both strategies should be used because they tend to find different kind of errors.

The basic idea of this paper is to semi-automatically generate test cases for object-oriented configurators by first translating the configurator's knowledge base (without the constraints to be tested) to an answer set programming (ASP) program. The constraints to be tested are then coded manually in ASP. Implementing the same constraint both in Java and ASP achieves the necessary diversity to detect conceptional errors (similar to N-Version programming [1]). The ASP solver runs this program and generates positive and negative test cases which are translated back into test cases for the object-oriented

The following section defines necessary features of the configurator and provides a brief introduction to the ASP systems. In Section 3 we describe the approach in more details presenting the OO-ASP mapping and examples for a small application. We show different ways to reduce the number of generated test cases to a reasonable size in Section 4 and conclude in Section 5.

## 2 Context

configurator.

For this work, we used a configurator based on Generative Constraint Satisfaction (GCSP) which is a combination of object-oriented and constraint-based technologies. In general however, any system that complies to the definition of the following subsection can be used. The current target system is the Potassco ASP suite<sup>4</sup> [12] - it could easily be replaced by another ASP system.

#### 2.1 Object-oriented constraint-based configurator

The results of this paper can be applied to any existing configurator framework which complies to the following definitions.

**Definition 1 (Knowledge Base, KB)** The knowledge base of an object-oriented and constraint-based configurator comprises an object model and a set of constraints.

The KB specifies the relevant domain knowledge in a declarative way. The solver comprises a general constraint solver which reasons over that knowledge, e.g. checks consistency, searches solutions (i.e. valid configurations), etc.

**Definition 2 (Object Model)** An object model contains classes, their inheritance hierarchy, attributes (Boolean, enumeration, integer), and associations (bidirectional).

The object model describes the structure of the possible configurations, including the multiplicities (cardinalities) of the parts. It can be specified by an UML class diagram [17].

**Definition 3 (Configuration)** A configuration is an instantiation of the object model.

Without loss of generality, only instances of leaf classes (classes without subclasses) are allowed in a configuration. For the course of this paper, it is assumed that the configurator maintains one current configuration. In an interactive configurator, the user would manipulate the current configuration by adding/deleting objects and setting

<sup>&</sup>lt;sup>1</sup> This work has been developed within the scope of the project RECONCILE (reconciling legacy instances with changed ontologies) and was funded by FFG FIT-IT (grant number 825071).

<sup>&</sup>lt;sup>2</sup> Siemens AG Österreich, Vienna, Austria firstname.{middleinitial.}lastname@siemens.com

<sup>&</sup>lt;sup>3</sup> Universität Klagenfurt, Austria firstname.lastname@aau.at

<sup>&</sup>lt;sup>4</sup> http://potassco.sourceforge.net

attributes and associations until a valid configuration is found. Alternatively the constraint solver can be used to extend a configuration to a valid configuration. Constraints are used to describe the valid configurations of the configurator.

**Definition 4 (Constraint)** A constraint is a condition which every valid configuration must satisfy.

This is a very general definition of the concept constraint. To make our approach broadly applicable, no special constraint techniques like domain-filtering, constraint propagation, etc. are required. A constraint can be thought of as an invariant constraint in UML/OCL. In its simplest form, constraints are Boolean methods of an objectoriented language defined over the current configuration. From a knowledge engineering view, constraints should correspond to some requirements that the product to configure must satisfy. The scope of a constraint can range from simple expressions like 'wheel1.size = wheel2.size' to 'The light-system of this vehicle is configured correctly' (represented by some complex code accessing sub-parts and their properties).

## 2.2 Answer Set Programming

Answer set programming is an approach to declarative problem solving which has its roots in logic programming and deductive databases. This is a decidable fragment of first-order logic extended with default negation, aggregation and weight constraints. ASP allows modeling of a variety of search and optimization problems in a declarative way [13, 7, 5] using model-based problem specification methodology. Efficient ASP solvers allow fast identification of solutions that correspond to answer sets of a program. Recent examples include areas such as molecular biology, decision support and planning. The DLV system [15] was used to plan shifts at Gioia-Tauro Seaport which reduced the time required to define working teams' assignments from hours to just a few minutes. A Potassco [12] program is able to detect inconsistencies in large biological networks.

Since configuration problems are a type of combinatorial (optimization) problems, ASP was used by Soininen et al. [18] in their approach which was one of the earliest industrial applications of ASP. This first approach to the configuration problem was extended by Friedrich et al. [10] to both configuration and reconfiguration cases. Recently, Gebser et al. [11] have suggested a novel ASP based modeling approach to configuration support of a Linux package management system.

This work uses the following language constructs of Potassco (similar constructs are available in DLV):

- · constant: lower-case string or number
- variable: upper-case string or \_
- predicate:  $predicatename(A_1, \ldots, A_n)$  with each  $A_i$  being a constant or variable
- condition: *P* : *C* (with P and C being predicates) generating a set of ground instances for P corresponding to the existence of ground instances of C
- (counting) aggregate:  $L\{A_1, \ldots, A_n\}U$  (with L being a lower bound, U an upper bound, and each  $A_i$  a predicate possibly generated by a condition) stating that the number of ground instances  $A_i$  shall be within the bounds
- fact:  $A_0$ . with  $A_0$  being a predicate
- rule:  $A_0:-L_1, \ldots, L_n$ . with  $A_0$  and  $L_i$  being predicates or aggregates,  $L_i$  possibly negated
- constraint: :- $L_1, \ldots, L_n$ . with  $L_i$  being predicates or aggregates, possibly negated

#### **3** Test case generation

Figure 1 shows the main use-case of our approach. To generate test cases for a specific constraint, one identifies the fragment of the object model relevant for the constraint. Using a generic OO-ASP mapping, described in the next section, this fragment is translated into an ASP program capable of enumerating all (up to a given upper bound) instantiations of the object model i.e. all possible configurations.



Figure 1. Generation of test cases

Since the main purpose of our approach is to detect conceptual errors, the tester has to reimplement the constraint to be tested in ASP, based on the requirements describing the constraint. Although possible, one cannot automatically translate the constraint from the OO configurator to ASP because an automatic translation would also translate the errors in the constraint. For the same reason the tester should be unaware of the implementation of the constraint in the object-oriented configurator. This process implements a black-box testing strategy like in traditional software engineering.

The generated ASP code and the ASP definitions for the constraint are used to compute answer sets that represent positive and negative test cases. These answer sets are then translated back into an objectoriented configuration and used in unit tests for the constraint.

## 3.1 OO-ASP Mapping

To illustrate the approach, a simple example domain for configuring bicycles (Figure 2) is used. A bicycle has a frame, two wheels and optional lights. A possible configuration can contain multiple bikes of different types, wheel sizes, etc. A valid configuration consists of a collection of correctly configured bicycles as defined by the allowed domains of the attributes (e.g. type), the given cardinalities of the associations (e.g. 0..1 for the lights), and two explicit constraints:

- constraintWheelsize disallows wheels of different sizes.
- constraintLights is complexer and requires that city bikes have lights, that racing bikes do not have lights, that mountain bikes may only have battery lights, and that the Boolean attribute hasLights must correspond to the existence of a Lights instance.

The object model is mapped to ASP according to the following schema:

 Every class C is mapped to two unary predicates <aspnameC>(X) and <aspnameC>Domain(X). The domain predicates are needed to describe the possible instances of a



Figure 2. UML-diagram for bikeshop example

class - similar to variables to be activated in conditional constraint satisfaction problems. The maximal number of instances is defined manually via predicate <aspnameC>MaxInstances(X). Instances are identified by integers values.

- Every attribute ATTR of class C is mapped to a binary predicate <aspnameATTR>(X,Y) where X is an integer representing an instance of class X and Y is a possible value of attribute ATTR.
- Every association ASSOC between class C1 and C2 is mapped to a binary predicate <aspnameASSOC>(X,Y), where X and Y are integers representing instances of class C1 and C2.

The mapping is controlled by an XML file. It can be used to ignore irrelevant information, e.g. the attribute type of the frame. The following excerpt shows those parts of the mapping which are needed for constraintWheelsize.

```
<classmapping>
    <javaname>bikeshop.kb.Bicycle</javaname>
    <aspname>bicycle</aspname>
    . . .
    <assocmapping>
        <javaname>wheels</javaname>
        <javaotherclass>bikeshop.kb.Wheel
        </javaotherclass>
        <aspname>bicycle2wheel</aspname>
    </assocmapping>
</classmapping>
<classmapping>
    <javaname>bikeshop.kb.Wheel</javaname>
    <aspname>wheel</aspname>
    <attrmapping>
        <javaname>size</javaname>
        <aspname>wheelSize</aspname>
    </attrmapping>
</classmapping>
```

By this mapping the Java class Bicycle is mapped to the unary predicate bicycle, class Wheel to predicate wheel, its attribute size to the binary predicate wheelSize, and the association between Bicycle and Wheel to the binary predicate bicycle2wheel.

Example of generated facts (for the listed part of the mapping):

```
bicycleMaxInstances(1).
bicycleDomain(1).
wheelMaxInstances(3).
wheelDomain(201).
```

wheelDomain(202).
wheelDomain(203).

Examples of user-defined maximum of instances and of facts generated by the solver as part of a test case like in Figure 5:

```
bicycle(1).
bicycle2wheel(1,201).
bicycle2wheel(1,202).
wheel(201).
wheelSize(201,24).
wheel(202).
wheelSize(202,25).
```

In order to be able to enumerate every possible configuration, the following additional ASP code is generated:

- 1. For every class C, the instances up to the given maximal number are generated by:
  - 0{<aspnameC>(X):<aspnameC>Domain(X)}MAX :-<aspnameC>MaxInstances(MAX).

#### Example:

- 0{bicycle(X):bicycleDomain(X)}MAX :bicycleMaxInstances(MAX).
- 2. For every attribute ATTR of class C and possible values V1..Vn, one rule is needed to ensure exactly one value:

```
1{<aspnameATTR>(X,V1),...,
<aspnameATTR>(X,Vn)}1 :- <aspnameC>(X).
```

#### Example:

3. For every association ASSOC between C1 and C2 and cardinality restrictions L.U, a rule is generated for the lower bound:

The upper bound of the association is checked with a constraint:

Example (upper bound = lower bound = 2):

2{bicycle2wheel(X,Y):wheelDomain(Y)} :bicycle(X).

4. Especially for big domains, some basic symmetry breaking constraints are required to avoid explosion of the number of generated test cases. Since the instances of a class are interchangeable we disallow usage of instances with a higher ID unless all instances with a lower ID are used as well:

```
:- <aspnameC>Domain(X), <aspnameC>Domain(Y), X<Y, <aspnameC>(Y), not <aspnameC>(X).
```

Example:

```
:- wheelDomain(W1), wheelDomain(W2), W1<W2, wheel(W2), not wheel(W1).
```

With this mapping it is possible to enumerate all configurations up to the given upper bound of the number of instances (preferring instances with a lower ID). The mapping is also used to translate an answer set back into a configuration of the object-oriented configurator. E.g. for the term bicycle(1) an instance of class Bicycle is created, for bicycle2wheel(1,201) the objects for bicycle with id 1 and the wheel with id 201 are associated, etc.

If the generated program does not have an answer set (unsatisfiable) then the object model itself is inconsistent. The UML class diagram in Figure 3 shows an example of an inconsistent object model.



Figure 3. UML-diagram for inconsistent model

For every instance of class A, two instances of B and three instances of C must exist. Since there is a 1-1 association between B and C this class diagram is inconsistent. In this case the testing system reproduces the functionality of an earlier method [8] that uses integer programming for automatic detection of inconsistencies in UML class diagrams.

#### **3.2** Test cases for constraints

To test a constraint, the tester needs to implement the constraint in ASP using the predicates of the generic mapping. With the generated program code of the preceding section and the manually written constraint, an ASP solver can find answer sets which satisfy the constraint or violate the constraint (counterexamples). By that, we get a set of test cases (represented as partial configurations) for each constraint. This approach is similar to the one supported by Alloy ([14]).

To avoid making the same conceptional errors as the implementer of the OO constraint, the tester should be unaware of the OO constraint code when writing the constraint. The implementer of the ASP constraint is only given a verbal description of the constraint or the requirement that should be checked by the constraint.

As an example, take the constraint that the wheels of a bicycle must have the same size (constraintWheelsize in Section 3.1). Following the convention that ASP constraints specify what is *not* a valid configuration, the tester expresses this with the following ASP code:

```
constraintWheelsize :-
bicycle(X),
```

```
bicycle2wheel(X,W1),
bicycle2wheel(X,W2),
W1!=W2,
wheelSize(W1,S1),
wheelSize(W2,S2),
S1!=S2.
% find positive test case
:- testpositive, constraintWheelSize.
% find negative test case (counterexample)
:- testnegative, not constraintWheelSize.
```

The two atoms *testpositive* and *testnegative* control whether the solver finds positive or negative test cases for the tested constraint. In a positive test case the constraint is satisfied, in a negative one it is violated.

Figure 4 shows a positive test case found by the ASP solver running the program for constraintWheelsize. In this automatically generated graphical representation, rectangles represent instances, ellipses represent values, and the edges are labeled by the predicates between the nodes.



Figure 4. Positive test case

Running the same program with the fact *testnegative* produces the negative test case in Figure 5, i.e. a counterexample for the constraint.



Figure 5. Negative test case

Each answer set represents one test case and can be translated into a partial configuration for the object-oriented configurator. All positive and negative test cases can be used for unit testing the constraint.

Note that the generated partial configurations for positive test cases might violate other constraints of the domain. For instance, constraintLights requires that the attribute bicycleHaslights is true, iff the bicycle has lights. This constraint is violated in the positive test case of Figure 4.

# 3.3 Unit testing

The whole process of test case generation can easily be integrated into unit testing. The test cases are added to the unit test suite of the configurator and used for regression testing. The following code sequence shows the unit test for constraintWheelsize which first runs all positive test cases and then all negative ones. The function generateTestcases executes the ASP solver as described in the preceding section and returns a list of answer sets. The function createConfigurationFor creates the partial configuration for an answer set which can be accessed by getter methods such as getBicyles.

```
public void testConstraintWheelsize() {
  List<Set<String>> tcs;
  tcs = generateTestcases("testpositive");
  for (Set<String> answerSet : tcs) {
    createConfigurationFor(answerSet);
    Bicycle bike = getBicycles().get(0);
    IConstraint c =
    bike.getConstraint(CONSTRAINTWHEELSIZE);
    assertEquals(Boolean.TRUE, c.getVal());
  }
  tcs = generateTestcases("testnegative");
  for (Set<String> answerSet : tcs) {
    createConfigurationFor(answerSet);
    Bicycle bike = getBicycles().get(0);
    IConstraint c =
    bike.getConstraint(CONSTRAINTWHEELSIZE);
    assertEquals(Boolean.FALSE, c.getVal());
}
```

If an assert fails (i.e. a test case reports a discrepancy) then the reason for it has still to be found. For example, consider the following faulty Java implementation of the constraint. Since it returns true for the counterexample, we know there is a discrepancy between the ASP and the OO implementation of the constraint. Looking at the Java code below it is easy to identify the error. Due to a typing error, w2 is never referenced.

```
// in class Bicycle
public boolean constraintWheelsize() {
  List wheels = getWheels();
  if (wheels.size()!=2) { return false; }
  Wheel w1 = wheels().get(0);
  Wheel w2 = wheels().get(1);
  return w1.getSize()==w1.getSize();
}
```

In many cases, comparing the two implementations (i.e. static analysis) is sufficient for identifying an error. If two constraint implementations use different parts of the model this is an indication of an error. For instance, if one constraint depends on an attribute value and the other does not then there is a high chance that the first is more specific than the other.

Note that an OO implementation of the constraint in the configurator is not needed to generate test cases with our approach. Therefore, this method can also be used for the test-first approach of Test Driven Development [2].

## 4 Improving the test cases

Uninformed test case generation as described in the last section creates many possible configurations. Usually, this leads to a good test coverage. However, the number of test cases gets too large for practical use, especially for large-scale configuration.

Therefore, a method is needed to choose test cases which are likely to detect errors in the implementation. To generate test cases with specific properties, the tester can add statements describing those properties to the ASP implementation. For instance, adding

1 { wheelSize(X,Y):Y=24..25 } 1 :- wheel(X).

will only generate test cases where the wheelSize is 24 or 25. Specifying all relevant test cases manually this way is a tedious task. The alternative is to use advanced filtering techniques like symmetry breaking.

# 4.1 Symmetry breaking

For black-box testing in software engineering, techniques such as equivalence partitioning and boundary value analysis have been developed to reduce the number of test cases [16]. These techniques define equivalence classes for the input data and test only one value from every equivalence class.

For constraint-based systems a similar effect can be achieved by defining equivalence classes over the possible configurations by using symmetry breaking techniques. For instance, in the positive test cases for constraintWheelsize, the actual value of the wheel size is irrelevant as long as the values are all the same (assuming a reasonable implementation). For negative test cases, at least two different values are needed, but it does not matter which values are actually chosen.

Detection of the equivalence classes for an ASP program is done by reducing it to the colored graph automorphism problem [6]. In this case, the grounded program is represented as a colored graph. The symmetry breaking tool is searching for such transformations of the graph (permutation) that map vertices of it to vertices of the same color. The coloring schema in [6] allows to identify permutations of graph vertices corresponding to equivalent grounded atoms, e.g. wheels of different sizes, in a program. The permutations are used by the preprocessor SBASS<sup>5</sup> [6] to generate symmetry breaking constraints that introduce a lexicographic order on elements of a solution space. The symmetry breaking constraints are added to the grounded program and the result is forwarded to the solver.

Roughly speaking, in the case of wheel sizes, constraints will require to use wheels of size 20 first, since 20 is lexicographically the smallest value. Only if it is impossible to find a configuration with wheels of the size 20, the solver will try the size 21 and so on.

Inclusion of the preprocessing step in the testing tool chain reduces the number of possible configurations for the bicycle example (without coding the two constraints in ASP) from 1459 to 129. For the test case generation example for constraintWheelsize as described in Section 3.2, execution of SBASS reduces the number of positive test cases from 163 to 13. Although the number of test cases can be reduced drastically by symmetry breaking, one still has to ensure that the coverage of the created test cases is enough to find potential errors.

For instance, consider the case where the knowledge base is modified by allowing bicycles to have more than two wheels (i.e. tricycles, etc). The following faulty constraint implementation works, if

```
<sup>5</sup> http://potassco.sourceforge.net/labs.html
```

the differences in wheel sizes always occur in the first two wheels. If symmetry breaking creates only such test configurations, one can no longer detect the error in the constraint implementation.

```
// faulty implementation, but works
// if the "first" 2 wheels are of same size
public boolean constraintWheelsize() {
  List<Wheel> wheels = getWheels();
  for(int i = 0 ; i<wheels.size() ; i++) {
    for(int j=i+1 ; j<wheels.size() ; j++) {
        if (wheels.get(i).getSize().equals(
            wheels.get(j).getSize())) {
            return true;
        }
    }
    return false;
}</pre>
```

## 4.2 White box testing

The kind of errors in the implementation often depend on the programming language used. A knowledge engineer using a specific Java framework will make different errors than a knowledge engineer using ASP. Therefore, the generation of test cases cannot be fully automated without additional information about likely errors.

By looking at the code (white box testing) an experienced developer can identify suspicious parts of the code which should be tested. From that, she can derive which properties a test case must have and can create such test cases manually. For automated test case generation, it is possible to generate test cases with specific properties by adding additional constraints to the ASP program similar to the example at the beginning of Section 4.

#### 4.3 Maintaining test configurations

By combining all the constraints of the domain, the described approach can also be used to generate complete and valid test configurations for the object-oriented configurator. The limiting factor here is that the performance of the generic mapping and the fact that all constraints of the configurator must be reimplemented in ASP.

Complete test configurations are often used for integration testing, system tests, etc. A common problem is how to maintain the consistency of the test configurations in case of knowledge base evolution. Whenever the requirement of a constraint changes, one needs to reconcile those changes with existing legacy test configurations.

Since many tests may depend on the existing test configuration, the changes in the legacy test configuration should be minimal. An ASP method for finding reconfigurations with minimal costs is suggested in [10].

## 5 Conclusions

We described how to map the object model and configurations of an object-oriented and constraint-based configurator to and from ASP. One application of this mapping is the generation of test cases for the OO configurator. Since the mapping is symmetric it could also be used to generate test cases for an ASP-based configurator.

The generation of test cases so far has been tried for toy examples like the bikeshop domain and some small fragments of real world domains. For the future we plan to evaluate translation of existing knowledge bases of our real-world configurators (>100 classes) into ASP. We expect that we have to refine the techniques of Section 4 in order to get sufficient performance.

The current approach cannot be used for test cases containing a lot of components. For instance, the bikeshop domain already uses more than 1GB of memory if the domain size is set to more than 50. Fortunately, test cases for single constraints usually do not involve hundreds of components.

## REFERENCES

- Algirdas A. Avizienis, *Software Fault Tolerance*, volume 2, chapter "The Methodology of N-Version Programming", 22–45, John Wiley & Sons, 1995.
- Beck, *Test Driven Development: By Example*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [3] Kent Beck. JUnit, 2010.
- [4] Kent Beck and Cynthia Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*, Addison-Wesley Professional, 2004.
- [5] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski, 'Answer set programming at a glance', *Communications of the ACM*, 54(12), 92–103, (2011).
- [6] Christian Drescher, Oana Tifrea, and Toby Walsh, 'Symmetry-breaking answer set solving', AI Commun., 24(2), 177–194, (2011).
- [7] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner, 'Answer set programming: A primer', in *Reasoning Web*, pp. 40–110, (2009).
- [8] Andreas Falkner, Ingo Feinerer, Gernot Salzer, and Gottfried Schenner, 'Solving practical configuration problems using UML', in *Proceedings* of ECAI 2008 Workshop on Configuration Systems, pp. 1–6, (2008).
- [9] Onofrio Febbraro, Nicola Leone, Kristian Reale, and Francesco Ricca, 'Unit testing in aspide', *CoRR*, (2011).
- [10] Gerhard Friedrich, Anna Ryabokon, Andreas A. Falkner, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner, '(Re)configuration based on model generation', in *LoCoCo*, eds., Conrad Drescher, Ins Lynce, and Ralf Treinen, volume 65 of *EPTCS*, pp. 26–35, (2011).
- [11] Martin Gebser, Roland Kaminski, and Torsten Schaub, 'aspcud: A linux package configuration tool based on answer set programming', in *Lo-CoCo*, pp. 12–25, (2011).
- [12] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub, 'Conflictdriven answer set solving: From theory to practice', *Artif. Intell.*, 187, 52–89, (2012).
- [13] Michael Gelfond and Vladimir Lifschitz, 'The stable model semantics for logic programming', in *5th International Conference and Symposium on Logic Programming*, pp. 1070–1080, (1988).
- [14] Daniel Jackson, 'Alloy: A logical modelling language', in ZB, p. 1, (2003).
- [15] Nikola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello, 'The DLV system for knowledge representation and reasoning', ACM Transactions on Computational Logic (TOCL), 7(3), 499–562, (2006).
- [16] Glenford J. Myers and Corey Sandler, *The Art of Software Testing*, John Wiley & Sons, 2004.
- [17] James Rumbaugh, Ivar Jacobson, and Grady Booch, 'The unified modeling language reference manual', in *The unified modeling language reference manual*, (2005).
- [18] Timo Soininen, Ilkka Niemelä, Juha Tiihonen, and Reijo Sulonen, 'Representing configuration knowledge with weight constraint rules', in 1st International Workshop on Answer Set Programming: Towards Efficient and Scalable Knowledge, pp. 195–201, (2001).