

TDK Report

Author:

Ágnes Barta



M Ű E G Y E T E M 1 7 8 2

Budapest University of Technology and Economics

Faculty of Electrical Engineering and Informatics

Department of Measurement and Information Systems

Abstract Test Data Generation for Autonomous and Distributed Systems

Author:

Ágnes Barta

MSc 2. semester

Advisors:

Zoltán Szatmári
Research Associate

Oszkár Semeráth
PhD Student

András Vörös
Research Associate

Kivonat

Napjainkban kiemelt feladat, hogy a modern szoftverrendszerek helyességét biztosítsuk, valamint a bennük felmerülő tervezési és implementációs hibákat felderítsük. Komplex autonóm és elosztott rendszerek vizsgálata sok kihívást rejt magában. Ennek egyik fő oka, hogy ezen rendszerek és környezetük felépítése idővel változik, ami a lehetséges rendszermodellek számának jelentős növekedését idézheti elő. A szoftvertesztelés egy olyan verifikációs módszer, amelyet széles körben alkalmaznak különféle hibák felderítésére, azonban bonyolult rendszerek ellenőrzése esetén nehéz minden számunkra fontos konfigurációs esetet lefedni. A munkámban az autonóm rendszerek tesztelésének támogatását céloztam meg.

A dolgozatom célja egy olyan módszer kidolgozása volt, amely képes absztrakt tesztadatok generálására. Az általam készített algoritmus a tesztadatok egy olyan halmazát állítja elő, ami felhasználható konfigurációs beállításként, illetve tesztelés alatt lévő rendszerek bemeneteként szolgál. Az általam javasolt módszer képes koncepcionálisan és strukturálisan különböző “absztrakt” tesztadatokat előállítani, amelyek lefedik a rendszer összes lehetséges felépítését. Ezen “absztrakt” tesztadatok annyiban térnek el a hagyományos tesztadatokról, hogy az objektumok attribútumai nem konkrét értékeket tartalmaznak, hanem a strukturális viszonyokat fejezik ki. A módszert egy testreszabható paraméterező nyelv támogatja, aminek a segítségével a tesztkonfigurációk számunkra releváns részeire tudunk összpontosítani.

A dolgozatomban bemutatok egy tesztgeneráló keretrendszert, ami olyan rendszerek tesztelését támogatja, amelyek metamodellel, példánymodellekkel és kényszerekkel definiálhatóak. A tesztgenerálási folyamat a következő lépésekből áll: 1) a rendszer definiálása és a tesztadat kiválasztási kritériumok meghatározása, 2) “absztrakt” tesztadat-objektumok generálása, 3) az objektumok attribútum értékeinek a konkretizálása. A tesztgenerálás támogatásához definiáltam egy nyelvet, amellyel a tesztleírások könnyedén specifikálhatóak. A nyelv sok paraméterezési lehetőséget kínál, melyek segítségével az algoritmus finomhangolható. Az algoritmus bemeneteit elsőrendű logikai problémaként formalizáltam, amely megoldását korszerű logikai következtetők szolgáltatják.

Az elkészült eszközt az Eclipse környezetbe integráltam, ami képes egy metamodell segítségével definiált, OCL (Object Constraint Language) és egyéb konzisztencia kényszerekkel ellátott rendszermodellt, illetve a kezdeti konfigurációt logikai problémává transzformálni. A logikai problémát az Alloy Analyzer képes értelmezni, megoldani és a logikai axiómákat kielégítő modellt generálni. Az eszközöm az Alloy Analyzer által előállított modellt visszaalakítja egy olyan példánymodellé, ami megfelel az eredeti metamodellnek és így később konkrét tesztekhez rendszermodellként felhasználható. A dolgozatomban a módszer egy alkalmazási lehetőségét egy autonóm robotokkal és egy elosztott rendszerekkel foglalkozó esettanulmányon keresztül mutatom be.

Abstract

Nowadays, complex software systems require deep analysis to ensure correctness or to find possible design and implementation flaws. Testing is a widely used verification technique to find these problems. However, the testing of systems like autonomous systems or distributed systems is a challenging task, as their structure evolves with time, thus leads to a potentially infinite number of different system configurations and structures. It is difficult to cover the relevant settings by testing. My work focuses on the complex task of testing distributed autonomous systems.

The first goal is a framework being able to generate abstract test data. The output of the algorithm is a diverse set of test data which is used as possible test configuration settings and as input data of the System Under Test (SUT). My approach is able to produce a wide range of conceptionally different abstract test data which is highly needed to cover the full scale of the possible system behaviours and configurations. A highly customisable parametrisation language supports the focusing on the relevant fragment of test configurations.

In this report I introduce a test generation framework to support the testing of systems described by metamodels, constraints and instance models. The test generation procedure is built from the following steps: 1) the description of the system and the test data selection criteria 2) the generation of various abstract tests 3) the concretization of the abstract tests to structurally different models of the metamodel (interpreted as test data). I introduce a language for describing test goals over arbitrary metamodels. In addition other parameters can also be defined to fine tune the algorithms. I formalized the input artifacts of the algorithm as a First Order Logic problem, which can be solved by advanced logic reasoners.

A prototype tool is successfully integrated to the Eclipse modelling tool, which is able to transform the system description defined in a Metamodel, OCL (Object Constraint Language) invariants, consistency constraints and the initial configurations to a logic problem, which is solved by the Alloy Analyzer. The generated logic models are transformed back to the standard models of the metamodel. Those models can be immediately used as test inputs. I demonstrate the usefulness of the approach with two case studies from the autonomous robotic and the distributed system domains.

Contents

1	Introduction	6
1.1	Problem statement	6
1.2	Research context	6
1.3	Objectives	7
1.4	Contribution	7
1.5	Previous works	8
1.6	Structure of the report	9
2	Background	10
2.1	Test data generation workflow	10
2.2	Mathematical logic	11
2.2.1	First order logic	11
2.2.2	Prover and solver techniques	11
2.3	Modeling background	13
2.3.1	Metamodeling	13
2.3.2	Well-formedness constraints	15
2.4	Partial Snapshot	15
2.5	Related work	17
3	Motivating scenarios and requirements	19
3.1	Test data generation for R3-COP	19
3.1.1	Metamodel	20
3.2	Distributed system domain	21
3.2.1	Metamodel	21

4	Overview of the approach	23
4.1	Test generation process	23
4.2	Input for the test data generation process	24
4.2.1	DSL specification	24
4.2.2	Test data generation parameters	24
4.2.3	Discriminator definition	25
4.2.4	Test criteria description language	26
5	Mapping DSLs to Alloy formulae	28
5.1	Foundation of the mapping	28
5.2	Mapping metamodels	29
5.2.1	Type mapping	29
5.2.2	Type hierarchy	29
5.2.3	References	30
5.2.4	Multiplicity	30
5.2.5	Inverse edges	31
5.2.6	Containment	31
5.2.7	Attributes	32
5.3	Mapping Partial Snapshots	32
5.4	Mapping OCL invariants	34
5.4.1	An overview of OCL mapping	34
5.4.2	Basic expressions	35
5.4.3	Collections	36
5.4.4	Collection operators	37
5.5	Abstract test data generation approach	39
6	Implementation	41
6.1	Architecture	41
6.1.1	Details	42

7	Case-studies	44
7.1	Test data generation for laser guided vehicles	44
7.1.1	Description	44
7.1.2	Initial Partial Snapshot	45
7.1.3	Parametrization	46
7.1.4	Generated test data	46
7.1.5	Negative partial snapshot	47
7.2	Model generation workflow in the distributed system domain	47
7.2.1	Description	47
7.2.2	Initial Partial Snapshot	49
7.2.3	Parametrization	49
7.2.4	Generated test data	50
7.2.5	The procedure of the leader election	51
8	Experiments and runtime performance	53
9	Conclusions and future work	54
	Bibliography	57

Chapter 1

Introduction

In order to be able to test a complex system, many artifacts are required. One of them is the possible test configurations and test data. The development of different set of test data is still a challenging task: these inputs of the testing workflow are required to be different with respect to their structure and other characteristics.

Generative environments like the Eclipse Modeling Framework (EMF) [31] supports the development of models extended by well-formedness constraints and design rules using for example Eclipse OCL [38]. These environments make the specification of the abstract test criteria of the generated test data efficient.

1.1 Problem statement

Systematic testing is a complex, time consuming and expensive task [24]. Model-based testing provides a systematic method for representing test data in the form of models. The first step is the development of the domain where the behaviours are interpreted. This covers the metamodel and the corresponding constraints, which is the language of the possible test data. This way a domain specific language is constructed, which is the input of the test data generation approach. Based on this specification, test data can be derived as instance models of the input metamodel. This makes it challenging to generate test cases automatically since state-of practice metamodels are highly complex. Furthermore, a naive algorithm to generate test cases according to a given source metamodel will produce many irrelevant or inconsistent test models; therefore, a test case generator must be highly intelligent.

Another challenging task in this procedure is to compare the generated target model for a given test case with the oracle. Both the oracle and the target are models. Comparing the oracle and target can be done either (1) syntactically, in which case, the comparison algorithm must compare two graphs. This task can be traced back to the graph isomorphism problem, which is NP-complete, or (2) semantically in which case, the comparison algorithm must have deep knowledge of the semantics of the target language.

1.2 Research context

Testing is an important requirement in the design of critical systems prescribed by many standards. Model-based test data generation is frequently used in the design of critical systems. The process of the model-based test data generation is shown in Figure 1.1. The steps of model-based test data generation [35]:

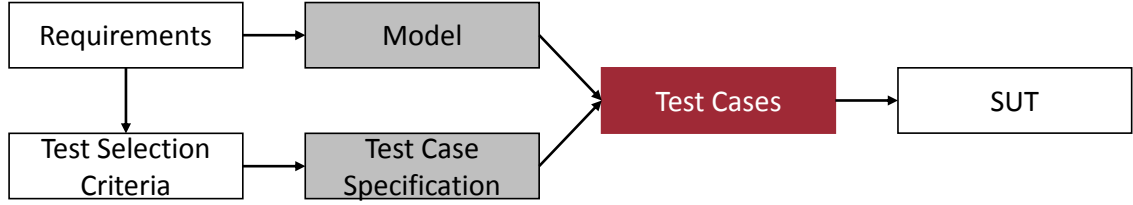


Figure 1.1: *The Process of Model-Based Test Data Generation*

- I. A model based representation of the SUT is developed according to the requirements and the specification documents.
- II. The test selection criteria are defined. Test selection criteria is used to navigate the search during the generation procedure.
- III. Test selection criteria are transformed into test case specification which formalises the notion of test selection criteria and render them operational.
- IV. With using the test case specification and the model a test suite is generated.
- V. After the test cases are generated, the next step is to prepare them to be runnable. Usually the models are more abstract than the system under test, so the generated test cases have to be mapped to the implementation level.

My goal was automatize the stages III. and IV. of the model-based test data generation process.

1.3 Objectives

The main objective of this work was to develop an automated framework which is able to generate abstract test data. The output of the algorithm is a diverse set of test data which is used as possible test configuration settings and as input data of the System Under Test (SUT). The steps of test data generation procedure are: (1) definition of the system (using metamodels, instance models and well-formedness constraints) and the test data selection criteria (2) generation of a set of various abstract test data (3) concretization of the abstract test data to structurally different models of the metamodel.

1.4 Contribution

I propose an approach for the abstract test data generation of domain specific languages which covers the handling of metamodels, well-formedness constraints and instance models. The essence of the approach is to generate structurally different abstract test data by mapping the test generation problem to a set of logical axioms. This logic problem can be solved by the Alloy Analyzer [14] which uses a SAT solver as a back-end solver. I also propose powerful approximation techniques to handle the mapping of complex language constructs [26]. It is carried out by completing a prototypical initial instance models (called partial snapshots) in accordance with the DSL specification. The benefit of this approach is that this approach supports the use of more than one metamodel and more instance models of the metamodels.

I implemented also a research prototype tool to demonstrate the practical feasibility of the approach. The tool takes DSL specifications in the form of EMF models, which is an open

source technology widely used in the industry. Model queries are specified using the standard Object Constraint Language (OCL) [23]. I integrated the Alloy Analyzer, which provides us various specification language, it uses SAT solvers in the background which are removable.

The tool has been successfully applied in case studies of industrial projects which come from the laser guided vehicle and distributed system domains.

1.5 Previous works

In this section I overview my contribution compared to my former TDK report. In Figure 1.2 the most important components of the two works are presented. In our previous work [6] we created a **Language Validation Framework** which can analyse different language properties of DSL specifications and additionally it supports concrete attribute value generation for partial models based on different well-formedness constraints. This model generation method was used to generate concrete test data for an existing abstract test data. For this purposes we used an SMT solver back-end (Z3 SMT solver [8]). The DSL specification was translated to SMT axioms and the output of the Z3 was interpreted as language properties or concrete test data.

In this report I introduce a new abstract test data generation process, where abstract test data is defined by structural requirements. I implemented a **Test generation framework** to overcome this problem since it is much more complex task than formalizing “simple” attribute constraints based on integer values (concretization). To solve this, the DSL specification is now translated to Alloy models and axioms which can be solved by the Alloy Analyzer [14]. The Alloy Analyzer uses different SAT solvers as back-end solver: this way I could fully exploit the efficiency of modern SAT solver algorithms and technology.

I also extend the abstract test data generation methodology with feed back loop which is implemented in the **Multiple Model Generator** component. This means that the submodels of the generated result can be fed back to the model generation process and based on this I can generate structurally different models in the upcoming iterations. In this work the equivalence of the generated test data is also formalized with which I can define the structural difference of the models (equivalence classes).

Additionally, in this report I present a *Test Criteria Description language* which is used to specify a logical combination of the elements of the DSL specification and that way the requirements of the abstract test data generation can be fine-tuned.

The goal of the two work was different, while the combination of them provides an integrated test data generation framework (e.g. the generated abstract test data can be concretized and finally ready-to-use test data can be produced).

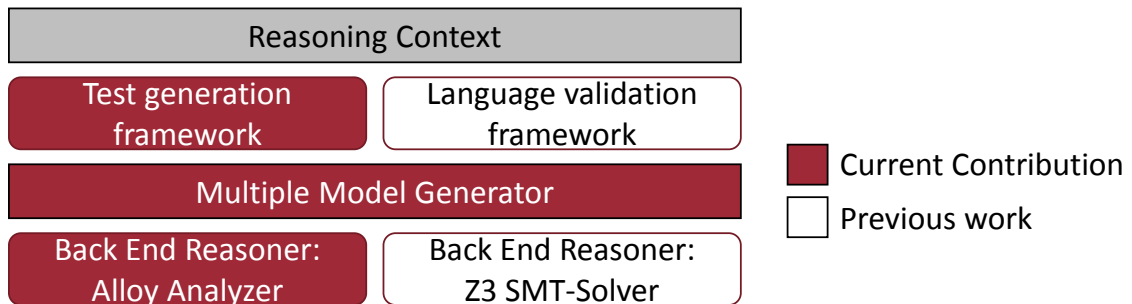


Figure 1.2: Differences of the TDK reports

1.6 Structure of the report

The rest of the report is structured as follows. In Chapter 2 I summarize the most important technical and theoretical background of my work. In Chapter 3 the motivating scenarios are presented. Then in Chapter 4 I give a brief overview of the proposed abstract test data generation approach. The features of the mapping are presented in Chapter 5, then Chapter 6 introduces the details of the implementations. The Chapter 7 presents the details of the industrial case-studies. Then in Chapter 8 I examine the scalability issues. Finally, I conclude my work in Chapter 9.

Chapter 2

Background

In this chapter the most important theoretical concepts are presented which are necessary for understanding the proposed approach. First the abstract test data generation workflow and the definitions of modeling, the properties of the metamodels and instance models are introduced. Afterwards the well-formedness constraints and components (OCL rules) are presented which can represent additional rules on the models. Finally the mathematical problem solvers, the problem classes (SAT, CSP, SMT) are shown then the related work is presented.

2.1 Test data generation workflow

The goal of my work is to create a test data generation framework for supporting the automatic generation of abstract test data. In this approach test data is constructed in the form of models, that represent the input for the testing processes (e.g., represents a context of an autonomous agent or distributed system). Abstract test data in my approach is a test data, which focuses only on the structural properties and do not take care on concrete attribute values.

The process of the abstract test data generation is shown in Figure 2.1. Test data generation is based on the artifacts that define the domain-specific modeling language (DSL) of the abstract test data models (metamodel and constraints) and the required partial snapshot patterns. All of these inputs are translated to a constraint solving problem in the **Model generator** component and based on different type of back-end solvers (e.g. SAT, SMT solver) the axioms are solved. The solver process can be parametrized additionally by the information of the **Parametrization** component. The solver produces a set of logical axioms which can be translated to a valid instance model, that can be interpreted as abstract test data.

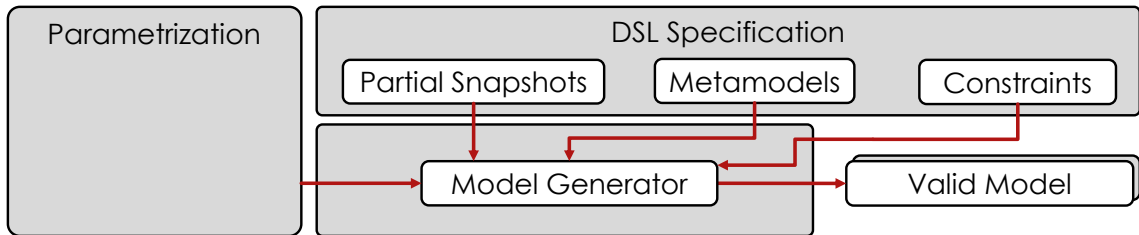


Figure 2.1: *Test data generation workflow*

2.2 Mathematical logic

In this part the formalism of the FOL, the classes of problem and their provers are presented.

2.2.1 First order logic

First order logic (FOL) is a formal language used in mathematics, philosophy and computer science. In FOL the domain of the model is a set of individuals which names are domain elements. The objects are in relations each other. So the FOL contains objects, relations and symbols which represent functions. The types of symbol are: (1) constant symbol which symbolizes the object, (2) predicate symbol which signs the relation and (3) function symbol which refers to the functions. All predicates and function symbols have arity. The semantics connect the sentences to the model which is able to determine the truth. To do this an interpretation is needed which link the real objects and the symbols.

The syntax of the FOL in Backus-Naur Form:

		(Sentence Connector Sentence)
		Quantifier Variable, ... Sentence
		$\neg$ Sentence
Atomic Sentence	$\rightarrow$	Predicate(Term) Term=Term
Term	$\rightarrow$	Function(Term, ...)
		Constant
		Variable
Connector	$\rightarrow$	$\implies$ $\wedge$ $\vee$ $\Leftrightarrow$
Quantifier	$\rightarrow$	$\forall$ $\exists$
Constant	$\rightarrow$	A X
Variable	$\rightarrow$	a x
Predicate	$\rightarrow$	T isAbstract ...
Function	$\rightarrow$	att() parent() ...

The sentences are connected to the model by the semantics, which is able to evaluate the truth value of a sentence on a model. The syntax elements are introduced in the following. The term is a logic expression which refer to an object. The terms and the predicate symbols are the components of the atomic sentences. To create complex sentences, logical connectives are used. In the FOL the sets of the objects can be formulated logical expressions thanks to the quantifiers. It has two types: the universal quantifier ($\forall$) and the existential quantifier ($\exists$). Variable is followed by quantifier, which symbolizes the objects. The $\forall x P(x)$ means that P is true for every object x. The $\exists x P(x)$ means that exists x which makes P true. In the FOL equality symbol can be used to create statement.

2.2.2 Prover and solver techniques

In this part the problem classes (SAT, SMT, CSP) and their solvers are introduced.

Satisfiability Problem (SAT)

To define the SAT language the Boole-formula should be introduced. The Boole-formula is built up from 0, 1 logical constants, 0-1 valued variables ($x_1, x_2, \dots x_n$), their negated expressions, the $\wedge$ (“and”) and the $\vee$ (“or”) operators. The variables and their negated expressions are the literals. The result of an evaluated formula is 0 or 1. The Boole-formula is satisfied if their variables have an evaluation where the value of the formula is 1. The SAT is the language of the satisfiable Boole-formulae. The SAT language is in the NP class: a good evaluation is the witness of satisfiability of the formula. The SAT has subsets which the conclusion is effective e.g. the 2-SAT is polynomial. Generally, the SAT is an NP-complete language, which is evidenced by S.A. Cook and L. Levin. The SAT solver searches substitution values which make the Boole functions true, an example SAT solver is the MiniSat [22].

Constraint Satisfaction Problem (CSP)

The formal definition of the CSP problem are formulated by the set of variables ($X_1, X_2 \dots X_n$) and constraints ($C_1, C_2, \dots C_m$). All X_k variables have a D_i domain which defines the possible values. Every C_l constraint restrict the subset of variables which define the value combination of the subset. A problem state is defined by the variable-value assignment. The assignment is complete if every variable is in the subset. This is a solution of the CSP. Usually CSPs have finite domain and its variables are discrete. The Boole CSP is the special case of NP-complete problems. For example the eight queens puzzle is a CSP. CSP solver is e.g. the Sugar [2].

Satisfiability Modulo Theories (SMT)

The SMT problem is a decision problem for logical formulae with combinations of background theories expressed in classical first order logic with equality. It is different from the SAT because predicates over suitable set of non-binary variables are used to. SMT formulae provide richer language than is possible with the SAT formulae.

SMT is used to software verification, planning, model checking and automated test data generation. The interest theories in these applications include formalizations of arithmetic, arrays, algebraic data types and functions. The SMT solvers use the standard SMT-LIB [3] language. SMT solvers are e.g. the Alt-Ergo [20], Barcelogic [30], Beaver [16], CVC4 [1], Mistral [33], SONOLAR [12], Yices [4], Z3 [8].

Alloy Analyzer

The Alloy language is a simple but expressive logic based language which is inspired by the Z specification language and Tarski’s relational calculus [14]. Alloy’s syntax is designed to make it easy to build models. The Alloy model is a collection of constraints that describes the set of structures.

The Alloy language is interpreted by the Alloy Analyzer software. This is a solver which takes the constraint of the model and finds the structures which are satisfy them. It is reduce the code to SAT problem which can be resolved by different SAT solvers e.g. Kodkod [34] which is a new model finding engine that optimizes the reduction.

The Alloy provides a logic language for describing graph-structures and constraints using high order logic and relation algebra. The language of Alloy contains various language elements,

paragraph ::=	factDecl enumDecl sigDecl
sigDecl ::=	sigQual* “sig” name + [“extends ref”] “{” decl,* “}” [block]
sigQual ::=	“abstract” “lone” “one” “some” “private”
enumDecl ::=	“enum” name “{” enumLiteralName ("," enumLiteralName)* “}”
factDecl ::=	“fact” “[name”] block
block ::=	“{“ expr* “}”
expr ::=	quant decl + blockOrBar
	unOp expr
	expr binOp expr
	expr arrowOp expr
	expr [“!” “not”]compareOp expr
	expr (“=>” “implies”) expr “else” expr
	expr “[“ expr,* “]”
	number
	“-” number
	“none”
	“Int”
	“(“ expr “)”
	block
	“{“ decl,+ blockOrBar “}”
quant ::=	“all” “no” “some” “lone” “one” “sum”
decl ::=	{ “private” } { “disj” } name,+ “:” { “disj” } expr
unOp ::=	“!” “not” “no” “some” “lone” “one” “set” “#” “~” “*” “^”
binOp ::=	“ ” “or” “&&” “and” “&” “<=>” “iff” “=>” “implies” “+”
	“_” “.”
compareOp ::=	“=” “in” “<” “>” “=<” “>=”
blockOrBar ::=	block
blockOrBar ::=	“ ” expr

Figure 2.2: *Language elements of Alloy*

but in this report only a subset of these elements are used. The most important elements are presented in Figure 2.2. The generated Alloy code has to be completed with a run command, with which the number of the generated objects can be defined.

2.3 Modeling background

The inputs of the test data generation framework are metamodels and instance models. To create an advanced modeling environment, the metamodel can be augmented with *well-formedness constraints*, which capture additional restrictions any well-formed instance model needs to respect. Such constraints can be defined as OCL invariants. In order to illustrate my abstract test data generation technique, I use an autonomous robotic and a distributed system domains which are defined with EMF metamodels, OCL constraints over the EMF metamodel.

2.3.1 Metamodeling

Metamodels define the main concepts, relations and attributes of the target domain to specify the basic structure of the models. In this report, the Eclipse Modeling Framework (EMF) [31] is used for domain modeling.

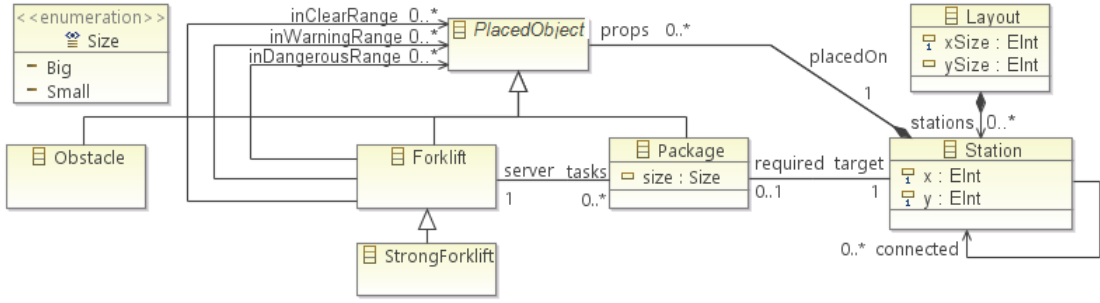


Figure 2.3: Metamodel of the R3Cop project

A simplified metamodel for an autonomous robotic domain is shown in Figure 2.3. The Layout element represents the root of a model, which contains each Station and it has a `xSize` and `ySize` attributes which are defined the size of it. Stations have a `x` and a `y` attributes which represents the coordinates of the Station, a `connected` and a `required` references. The Station contains PlacedObjects which is the supertype of the Forklift, Package and Obstacle. The Package has a `size` attribute which value should be Big or Small. The Package has a `server` and a `target`. The Forklift is the supertype of StrongForklift. The Forklift has `tasks` and with its references the distances between the Forklift and the PlacedObject are represented.

The formal definition of the EMF model elements [26]:

- **Classes (CLS):** In this formalism the concepts are represented by *EClasses* (which are simply referred as classes) that can be instantiated to *EObjects* (or objects). The meta-model can specify finite types with predefined set of $\{1, \dots, n\}$ literals by *EEnums*. For both classes and enums, if an e is instance of a T type it is denoted as $T(e)$.
- **Generalization (GEN)** can be specified between two classes to express that a more specific (child) class has every structural feature of the more general (parent) class. From a typing perspective, for all metamodel classes $abst, spec$ with $super(abst, spec)$, we have $\forall e : spec(e) \implies abst(e)$.
- **Abstract (ABS):** If a class is defined as *abstract*, it is disallowed to have direct instances, i.e. for all abstract class abs and an instance o with $isAbstract(abs) \wedge abs(o)$, there exists a non-abstract subclass cls of abs with o with $\neg isAbstract(cls) \wedge cls(o)$.
- **References (REF)** *EReferences* defined between classes capture the relations of the domain. When two o and t objects are in a relation r , an *EReference* (or an *EAttribute*) is instantiated leading from o to t denoted as $r(o, t)$.
- **Attributes (ATT)** *EAttributes* enrich the expressiveness of classes with values of predefined primitive types like integers, strings, etc. Each attribute of a class needs to have a valid value, thus $\forall o, att \exists v : att(o, v)$. Metamodels may also define the default value def of an attribute.
- **Type compliance (TC)** Type compliance requires that for any relation $r(o, t)$, its source and target objects o and t need to have compliant types.
- **Multiplicity (MUL)** The multiplicity of structural features can be limited with upper and lower bound in $lower..upper$ form, i.e. for any relation r leading from o , we have $lower \leq |\{t : r(o, t)\}| \leq upper$.

- **Containment (CON)** EMF instance models are arranged into a strict containment hierarchy, which is a directed tree along relations marked in the metamodel as containment. Any non-root model element has exactly one parent as container. Formally, $\forall t \exists \neg o : \text{nonRoot}(t) \implies \text{parent}(t, o)$.
- **Inverse (INV)** Two parallel but opposite directional *inverse* reference can be defined as inverse of each other to specify that they always occur in pairs. This means that for all pairs of references *forw* and *back* that *inv*(*forw*, *back*) we have $\forall o, t : \text{forw}(o, t) \iff \text{back}(t, o)$.

A model M is a valid instance of a metamodel $META$ (denoted with $M \models META$) if all the corresponding constraints above are satisfied, i.e.

$$M \models \text{CLS, GEN, ABS, REF, ATT, TC, MUL, CON, INV.}$$

2.3.2 Well-formedness constraints

The Object Constraint Language (OCL) [10] [9] [23] is a declarative language, that extends the structural metamodels using extra constraints. In this report I deal with the subset of invariants, which are logical expressions that need to be satisfied at anytime during the lifecycle of an object. These expressions represent constraint over the model and a constraint solving technology can be used to add extra constraints to the metamodels and based on the semantic, unspecified missing attributes could get a value, relations or objects can be added to the model. The queries and the other types of the OCL expressions are not suitable for purpose, because they cannot be used as constraints and support model or attribute value generation.

Structural well-formedness (WF) constraints (aka design rules or consistency rules) complement metamodels with additional restrictions have to be satisfied by a valid instance model, so with these we can accurately specify the test data. Such constraints can be defined by query languages like OCL invariants, in fact, my approach supports this formalism. In many practical cases, well-formedness constraints are defined by queries which capture ill-formed model structures that are disallowed to have a match in a valid model. In the presence of a set WF of well-formedness constraints, a model M is called valid if $M \models META \wedge WF$.

Example 1 A WF constraint captures that size of the `Layout` object is bigger than 0. This OCL constraint is: `context Layout: self.xSize>0 and self.ySize>0`.

2.4 Partial Snapshot

I argue in this report that traditional (instance) models are not sufficiently flexible means to serve as inputs and outputs of the test data generation process. For instance, an abstract class is not allowed to have instances in a regular domain model (e.g. the model editor may prohibit to construct such an instance). However, allowing the use of an abstract instance can highly abbreviate many test data specification retrieved by a solver. Similarly, a small model fragment, which highlights one special property of the problem may still violate other constraints. Therefore, in the work, I use a more permissive instance level formalism called *partial snapshots (PS)* as an additional input or output of DSL validation where certain language constraints are relaxed. During a typical validation run, such a partial snapshot will be extended with further information to obtain a compliant instance model (called *completed model*) [26].

Definition 1 (Partial snapshot) A partial snapshot PS is a valid instance of a meta-model $META$ (denoted as $PS \models META$) when all the following (typing-specific) constraints above are satisfied: $PS \models CLS, GEN, REF, TC$.

I briefly describe below which constraints are dropped of the definition of regular instance models.

1. **Undefined attributes:** In a normal EMF instance object each attribute has a value (or presented default value), while a partial snapshot may contain attributes with undefined values.
2. **Abstract objects:** Partial snapshots allow to instantiate abstract **EClasses**. Such instances are handled similarly to regular objects thus they can have attributes and references. When completing a PS into a valid instance model, the type of a non-concrete object has to be refined in to a concrete subtype.
3. **Unconnected partitions:** While EMF requires that an instance model is arranged in a strict containment hierarchy, PSs allow to define instance models that are unconnected and consist of multiple model fragments. Such model fragment will be completed during the run by linking the partitions to a well-formed hierarchical containment tree.
4. **Missing / extra edges:** PSs may not contain references without their inverse relation counterparts. Similarly, PSs may also violate multiplicity constraints.
5. **New objects:** A PS defines only an initial model structure which can be extended with additional objects.

Example 2 Figure 2.4 shows four PSs generalized from instance models by removing certain model elements.

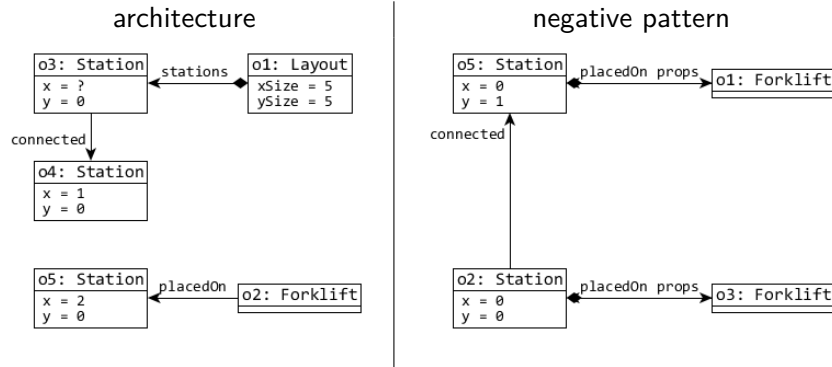


Figure 2.4: Partial snapshots with semantic modifiers

- **architecture:** This PS defines a core structure of an warehouse architecture. The instance models must contain this PS.
- **negative pattern:** This PS contains two Forklifts which are positioned side by side. This PS prohibit this structure.

Multiple PSs will be passed as an input parameter to the test data generation process (see in Section 4.1) and the solver will try to construct a valid instance model which satisfies each of them. However, there are multiple semantic modifiers for combining these PSs into a single valid completed model which are discussed below (default modifier values are underlined).

1. Positive / Negative: A **positive** PS is an incomplete model that every output model has to contain as a submodel. Formally, a model M satisfies a positive snapshot POS if there is a match m along which POS can be matched as a query: $M \models POS(m)$.

A **negative** PS has the opposite effect: if a model contains it as a submodel then it is invalid. Thus a model M satisfies a negative snapshot NEG if there is no match m along which NEG can be matched as a query: $M \not\models NEG(m)$.

2. Injective / Shareable: In case of an **injective** PS, the objects of the snapshot have to be mapped to different instance objects in the output model. Formally, $M \models INJ(m)$ when m is a query match with $\forall o_1, o_2 \in INJ : INJ(m/o_1) = INJ(m/o_2) \implies o_1 = o_2$ (where $INJ(m/o_i)$ denote the match of query element o_i in M).

In a **shareable** PS, multiple PS elements can be mapped into the same model object if it satisfies the local conditions prescribed by each mapped PS element. Note, however, that each PS is evaluated independently from each other, thus different injective PSs may share objects in the output model.

3. Modifiable / Unmodifiable: A **modifiable** PS means that the reasoning process can add extra objects and links or fill empty (undefined) attributes in the output model to satisfy the PS.

An **unmodifiable** PS means that the reasoning process can only change the context of the match of the PS (by adding/removing objects, links and setting attributes) but not in the match itself. For instance, if two objects in the PS are not linked by a certain relation, they need to remain unlinked in the result model. However, embedding the PS into a context with newly created incoming or outgoing relations (where their source or target model element is not in the match) is still allowed.

Example 3 Figure 2.4 contains several semantic modifiers for the partial snapshots:

- **architecture**: This PS can be embedded into an output model in accordance with modifiers **positive**, **injective** and **modifiable**, which is the default semantics: arbitrary extensions are allowed, but the instance model needs to contain at least three **Station**, a **Layout** and a **Forklift** objects.
 - **negative pattern**: This PS should be treated as **negative**, **injective** and **modifiable**. A successful match of this PS invalidates the output model.
-

2.5 Related work

Testing autonomous systems has become a very hot topic in recent years. As stated in a recent paper [37], testing of these systems is still an open problem. My approach combines context-based (model-based) testing and test data (instance model) generation algorithms. Here I show the most important approaches for both of these two areas and present the main differences compared to my integrated solution.

Testing approaches Most of the existing testing approaches lack of easy-to-use mechanisms to express and formalize context-aware behaviour (although these mechanisms are a prerequisite of requirements-based automated test generation and test evaluation). Most noticeably, in existing standard test description languages there is no support to express changes in the context [36]. To overcome this problem of capturing test requirements, I defined a language which is based on context models and scenario based behaviour specification.

Model generation approaches Model-driven frameworks provide different approaches to define and generate model instances.

Formula[15] is tool for DSL validation and instance model generation. Based on a metamodel, a partial instance model and a set of constraints and fixed rewriting rules, it aims to extend the partial instance model. The basic ideas of my model generation approach are similar, but I implemented multiple test data generation related features, like combination of partial instance models. The technological difference is that it uses their own modelling language, while I support the standard Eclipse technology, that is used in the industry.

There are several approaches and tools aiming to find a valid instance model for a specification enriched with OCL constraints and SAT-based technologies [28, 5, 7, 19, 29]. While these approaches provide instance model generation for an OCL enriched metamodel, my approach supports initial partial models (see in Section 2.4) and an iterative framework of optimisation purposes.

The idea of using *partial models*, which are extended to valid models also appears in [27, 15, 18]. These initial hints are provided manually to the model generation process. In my approach, these models can be combined and also assembled from a previous run in an iterative test generation algorithm. Partial models also share certain similarity with uncertain models [11], which are used by the Alloy Analyzer [25] same as my approach.

There are also mappings from programming languages extended with OCL constraints to reasoners such as Testera [17] (from Java to Alloy) and Pex [21] (from C# to Z3).

Chapter 3

Motivating scenarios and requirements

3.1 Test data generation for R3-COP

The industrial motivating scenario is the Artemis R3-COP European Union research project that aims to support automated test data generation for autonomous robots. The main goal of this project is to develop a model-based test data generation approach, that provides relevant test data for the industrial partners (e.g. Elettric80).

The Elettric80 company produces laser guided automatic forklifts (Laser Guided Vehicle, LGV), which should operate in a warehouse and fulfill safety and security requirements. The goal is to test these LGVs using a black box testing method: environments are generated (test data), during test execution the test trace is recorded and finally the trace is evaluated based on the requirements. The environment of the truck is modeled using a domain specific modeling language. Test data refers here to an environment of the autonomous agent (environment objects including dynamic objects that may appear, disappear or move).



Figure 3.1

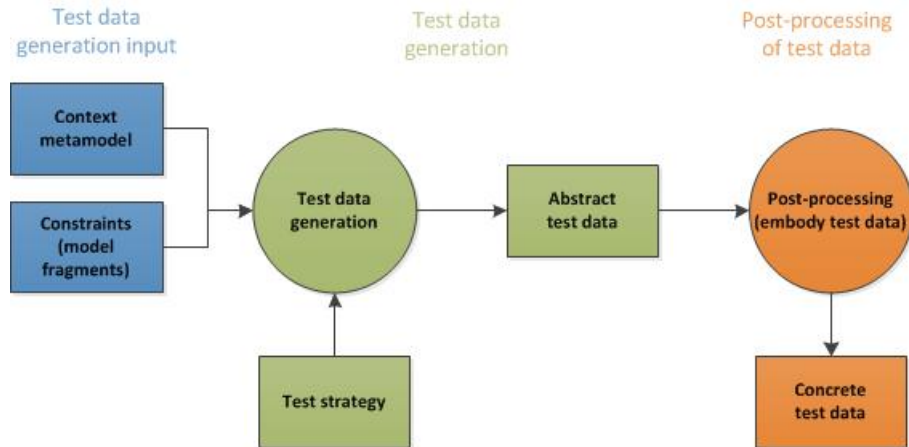


Figure 3.2: Test generation scenario

A high-level overview of the test data generation is depicted in Figure 3.2. The approach uses the environment metamodel of the system under test constructed by domain experts, and represents test data as model instances conforming to this metamodel. On the basis of the metamodel the additional well-formedness constraints are captured using OCL invariants. Based on this context description language a test data should be generated in the form of valid instance models, which conform the metamodel and satisfy the constraints.

The test data generation algorithm is based on constraint satisfaction problem, where the constraints originate from the metamodel (e.g. type hierarchy, associations, multiplicity) and the well-formedness constraints. The result of this should be a valid instance model.

The test data generation process is divided in two phases to support different test data generation strategies and deal with the problem complexity. First, only the structural constraints are satisfied and an abstract test data is generated. Here abstractions are used in the model to avoid the use of concrete attribute values (e.g., two objects can be *far* or *near* (association), but the exact position is not given with coordinates). In the second, post-processing step, these abstractions are resolved and concrete attribute values are generated. This post-processing step was presented last year in my TDK report [6].

Due to the two phased test generation, not all the input constraints are taken into account in every step (e.g. first an abstract test data is constructed, where some constraints can be violated or abstract model elements can be used). The support of the selection of the relevant constraints during each step is also important. A domain expert should choose (or formalize using some logical expression) which constraint should be included or not in the given test data generation step.

This report is motivated by the previously mentioned abstract test data generation step. Additionally I prepared the logical language that is used to define which constraints are affected in the given test data generation step.

3.1.1 Metamodel

The metamodel of the R3-COP industrial project domain is shown in Figure 3.3. The detailed descriptions of the model elements are shown in Section 2.3.1. The warehouse which the model is presented contains layouts which have size. The layout contains stations which have got coordinates. The stations should be placed to each other. The stations contain different type of warehouse fixtures (PlacedObject), which should be Obstacle, Forklift and Package. The package should be small or big. In the warehouse the big packages can be transported by strong forklift and the small one can be transported by forklift.

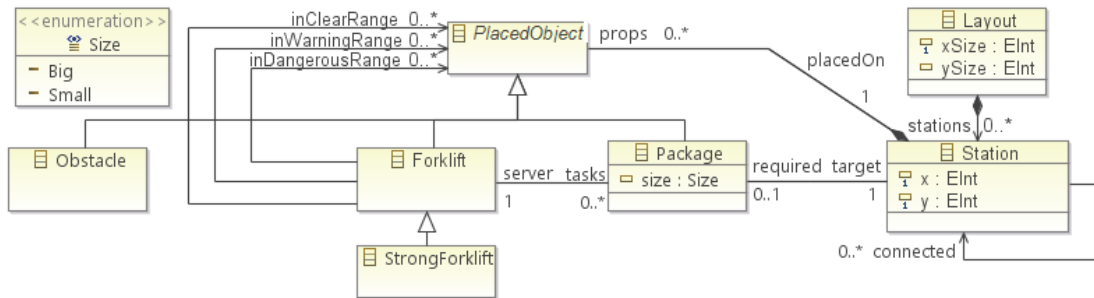


Figure 3.3: Metamodel of the R3-COP project

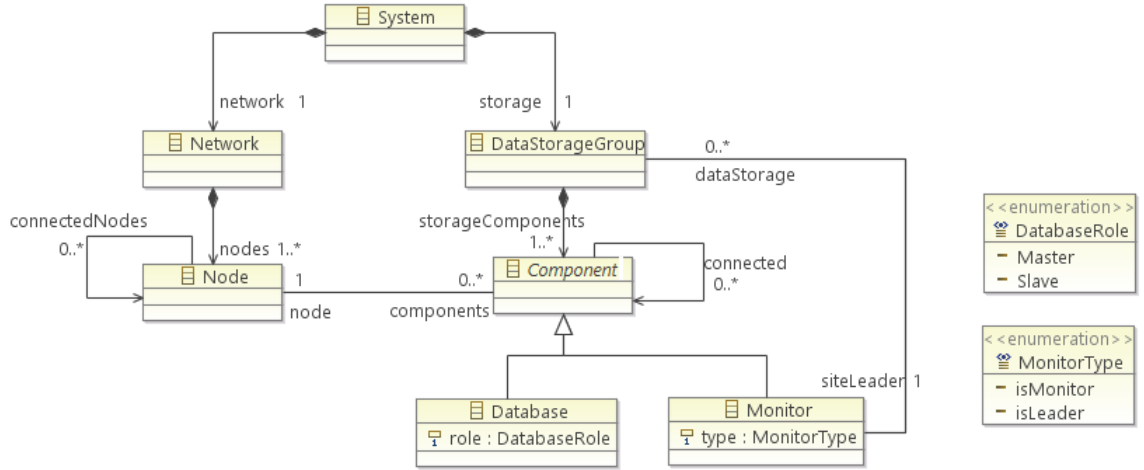


Figure 3.4: Metamodel of the distributed system domain

3.2 Distributed system domain

Distributed systems are widely used by the industry from small-scale to up to huge cloud systems. In this section I introduce an example based on a real case study, a distributed cloud system, however I reduced the size of the metamodel and also the number of constraints to fit here. The structure of distributed systems usually evolves with time, where many algorithms run parallel to maintain the consistency and reactivity of the systems. For this purpose these systems have a layered architecture, where each layer uses the services of the underlying layers and provides services for the upper layers. The analysis of such systems is a complex task where testing can provide a solution. However, to be able to produce the inputs of the testing procedure needs manual development of the test data and configurations, or requires automatic techniques being able to work on the structural description of the system. In this section I show the metamodel of a simple distributed system enriched with well-formedness constraints. The test data generation approach is able to produce different configurations from the input metamodel, which serves as test input data or possible test layout, configuration.

3.2.1 Metamodel

In cloud systems virtualization is widely used which is also a key element in our example: the physical resources are the **Nodes** in a **Network**, where the network connections are represented by the references **connectedNodes**. These elements define the physical layout of our system. The logic layer of the system is represented by the **DataStorageGroups**, which provides data storage services. Fault tolerant behaviour is provided by the applied redundancy: in each **DataStorageGroup** there are multiple **Databases**. There is a distinguished **Database** at each **DataStorageGroup**, the **Master**. If there is a problem with the **Master Database**, then a master election procedure is initialized and the next **Master** is elected. There is also a monitoring system, a **Monitor** can be **isMonitor** or **isLeader**. **Monitors** monitor the **Databases** and collects data of their behaviour which is then sent to the **Leader**. Each **DataStorageGroup** has one **Leader**, which is the **siteLeader** of the **DataStorageGroup**. Each **Component** is allocated to a physical **Node**. **Components** are connected through the **connected** relation if the **Nodes** which are allocated to are same.

The metamodel of the distributed system domain is shown in Figure 3.4. The metamodel represents the connection between the physical and logical elements. The **System** contains the **DataStorageGroup** through the **storage** reference. The **DataStorageGroup** contains the **Component**

elements which are referred by the `storageComponent` reference. The `Component` is an abstract class which symbolizes the logical components of the distributed model. The `Component` should be connected by each other through the `connected` reference. The subtype of the `Component` are the `Database` and the `Monitor`. The `Monitor` has `type` attribute which type is the `MonitorType` enum which value should be `isMonitor` or `isLeader`. The `Database` has a `role` attribute which type is the `DatabaseRole` enum which value should be `Master` or `Slave`. The `DatastorageGroup` has a `siteLeader` reference which is referred to the leader `Monitor`. The `System` contains `Network`, it is referred by the `network` reference. The `Network` contains `Node` through the `nodes` reference. The `Node` represents the physical components of the system. The `Node` elements should be connected each other with the `connectedNodes` reference. The logical components have to be assigned to the physical nodes. The components and nodes which are assigned to each other are connected by the `node` and `components` references which are the opposite of each other.

Chapter 4

Overview of the approach

4.1 Test generation process

In this section I summarize the high level view of the proposed test data generation approach and then I describe also the most important building blocks of the algorithms. Test data generation (Figure 4.1) is started by the description of the data domain of the tests, this is the input metamodel of the algorithm. Additional parameters are defined with the help of the Test Description Language: initial model fragments are provided as partial snapshots and also OCL constraints are used for the definition of well-formedness requirements. The approach is capable of generating multiple different test data, in this case the discriminator property should be provided to help the algorithm find structurally different instance models i.e. abstract test data. The number of required output test data can also be defined in the approach. The model generator transforms the problems into the input language of the Alloy tool, which is able to traverse the state space of the possible models and constructs valid models. These output models servers as abstract test data for the testing of complex systems.

The definition of Test Criteria Description Language is presented in Section 4.2.4 which defines the logical connection between the partial snapshots and constraints. The mapping of the **Meta-models**, **Partial Snapshots** and **Constraints** to Alloy formulae is formalised in Chapter 5. With the **Discriminator Property** the structural difference of the output valid models can be defined which is presented in Section 4.2.3. In this approach the number of the models and the size of the output test data can be defined which is introduced in Section 4.2.2.

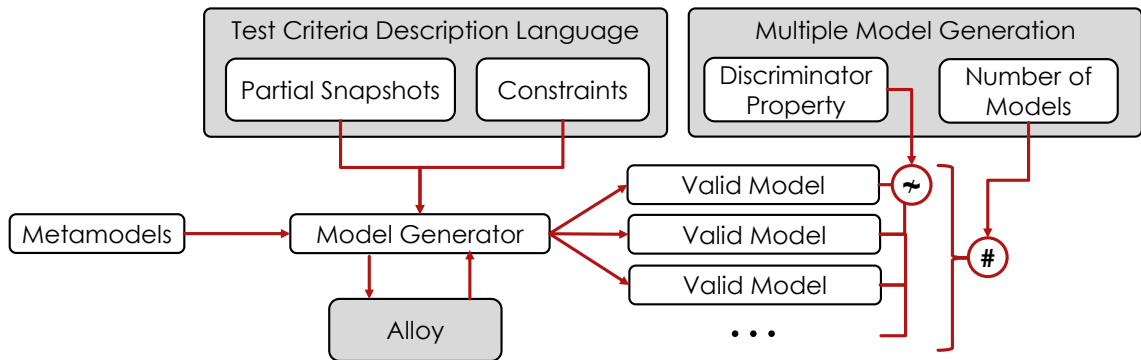


Figure 4.1: Functional overview of the approach

4.2 Input for the test data generation process

In this section the required input parameters of the abstract test data generation approach are presented.

4.2.1 DSL specification

Parameters in the **DSL Specification** refer to the actual target of the generation process. It is important to note that the input elements are fully functional standard artifacts of the tool.

Metamodel The metamodel contains the main concepts and relations of the test data and defines the graph structure of the instance models. To enrich the expression power of the language attributes are added to the concepts. By doing this, the language can be extended with predefined domains of data types (like integers, strings) that supported by the metamodeling language. Additionally, some structural constraint might be specified with the elements like multiplicity.

To more precisely specify the range of valid instance models, different constraints might be added to the test data generation input. Those constraints can be included to the **Constraints** of the reasoning phase.

Constraints The goal of the well-formedness constraints is to define rules that have to be satisfied in a valid model, with this the test data can be more precisely specified.

The search should be controlled by some practical preconditions.

Partial Snapshot (PS) By adding an initial structure to the **Instance** the reasoning process will be more directed as the tool checks only the cases that contains this initial structure as a submodel.

The tool is capable of deriving a PS from any EMF model, and a valid PS can be automatically transformed back to a normal instance model. So if the user does not need any of the previous options, standard instance models also can be used.

4.2.2 Test data generation parameters

The parameters in the **Generation** allows to customise the reasoning process. To more precisely control the generation process many more logic-dependent options can be added to the tool. Some of them might cut down the search space (like a fixed model size), others adjust the transformation tool to be more efficient for special tasks.

Model size In Alloy the size of the generated model has to be explicitly defined. With the parametrization language the model size and the number of the int elements can be defined.

Number of structurally different generated models This approach supports the generation of the structurally different test data. With the parametrization language we can specify the number of the generated models if exists enough different model.

Discriminator Definition By Defining the discriminator property the test engineer is able to specify the required difference between the test cases in order to generate diverse set of test cases.

Test Criteria Description Test Criteria Description is used to define the input combination of the artifacts in the DSL Specification that should be used in the generation process. This allows the parametrization of the test data generation process with variously defined test criteria.

4.2.3 Discriminator definition

The goal of the approach is to generate multiple test input models which are not similar to each other. In this report, similarity between the output models are defined by an equivalence relation. First, in order to define the equivalence between model structures the formal definition of the equivalence relation is introduced:

Definition 2 (Equivalence relation) $\sim$ in an equivalence relation on a set H if $\sim \subseteq H \times H$ and satisfies the following criteria:

- it is reflexive, so $\forall a \in H : a \sim a$.
- it is symmetric, so $\forall a, b \in H : [a \sim b] \implies [b \sim a]$.
- it is transitive, so $\forall a, b, c \in H : [a \sim b \wedge b \sim c] \implies [a \sim c]$.

In order to differentiate the output models and support the generation of structurally different models the model equivalence should be also defined. During the generation process models are needed that are not isomorphic. Two models are isomorphic if between their objects, references and attributes exists an f bijective function.

Definition 3 (Partial snapshot isomorphism) The partial snapshot isomorphism is an equivalence relation on partial snapshots (and thus it is also applicable to instance models). Two partial snapshots PS_1 and PS_2 are given which are defined with their objects (OBJ), references (REF) and their attributes (ATTR). The partial snapshots are isomorphic if there is a bijective function $f : PS_1^{OBJ} \rightarrow PS_2^{OBJ}$ that satisfies the following constraints:

- for each type $T \in \text{Class}$ the each object o is same typed as the image: $T(o) \Leftrightarrow T(f(o))$
- for each reference $R \in \text{Reference}$ an o_1 object refers to o_2 when the image does: $R(o_1, o_2) \Leftrightarrow R(f(o_1), f(o_2))$
- for each attributes $A \in \text{Attribute}$ an o object has the value v when the image does: $A(o, v) \Leftrightarrow A(f(o), v)$

To work with the structurally different models a representation function is introduced.

Definition 4 (Representation function) There is a H set and the $\sim \subseteq H \times H$ equivalence relation is defined on it. The $\text{rep} : H \rightarrow C$ function is the representation function, the elements of C are the representations, if the statement is satisfied:
 $\forall a, b \in H (a \sim b \Leftrightarrow \text{rep}(a) = \text{rep}(b))$

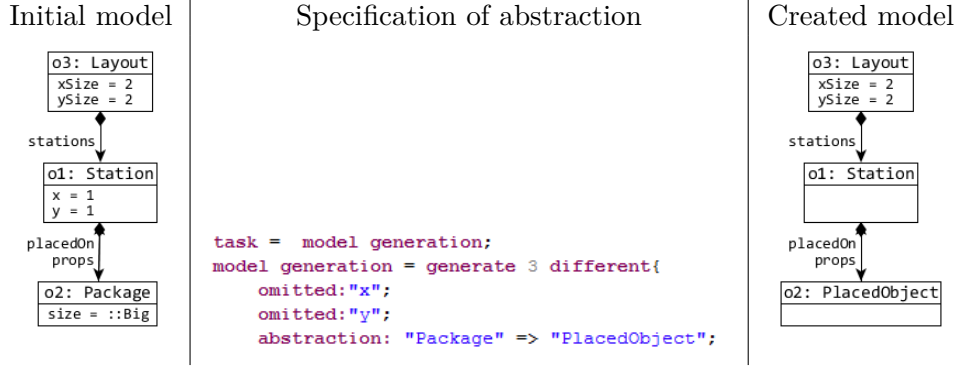


Figure 4.2: Abstraction of the feedback model

The representation function in this case is a text which specifies which elements are similar in the two model. The example of the definition of it is shown in Figure 4.2. The defined elements with the abstraction language should be similar in the structurally different models. If from the two structurally different model the elements which are presented in Figure 4.2 example are emphasized from the $PS1$ and the $PS2$ models then the result is $PS1_1$ and $PS2_1$ where the $rep(PS1_1) = rep(PS2_1)$ is satisfied. If from two models the elements which are defined with abstraction language are left then the remaining subgraphs are not isomorphic. So the goal of my work is to find structurally different models on which the $rep(PS1) = rep(PS2)$ is not satisfied.

I defined an abstraction function $\mathcal{A}$ in the following way

- *Attribute and reference removing* The attributes, references of the instance model can be left. With parametrization language the name of the references and attributes are referred.
- *Type removing* The objects which specified with their types can be left.
- *Object type abstraction* The object type can be changed to the one of the supertypes of it. With parametrization language the name of the actual type and the new type are referred. If the type is abstracted then the latter type specified attributes and references are removed too.

Example 4 In Figure 4.2 an abstraction of a model is presented. The initial model contains a Layout, a Station and a Package. The attributes of the objects are filled so it is a complete and valid instance model. With the parametrization language I specify that the x and the y attributes are omitted and the Package type is changed to PlacedObject. As the result the x and y attributes are deleted and the type of the o2 object is PlacedObject. The attribute of the o2 is deleted because it was a Package specific element.

4.2.4 Test criteria description language

An other goal of my work is to support the definition of the different test criteria and to support that the test data generation process should be parametrized arbitrarily with input parameters. To reach this goal a test criteria description language (TCDL) is defined with which the requirements of the test data generation can be formalized.

For this task predicates are introduced that are commonly understood to be a Boolean-valued functions $P : X \rightarrow \{true, false\}$. Each constraint has been annotated with a unique label which

can be used as a predicate in the TCDL. If the value of the predicate is true, then the criteria has to be satisfied, else it has to be violated.

Just to give a small example, let expect test data satisfying an OCL constraint: *OCL1*. This OCL invariant describes a desired structural behaviour of the system (detailed examples are in Chapter 7). In the introduced *TCDL* language the following holds: we can define a property by simply writing *P*, which refers to the OCL invariant *OCL1*. This means that the predicate $P(OCL1)$ is satisfied and then the generated model satisfies the OCL constraint (*OCL1*).

The test criteria description language contains predicates which are assigned to the input parameters like partial snapshots and constraints. The goal of the test data generation can be specified with the predicates of the inputs and with the defined boolean operators between them. The TCDL enables the using of the $\wedge$, $\neg$, $\vee$, $\implies$, $\Leftrightarrow$ operators. With this language the test engineer can specify the test descriptions and the logical operators between them.

The grammar of the TCDL:

TCDL	$\rightarrow$	$\langle Label \rangle \mid \neg \langle TCDL \rangle \mid$
		$\langle TCDL \rangle \wedge \langle TCDL \rangle \mid$
		$\langle TCDL \rangle \vee \langle TCDL \rangle \mid$
		$\langle TCDL \rangle \implies \langle TCDL \rangle \mid$
		$\langle TCDL \rangle \Leftrightarrow \langle TCDL \rangle \mid$
Label	$\rightarrow$	$[OCL\ constraints] \mid [PS]$

The *Label* should be OCL constraint or partial snapshot, between the predicates of them can be defined logical connections.

Example 5 A metamodel (M), two partial snapshots (PS1, PS2), and an OCL constraint (OCL1) are defined as input parameter of a test data generation process.

The $TCD = P(PS1) \wedge (\neg P(PS2)) \wedge P(OCL1)$ test criteria description means that the generated test data has to satisfy the first PS (PS1) and the OCL constraint (OCL1) and it must not satisfy the second partial snapshot (PS2).

Chapter 5

Mapping DSLs to Alloy formulae

I discuss in this section the details of mapping DSLs artifacts to Alloy formulae to be processed by SAT solvers. First I present some theoretical foundations and the detailed mapping of meta-models and partial snapshots in Section 5.2 followed by mapping of the OCL constraints into Alloy in Section 5.4.

During the description of the mapping I use the following form: the metamodel, partial snapshot and the OCL mapping are isolated, the mapping of the language elements are separated. I formalize and introduce the previously unused elements then the equivalent Alloy expression is presented. In complicated cases I demonstrate the mapping method on an example, which is derived from the R3-COP LGV case study.

5.1 Foundation of the mapping

The mapping takes a DSL context as input to create a set of axioms called DSL_F as output (where F denotes that this is a set of logic formulae), which is satisfiable if and only if the original DSL context was consistent. If the DSL_F is satisfiable then by definition there is an interpretation M_F that satisfies DSL_F . Additionally, I back-annotate the logic structures M_F derived as a result of the test data generation process to an actual instance model (or partial snapshot) of the DSL, formally:

if $forward(DSL) = DSL_F$ and $backannotation(M) = M$ then

$$\begin{array}{ll} 1. & M_F \models DSL \Leftrightarrow M \models DSL \\ 2. & DSL_F \text{ unsatisfiable} \Leftrightarrow DSL \text{ inconsistent} \end{array} \quad (5.1)$$

Function *forward* consists of different transformation steps each of which takes certain DSL artifacts and yields a set of corresponding logic axioms:

- The metamodel transformation creates the formulae $META_F$ from the metamodel (detailed in Section 5.2).
- The instance model transformation derives the formulae PS_F from the partial snapshots (discussed in Section 5.3).
- Finally, OCL transformation creates formulae OCL_F for OCL constraints (Section 5.4).

So the mapping of DSL into Alloy is partitioned in the following way:

$$DSL_F = META_F \wedge Criteriy(PS_F \cup OCL_F) \quad (5.2)$$

where the *Criteriy* function compose the constraints derived from OCL constraints and partial snapshots.

5.2 Mapping metamodels

In this section the mapping of the metamodel is presented.

5.2.1 Type mapping

The objects of the domain specific language are mapped to the elements of a newly crated `Object` Alloy signature, so I included the abstract `Object` class to the hierarchy with which is the explicit superclass of the classes. One benefit of the introduction of the `Object` is that in Alloy we have to specify the number of the generated elements. In this case we only have to specify the number of `Object` elements and the reasoner automatically identifies the concrete type of the model elements. Additionally, some generic, type-independent property can be defined on the `Object`, like the containment hierarchy. The objects are arranged in a containment hierarchy which is modelled by `contains` relation between the objects which represents the union of the containment edges in the model. The `contains` edge make easier the definition of the containment hierarchy which is presented in the latter section. The model is automatically completed with this object which only appears in the Alloy code. The input metamodel is not changed and the generated test data does not contain unknown typed elements.

The `Object` signature is defined in the following way:

```
abstract sig Object { contains: set Object }
```

In Alloy the equivalent formula of the class is the `sig` language element. For example, the `Layout` class is mapped to `sig Layout { }`. If the class is abstract or interface the signature is extended with the `abstract` keyword.

Enum types are also mapped to their Alloy equivalent. The following example shows the result of the mapping of the `Size` enum which has the `Big` and the `Small` literals:

```
enum Size { Big, Small }
```

5.2.2 Type hierarchy

EMF metamodels define generalisation between classes. The Alloy language supports single inheritance which is expressed with the `extends` keyword. Mapping generic inheritance hierarchy og the EMF to Alloy can be implemented in the multiple ways:

1. First, the most trivial solution is to simply not support metamodels with multiple inheritance. The inheritance between two class is implemented as follows:

```
sig A extends B
```

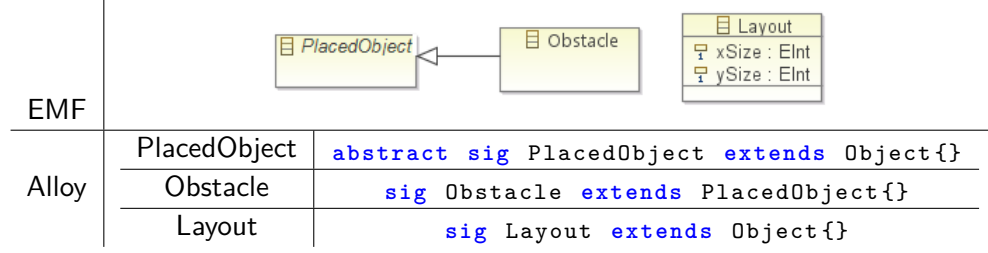


Figure 5.1: Mapping type hierarchy

2. Instead of the using of built-in types I should define every objects with `Object` type, and type-predicates should be use to define the concrete type of the object. The disadvantage of this approach is that the possible combinations of the predicate have to be defined by a complex constraint. The

```
sig Object{ type: set Type }
enum Type{A,B,C}
fact { /* each object has correct type */ }
```

3. Each attribute and reference is copied to the subclasses, the inheritance is solved directly. The difficulty with this approach that the elements of a type cannot be selected simply. A possible solution for referring a type is unioning each subclass like the following example:
(A + B + C)

This report uses the first option because it is handled by the solver the most effectively. Additionally, further contributions can be presented more easily with simple type mapping. An example of the transformation of class hierarchy is shown in Figure 5.1. This example shows an abstract `PlacedObject` class, the `Obstacle` class which is subtype of the `PlacedObject` and the `Layout` class are mapped. In Alloy the appropriate `sig` element and the `extends` and `abstract` modifiers are used.

5.2.3 References

The references of the metamodels define the directed edges between the instance objects. For EMF models, we allow directed loops but disallow parallel edges of a certain type between the same objects. In such a case, edges can be treated as relations (in the mathematical sense).

The definition of the `server` reference is translated to the following relation in Figure 5.2 (**REF**). Since an edge can lead between properly typed objects, such type information needs to be attached to the two relation ends (see `Package(o)` and `Task(t)`).

The references are be treated as relations in Alloy which are automatically type correct. The source signature contains the reference which type is equivalent to the type of the reference.

5.2.4 Multiplicity

By default, references with `0..*` multiplicity are modeled with relations in FOL. However, with explicit multiplicity restrictions, further constraints are required. With an `n..m` multiplicity, the lower bound `n` means that every object is in relation with `n` different to one, which is checked using an existential quantifier (if `n > 0`).

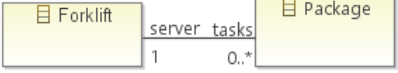
EMF		
FOL	REF: $server : Package \times Forklift \mapsto \{true, false\}$ TC: $\forall o, t \in Object : (tasks(o, t) \Rightarrow (Package(o) \wedge Forklift(t)))$ MULT: $\forall src \in Object : Package(src) \Rightarrow \{trg \in Object server(src, trg)\} = 1$ INV: $\forall o, t \in Object : (server(o, t) \Leftrightarrow tasks(t, o))$	
Alloy	REF: <code>sig Package { server: one Forklift }</code> INV: <code>fact{ server = ~tasks }</code>	

Figure 5.2: Mapping references

EMF multiplicity	Alloy multiplicity
0..*	<code>set</code>
1..*	<code>some</code>
0..1	<code>lone</code>
1..1	<code>one</code>

Figure 5.3: Mapping multiplicities

References with 0..*, 1..*, 0..1 and 1..1 multiplicities are supported in Alloy. However, with explicit multiplicity restrictions, further constraints are required. The transformation of the supported references are visible in Figure 5.3.

In Alloy the cardinality operator # should be used to define lower and upper bounds. The definition of the lower bound (n) (if $n > 1$):

```
all o: Forklift | # o.tasks >= number
```

The definition of the upper bound (m):

```
all o: Forklift | # o.tasks <= number
```

5.2.5 Inverse edges

Inverse edges in metamodels express in FOL that if there is a relation rel from the object o to the target t then there has to be an inverse relation inv from t to o , formally, $\forall o, t : rel(o, t) \Leftrightarrow inv(t, o)$.

In Alloy the inverse edges should be expressed by “~” language element which means the inverse of the marked element.

The inverse relationship between `tasks` and `server` references is illustrated in Figure 5.2 at line **INV**.

5.2.6 Containment

The objects of an EMF model are arranged in a directed tree hierarchy along the containment edges. This relationship is formalized by multiple formulae. First the FOL formalization of the problem is presented.

1. The containment relation is defined as the union of the containment-edge relations:

FOL	$contains : Object \times Object \mapsto \{true, false\}$ $\forall p, c \in Object : contains(p, c) \Leftrightarrow (subElement(p, c) \vee \dots)$
Alloy	<pre>sig Object { contains: set Object } fact { contains = containsTask + containsPackage + ... }</pre>

2. Exactly one root object does not have parent, every other object has exactly one parent:

FOL	$\exists! root \in Object \forall parent \in Object : \neg contains(parent, root)$ $\forall o \in Object : (o = root) \vee \exists p \in Object : (p \neq o) \wedge (contains(p, o))$
Alloy	<pre>one root : Object (root.^contains = none) and (all other : Object other != root => # other.contains = 1)</pre>

3. The tree hierarchy also requires acyclicity which means that any object is unreachable from itself along a path of containment edges.

FOL	$\forall o \in Object : \neg contains^+(o, o)$
Alloy	<pre>no o: Object o in o.^contains</pre>

Alloy does not support the containment but the directed tree hierarchy can be expressed.

5.2.7 Attributes

The attributes of a metamodel are the properties of the classes with built-in type. My mapping support the single valued attributes with predefined types such as integer attributes (which have an appropriate type in Alloy), and an enum type constructed of user-defined literals. The FOL formalisation of the attribute is same as the references but the second parameter (i.e. the range) of the relation defines the value of the attribute. In Alloy the EMF attribute is transformed to relation same as the reference, but the type of the reference is a built-in primitive type.

For example, three attributes are defined in Figure 5.4, where the mapping of the Size enum with Big and Small literals, the mapping of the size attribute which type is Size and the mapping of the id which type is Int are presented.

EMF	
FOL	$Size = \{Big, Small\}$ $size : Object \times Size \mapsto \{true, false\}$ $id : Object \times \mathbb{Z} \mapsto \{true, false\}$
Alloy	<pre>enum Size{ Big, Small } size: one Size id: one Int</pre>

Figure 5.4: Mapping attributes

5.3 Mapping Partial Snapshots

The basic approach of mapping instance models into Alloy is to create a statement to express that the output model needs to contain the partial snapshot. Therefore PS objects are transformed into existentially quantified correct typed variables, and the structure of the PS is defined over

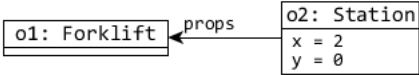
			
PS			
FOL	Containment:	$\exists o_1, o_2 \in object : [...]$	
	Distinct:	$o_1 \neq o_2$	
	Types:	$Forklift(o_1) \wedge \neg StrongForklift(o_1) \wedge Station(o_2)$	
	References:	$props(o_2, o_1)$	
	Attributes:	$x(o_2, 2) \wedge y(o_2, 0)$	
Alloy	Containment:	<code>fact {some o1, o_2, ... : Object /* structure */ }</code>	
	Distinction:	<code>some disj o1, o2, ...</code>	
	References:	<code>o1 in o2.props</code>	
	Attributes:	<code>2 in o2.x and 0 in o2.y</code>	

Figure 5.5: Example: mapping a partial snapshot

those variables. When every feature of the PS is transformed, the generated statement is added to the set of axioms derived for a DSL context to express the occurrence or the absence of the PS structure. The partial snapshots are transformed independently to Alloy, each of them has to be satisfied separately, and traceability information needs to be produced for each of them.

The mapping of a PS with two objects, attributes and references between them in accordance with the **Positive, Injective, Unmodifiable** semantics is illustrated in Figure 5.5, and discussed in details in the sequel.

The type of the object must also be specified in the statement of the partial snapshot. Note that abstract classes can be instantiated in a PS, and thus they are transformed to a structural constraint. For example, if the type of o_n is **PlacedObject** but it has a concrete type in the generated test data.

The objects are transformed to typed variables, which are the attributes of **some** expression. The references between the objects can be defined by expressing that the pair of the source and the target object is in a given relation. The mapping of reference **props** is depicted in Figure 5.5 (see label **References**), which states that o_1 is contained by o_2 . Attributes in PSs are defined similarly, see the corresponding **Attributes** line in Figure 5.5. Additionally the equivalent of the objects should be banned if I want to extort **Unmodifiable** property of the partial snapshot.

Semantic modifiers

In the previous section, I defined the mapping of partial snapshots according to the **Positive, Injective, Unmodifiable** semantics, which defines the most complete (restrictive) specification of a PS. Other semantic modifiers can be handled by simply relaxing (removing) certain constraints from this complete set derived for the **Positive, Injective, Unmodifiable** case.

This section shows how different configurations of partial snapshots with respect to a **Positive, Injective, Unmodifiable** one.

- **Shareable:** If the PS is configured to be shareable, the **Distinct** constraint have to omitted from the formula, so multiple variables can be bound to the same object variable of the PS.

PS	o1: Station x = 1 y = 0 o3: Station x = 2 y = 0	
FOL	Containment:	$\exists o_1, o_2 \in object : [...]$
	Distinct:	$o_1 \neq o_2$
	Types:	$Station(o_1) \wedge Station(o_1)$
	References:	$\neg connect(o_2, o_1) \wedge \neg connect(o_1, o_2)$
	Attributes:	$x(o_1, 1) \wedge y(o_1, 0) \quad x(o_2, 2) \wedge y(o_2, 0)$
Alloy	Containment:	<code>fact {some o1, o2, ... : Object /* structure */ }</code>
	Distinction:	<code>some disj o1, o2, ...</code>
	References:	<code>not o2 in o1.connected and not o1 in o2.connected</code>
	Attributes:	<code>2 in o2.x and (no other: Int other != 2 and other in o2.x</code>

Figure 5.6: Example: mapping an unmodifiable partial snapshot

- **Modifiable:** If the PS is unmodifiable then the absence of a reference or attribute is captured as a negated predicate. By omitting the negative predicates from **References** and **Attributes**, it is allowed to add new references to the model, an example is shown in Figure 5.6. If we use only instantiated types in the PS then the model is unmodifiable. With the usage of abstract types the type changing is automatically guaranteed. As a result, even abstract objects can be used in a PS, and they will be matched to a concrete one.
- **Negative** Depending on that the PS is configured as positive or negative, the expression of the generated statement or its negation is inserted to the axioms.

5.4 Mapping OCL invariants

OCL constraints (invariants) are widely used means to express well-formedness rules of meta-models and DSLs which has to be satisfied by all valid instance models. Here I present a mapping of (from a subset of) OCL invariants to Alloy and I also formalize the OCL expressions in FOL. The combined use of the rich partial snapshot language and OCL invariants also allows to specify precisely the requirement of the generated test data.

5.4.1 An overview of OCL mapping

The syntax of an OCL invariant expression is presented in the template below, where `context` specifies the environment (e.g. class) on which the constraint is interpreted, the name of the constraint can be given after the `inv` keyword and finally the expression is specified.

`context<classifier>inv[<constraint_name>]: <expression>`

My mapping takes an OCL invariant as input and synthesizes Alloy `fact` formulae as output. Certain OCL language elements are too expressive to be expressed using in FOL, but they can be approximated. Each supported language element is presented in this subsection.

The OCL semantics defines a four valued logic (with `true`, `false`, `null` and `undefined` values), while I mostly restrict my mapping to a two-valued logic, while the `null` is also supported as input of the comparison operators.

The structure of an OCL invariant is similar to its Alloy counterpart, so the mapping algorithm transforms the elements explicitly, one-by-one to Alloy formulae.

The mapping algorithm traverses the *abstract syntax tree* (AST) of the OCL invariant recursively and applies the corresponding mapping rules to each subexpression element. However, there are some special cases which require pre- or post-processing the result, e.g. for comparison functions, null references and collection operators. The mapping also handles a restricted set of higher-order structures (like sets).

5.4.2 Basic expressions

The mapping rules of the basic expressions (primitive-types, simple arithmetic and bool expressions and variables) of OCL are depicted in Table 5.1. In OCL, I cover the primitive types Integer while the String, Boolean and Real types are not supported, because the Alloy is not supported them, but it can execute operations on expressions which returns with boolean value. The logical and mathematical operators are directly mapped to Alloy, since they have their equivalent counterpart in the Alloy if they are interpreted on Integers.

OCL	Alloy
<code>variable</code>	<code>variable</code>
<code>a+b</code>	<code>a+b</code>
<code>a-b</code>	<code>a-b</code>
<code>a*b</code>	<code>a*b</code>
<code>a/b</code>	<code>a/b</code>
<code>a=2</code>	<code>a=2</code>
<code>a and b</code>	<code>a and b</code>
<code>a or b</code>	<code>a or b</code>
<code>a implies b</code>	<code>a implies b</code>
<code>not a</code>	<code>not a</code>

Table 5.1: Mapping of basic OCL expressions

Objects, as complex structures require special transformation rules. *Single-valued attributes* of objects are translated to relations. The example shows the mapping of the `size` reference.

OCL	<code>package.size</code>
FOL	$size(package, Big)$
Alloy	<code>package.size</code>

Navigation along *references with at most one multiplicity* is compiled into a relation call, the example represents the mapping of the `target` reference.

OCL	<code>package.target</code>
FOL	$\exists s : station(package, s)$
Alloy	<code>package.target</code>

The *equal* operator of OCL (=) is mapped similarly to other mathematical operators unless objects need to be compared. The mapping of such equality expressions are divided into two cases: (1) if a variable is placed on the right hand side, then the equal operator is used to avoid undefined values, (2) if the comparison is to value null, then the expression is mapped using the equal operator and the equivalent expression of the null is none.

OCL	<code>package.target=s</code>
FOL	$\exists s : target(package, s)$
Alloy	<code>package.target=s</code>
OCL	<code>package.target=null</code>
FOL	$\neg \exists s : target(package, s)$
Alloy	<code>package.target=none</code>

The mapping of the *not equal* operator of OCL (<>) uses the dual counterparts of equal operation by adding a negation to the corresponding Alloy expressions or use the not equal (!=) operator.

5.4.3 Collections

OCL collections are special sets in the mathematical aspect obtained mostly by specific language constructs (e.g. `allInstances`) or navigation along references.

Every collection is represented by a predicate $P(x)$ captured in FOL, but in Alloy there are built-in language elements and methods with which collection can be treated so I used that in my approach.

The OCL operation `allInstances()` refers to all instances of a certain type, that way, the corresponding FOL formula collects the objects with the referred type. In Alloy I can refer to same type elements with a correct typed variable. I also can refer to the set of elements with a certain type as defined by the context of the invariant using `self` keyword, which is handled exactly as the `allInstances` construct. The example show how can refer to `Station` typed elements.

OCL	<code>context Station inv: self...</code>
OCL	<code>Station.allInstances()</code>
FOL	$P(x) \equiv Station(x)$
Alloy	<code>x:Station</code>

A set of instances can be also referred by a reference with more than one multiplicity. The predicate of the reference and the predicate of the variable is included in the FOL expression during the mapping. In the example the expression is referred to the objects which is connected to the `station` named object trough the `connected` reference.

OCL	<code>station.connected</code>
FOL	$P(x) \equiv connected(station, x) \wedge Station(station)$
Alloy	<code>station.connected</code>

Navigation along references (see next example) with more than one multiplicity is also allowed in OCL (first line), but it is a shorthand of the `collect()` operator and is mapped to this operator

directly by the OCL parser (second line). The equivalent FOL expression contains the predicates of the references included in the path and a temporary variables with universal quantifier for the intermediate points. The predicates of the intermediate- and endpoints are not needed, since the predicates of the references include them. In Alloy the references can be concatenated with the join operator (`.`).

OCL	<code>layout.stations.connected</code>
OCL	<code>layout.stations->collect(v v.connected)</code>
FOL	$P(x) \equiv \forall v : stations(layout, v) \wedge connected(v, x) \wedge Layout(layout)$
Alloy	<code>layout.stations.connected</code>

The `select()` and `reject()` operators are used to define a special subset of the collection by an expression. The `select()` is used to select, which elements should be included, while the `reject()` defines the elements that should be excluded from the collection. The transformation of these operators adds the mapping of the expression to end of the expression. The equivalent of the references and the attribute is connected by `and` operator.

OCL	<code>l.stations->select(s s.x>10)</code>
FOL	$P(s) \equiv Layout(l) \wedge stations(l, s) \wedge x(s) > 10$
Alloy	<code>s in l.stations and s.x>10</code>
OCL	<code>l.stations->reject(s s.x>10)</code>
FOL	$P(s) \equiv Layout(l) \wedge stations(l, s) \wedge \neg(x(s) > 10)$
Alloy	<code>s in l.stations and not(s.x>10)</code>

5.4.4 Collection operators

I now overview the mapping of different OCL operators which are applied on collections.

The OCL operation `includes()` is evaluated to *true* if the collection contains at least one element satisfying the argument expression of the function. I search an element which is equivalent to the expression of the `includes`. The `includes()` is transformed to an expression which is tested that the marked object is contained by the collection on which the `includes()` is interpreted. The `excludes()` OCL function is the dual of `includes()`, so it is transformed to the negated expression of `includes()`. This expression is satisfied if the elements of the collection do not satisfy the formulated condition.

OCL	<code>station.connected->includes(s)</code>
FOL	$connected(station, s)$
Alloy	<code>s in station.connected</code>
OCL	<code>station.connected->excludes(s)</code>
FOL	$\neg connected(station, s)$
Alloy	<code>not(s in station.connected)</code>

OCL operation `forAll()` is an iterator over a collection to state that certain conditions hold for each member of the collection. The Alloy equivalent of it is the `all` language element and it can be

represented by the universal quantifier in FOL. The $\text{expr}_{\text{Alloy}}$ and expr_{FOL} refers to the mapping of the expr_{OCL} in the next examples, which is defined by other mapping rules.

The OCL operation `exists()` implements an iterator and the Alloy equivalent is the `some` and in FOL it is equivalent to the existential quantifier.

OCL	<code>Station.allInstances()->forall(station expr_OCL)</code>
FOL	$\forall station : Station(station) \implies \text{expr}_{FOL}$
Alloy	<code>all station:Station expr_Alloy</code>
OCL	<code>Station.allInstances()->exists(station expr_OCL)</code>
FOL	$\exists station : Station(station) \wedge \text{expr}_{FOL}$
Alloy	<code>some station:Station expr_Alloy</code>

The OCL function `notEmpty()` is applied on collections and it is satisfied if the collection is not empty. This operation is transformed to an existentially quantified predicate which means that there is at least one object in the given set or collection. The OCL function `isEmpty()` is the dual of `notEmpty()` handled with an additional negation or with the `no` Alloy language element.

OCL	<code>station.connected->notEmpty()</code>
FOL	$\exists s : \text{connected}(station, s)$
Alloy	<code>some s:Station s in station.connected</code>
OCL	<code>station.connected->isEmpty()</code>
FOL	$\neg \exists s : \text{connected}(station, s)$
Alloy	<code>no s:Station s in station.connected</code>

The OCL function `size()` returns the size of the collection. The mapping of the less than (`<`), greater than (`>`) an equal (`=`) operators provides the basis for the mapping of the equality and inequality operators. In Alloy with the cardinality operator (`#`) can query the size of the collection.

OCL	<code>context Station inv:self.connected->size()>=2</code>
FOL	$\forall self \in Object : Station(self) \implies \{s \in Object \text{connected}(self, s)\} \geq 2$
Alloy	<code>all self:Station #(self.connected)>=2</code>

The handling of equality and inequality operators has specific rules if two collections are passed as input. Two collections are equal if and only if neither of them contains an element which is not in the other set. In Alloy it can be expressed by the equal (`=`) operator.

OCL	<code>s1.connected=s2.connected</code>
FOL	$\neg \exists s : (\text{connected}(s1, s) \wedge \neg \text{connected}(s2, s)) \vee (\neg \text{connected}(s1, s) \wedge \text{connected}(s2, s))$
Alloy	<code>s1.connected=s2.connected</code>

Passing two collections as input, the inequality operator evaluates to *true* if there exists at least one element which is not common in both of them.

OCL	<code>s1.connected<>s2.connected</code>
FOL	$\exists s : (\text{connected}(s1, s) \wedge \neg \text{connected}(s2, s)) \vee (\neg \text{connected}(s1, s) \wedge \text{connected}(s2, s))$
Alloy	<code>s1.connected!=s2.connected</code>

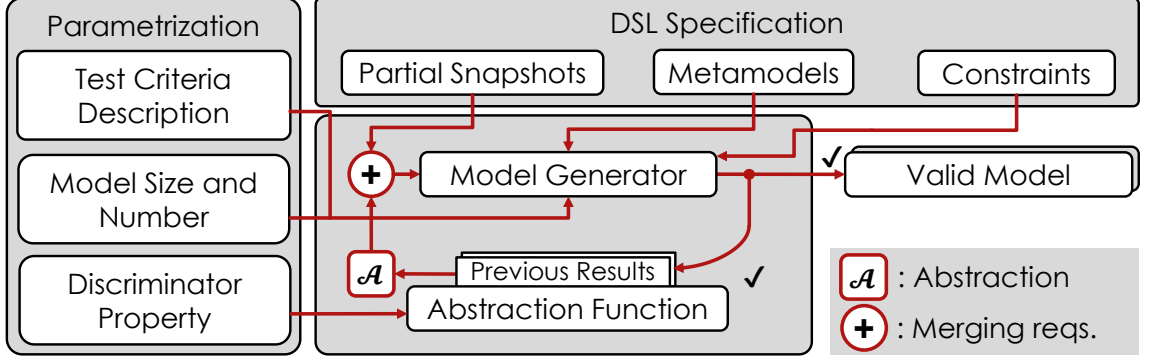


Figure 5.7: Functional overview of the approach

5.5 Abstract test data generation approach

The current section introduces a model generation algorithm that aims to generate valid instance models that can be used as abstract test data. The approach supports the output of language development tools like EMF as input. These frameworks are able to produce metamodels to define the types and the basic structure of the generated test model elements. To constraint the range of the valid instance models the metamodel can be supplemented with additional constraints: our approach supports well-formedness rules developed in OCL. The output of the test data generation process is a diverse set of valid instance models of this language that satisfy all required constraints. The generation process can be further constrained using (1) partial snapshot that defines an initial submodel that have to be contained (at least once) in the resulting instance models and (2) parameters that define global boundaries for the generation process. These task specific parameters can be defined by the (1) Test Criteria Description Language: it defines the logical connection between the components of the DSL specifications and (2) Model Size and Number input parameters to define the required model size and the number of the structurally different models which have to be generated.

Based on these inputs the Model Generator maps the metamodel, partial snapshots and all its constraints to the Alloy language and applies a reasoning tool (SAT solver) to generate a valid instance model of the metamodel that fulfills all constraints and parameters. This instance model represents an abstract test data. If the input is unfeasible then the model generator confirms the unsatisfiability.

An important feature of the presented algorithm is that it is able to generate structurally different test data. The implementation uses the following algorithm to generate the *New* model from the inputs and the formerly generated “Previous” set of *Prev* models:

$$New \models MM \wedge Criteriy(PS \cup OCL) \wedge \left[\bigwedge_{\substack{Prev \in \\ Previous}} \neg \mathcal{A}(Prev) \right]$$

The following equation holds for the abstraction function:

$$M_1 \not\sim M_2 \Leftrightarrow M_1 \models \neg \mathcal{A}(M_2)$$

The novel part of my approach is that each result is persisted along the iterations by the Abstraction Function component: it feeds back relevant information about the previous models

to ensure that the model generator will generate diverse models. To achieve this behaviour, the proposed generation workflow (depicted in Figure 5.7) extends the classic generation process with two conceptually new steps: (1) the abstraction (depicted by “ $\mathcal{A}$ ” symbol) step that transforms the instance models to represent an equivalence class to increase its expressive power, and (2) a loop to feed back and reuse the previous results in the model generation in order to generate structurally different instance models.

Additionally, to support the efficient definition of partial snapshots the user can generalise them from example instance models by a simple manual abstraction technique. This way representation of equivalence classes can be defined also manually.

Chapter 6

Implementation

In this chapter the most important implementation questions, decisions and steps are presented.

6.1 Architecture

The architecture of the tool is presented in Figure 6.1 The architecture and the processing of the statements are divided four sections. First the input parametrization is introduced that:

- I. The metamodels are given as EMF Ecore model by DSL specification.
- II. The constraints add extra rules to the problem, they can be OCL invariants. It helps to specify the problem more precisely.
- III. The partial snapshots which are derived from the EMF instance models.
- IV. It contains the special requirements (e.g. the size of the generated model) of the output EMF instance models.

The partial snapshot is created by the reflective editor of the EMF instance model, which is shown in Figure 6.2.

The tool is executed using the defined test data generation parametrization. The Transformation handler gets the input which calls the transformation components: first the metamodel transformer, then the instance model transformer and finally the constraint transformer. If the transformation is successful then an Alloy model and special set of axioms is generated which contains the collected outputs of the previous components.

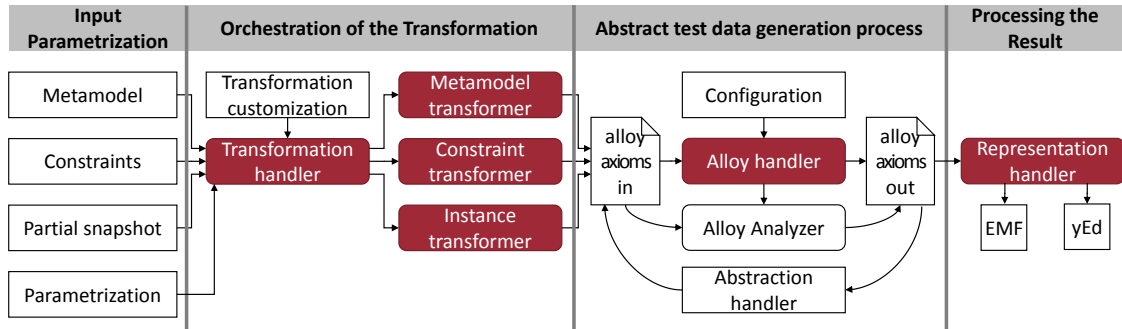


Figure 6.1: Architecture of the tool

The generated axiom system is passed to the Alloy handler which parametrize the Alloy Analyzer and run the background SAT solver (in this special case the Kodkod). The output of it is an Alloy Solution which contains axioms then it is parsed by the reasoner handler and it builds the extended partial snapshot by processing of the Alloy Solution.

The results are structurally different abstract test data. The results can be represented by different visualization tools. The representation handler represents the result, if the instance model is valid it can visualize the result as EMF model in EMF editor or yED [39] visualizer.

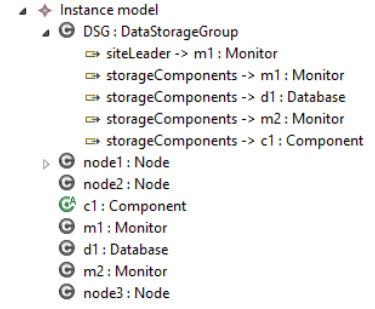


Figure 6.2: *The created editor of the partial snapshot*

6.1.1 Details

In this section the details of the implementation, the most important components are presented.

Parametrization language

The test data generator parametrization language is a simple language to support the definition of the inputs of the algorithm. To do this I constructed a grammar with the tool Xtext [32]. The language is defined by a generative grammar, which means a list of applicable rules that produce all elements of the language. During the transformation the defined symbols are used, text is not generated in the test data generation process. I can refer to the language element easily.

OCL transformation

The OCL rules and their contexts are given as a string. They must be read, parsed and finally their AST is built [13]. To achieve the goal the `OCLHelper` class is used which provides an API for parsing constraints and query expressions. The context can be identified by name which defines the place of the constraint in the model. Its `createQuery(String s)` function (1) parses the string, (2) connects the elements of the constraints to the metamodel, (3) defines the type of the OCL elements, (4) creates the AST and (5) checks the syntax of the expression. The built AST has to be visited recursively by using a visitor class and it guess the type of the language element and it call the appropriate function of the OCL transformation component.

Alloy API

The metamodel, partial snapshot and OCL transformator components have to produce Alloy axioms. The production of this code is made by using Alloy API. This API provides Java methods with which the appropriate language elements can be produced.

Visualization

The visualization of the results is very important because the result can be validated by the user. Different type of representations are realized because they can have different goals. yEd give a

nice solution which the user can edit in the yEd editor that contains a lot of built-in arrangement algorithm. Figures of the model can be saved and used for presentation, documentation. The EMF advantages is that the Eclipse modelling edition contains it. The advantage of the tool is that the visualisation components are isolated, the output is only transformed to the language of the installed visualization plugins.

Chapter 7

Case-studies

In this chapter the approach is illustrated on the R3-COP project and the distributed system domain. The different cases and the proposed solutions are presented in two case studies. Each scenario contains the requirements, the initial model fragments representing desired structural properties and the given solutions. The test data generation algorithm extends the initial model fragments with additional references, attributes, relations and model elements to produce the proper test data.

7.1 Test data generation for laser guided vehicles

In this section a scenario from the laser guided vehicle domain is presented.

7.1.1 Description

The project deals with the structure of a warehouse. The main concept of this scenario is to define the coordinates of the **Station** objects, so the **Forklifts**, **Obstacles** and the **Packages** have to be placed on the layout. The Alloy Analyzer is weak in the treatment of the numerical data but this scenario shows that the tool can still be efficient if the model contains numerical constraints. The metamodel of the project is presented in Section 3.1.1.

The constraints of the scenario are defined by OCL constraints:

- I. The coordinates of the **Station** objects are different.

```
Station.allInstances()->forall(s1, s2 | (s1.x=s2.x and s1.y=s2.y) implies  
s1=s2)
```

- II. The **Station** objects are in the area of the **Layout**.

```
Layout:self.stations->forall(s | s.x>0 and s.x<self.xSize and s.y>0 and  
s.y<self.ySize)
```

- III. The **Station** objects are neighbours of each other then they are connected with the connected reference.

```
Station.allInstances()->forall(s, s1 | not(s1=s) and ((s.x=s1.x) or  
(s.y=s1.y) or (s.x-s1.x=s.y-s1.y) or (s.x-s1.x=s1.y-s.y)) implies  
s1.connected->includes(s))
```

IV. The connected reference is symmetric.

```
Station.allInstances()->forall(s1, s2| s1.connected->includes(s2) implies
s2.connected->includes(s1))
```

V. A PlacedObject and a Forklift are connected by inDangerousRange if their distance is smaller than 2 units.

```
Forklift.allInstances()->forall(f| PlacedObject.allInstances()->forall(p|
((f.placedOn.x-p.placedOn.x) * (f.placedOn.x-p.placedOn.x) +
(f.placedOn.y-p.placedOn.y) * (f.placedOn.y-p.placedOn.y) < 4) implies
f.inDangerousRange->includes(p)))
```

VI. A PlacedObject and a Forklift are connected by inWarningRange if their distance is bigger than 2 units and it is smaller than 5 units.

```
Forklift.allInstances()->forall(f| PlacedObject.allInstances()->forall(p|
((f.placedOn.x-p.placedOn.x) * (f.placedOn.x-p.placedOn.x) +
(f.placedOn.y-p.placedOn.y) * (f.placedOn.y-p.placedOn.y) <= 25 and
(f.placedOn.x-p.placedOn.x) * (f.placedOn.x-p.placedOn.x) +
(f.placedOn.y-p.placedOn.y) * (f.placedOn.y-p.placedOn.y) >= 4) implies
f.inWarningRange-> includes(p)))
```

VII. A PlacedObject and a Forklift are connected by inClearRange if their distance is bigger than 5 units.

```
Forklift.allInstances()->forall(f| PlacedObject.allInstances()->forall(p|
((f.placedOn.x-p.placedOn.x) * (f.placedOn.x-p.placedOn.x) +
(f.placedOn.y-p.placedOn.y) * (f.placedOn.y-p.placedOn.y) >25) implies
f.inDangerousRange->includes(p)))
```

VIII. The StrongForklift transports the Packages with Big size.

```
StrongForklift.allInstances()->forall(f| f.tasks->forall(p| p.size=
Size::Big))
```

7.1.2 Initial Partial Snapshot

The initial partial snapshot of the case study explains the expected fragment structure of the test data which is shown in Figure 7.1. The initial snapshot contains two separated components. One of them is a Layout object which size is 5x5 units. The other component contains 3 Station objects, the stations are connected by the Forklift, the Obstacle and the Package objects. The forklift is referred to the obstacle with an inWarningRange and it is connected to the package with the inDangerousRange reference. The package is referred to the obstacle with an inWarningRange and it is connected to the station with the placedOn reference.

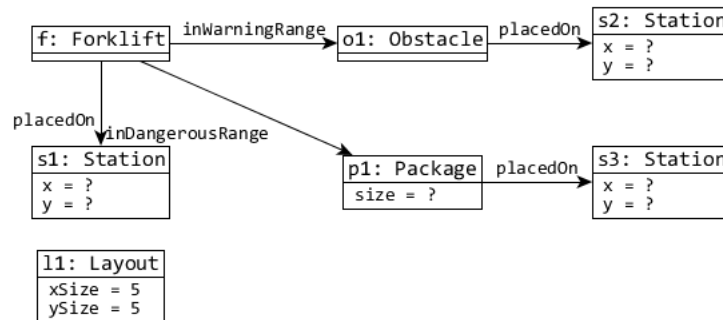


Figure 7.1: Initial partial snapshot

7.1.3 Parametrization

The parametrization of the abstract test data generation process:

- **Metamodel:** the metamodel of the Laser Guided Vehicle Domain.
- **Constraints:** the extra constraints are formalised by well-formedness constraints by using OCL. The OCL constraint are specified in Section 7.1.1.
- **Partial Snapshot:** An initial partial snapshot is used which is presented in Section 7.1.2.
- **Model size:** The size of the model should be 7. This can be an arbitrary value chosen by the user.
- **Structurally different models:** Now, only one example is generated.
- **Abstraction:** As now only one output is shown an abstraction function is not specified.

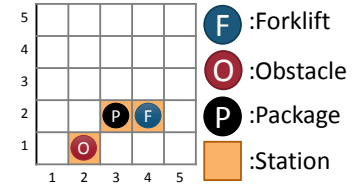


Figure 7.2: *The layout of the warehouse*

7.1.4 Generated test data

After the parameterization of the test data generator the process is executed. The generated instance model is depicted in Figure 7.3. The missing attributes, coordinates are filled according to constraints. The model is extended by the algorithm with additional edges, for example connected references. The layout of the warehouse is shown in Figure 7.2.

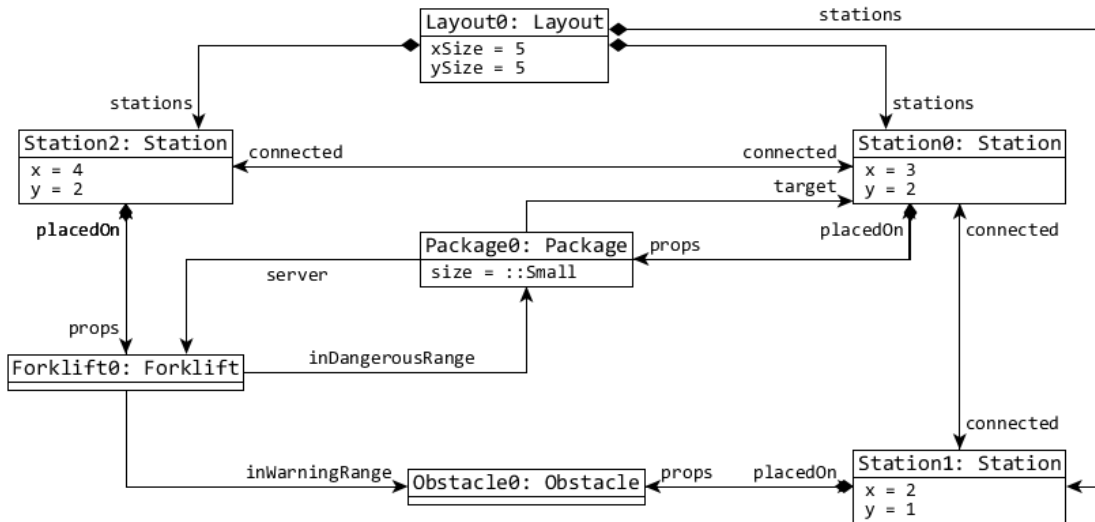


Figure 7.3: *Generated abstract test data*

7.1.5 Negative partial snapshot

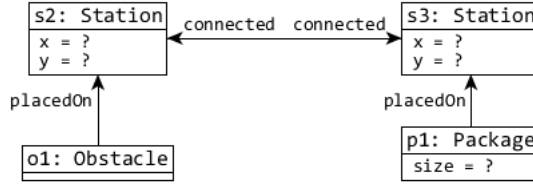
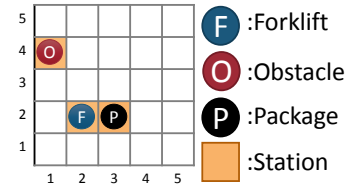


Figure 7.4: *Negative partial snapshot*

In this section I will show the effect of adding a negative partial snapshot to the input and I would like to generate an other instance model in which the obstacle and the package are not in the neighbourhood of each other. So I define a negative partial snapshot which prohibits the adjacency of them. The PS is depicted in Figure 7.4.



With the earlier defined constraints and parametrization the test data generation process is executed, the result is shown in Figure 7.6. The layout of the warehouse is shown in Figure 7.5.

Figure 7.5: *The layout of the warehouse - with negative PS*

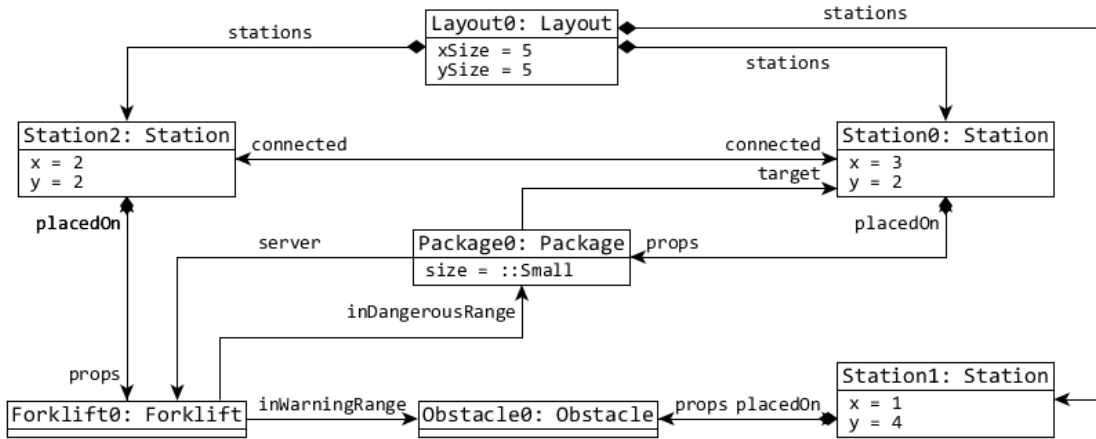


Figure 7.6: *Generated instance model*

7.2 Model generation workflow in the distributed system domain

In this section a scenario from a distributed system domain is presented, where our goal is to generate different configurations. These configurations can be used as input test data for a cloud controller.

7.2.1 Description

The project defines the connections between the logical and physical components of the distributed system model. The main concepts of this scenario is to assign the logical components

to the physical nodes and specify the most important connections between the elements and define the properties of the logical components. The metamodel of the industrial project is presented in Section 3.2.1.

The goal of the test data generation is to find configurations, where:

- the system is in the correct configuration,
- the procedure leader election should be initialized.

The constraints of the scenario are formalized by OCL rules:

I. The `connectedNodes` reference is symmetric.

```
Node.allInstances()->forAll(n| n.connectedNodes->forAll(n2|
n2.connectedNodes->includes(n)))
```

II. The endpoints of `connectedNodes` are different.

```
Node.allInstances()->forAll(n| not(n.connectedNodes->includes(n)))
```

III. The type of the `Monitor` is `isLeader` if it is the site leader of the `DataStorageGroup`.

```
DataStorageGroup.allInstances()->forAll(d| d.siteLeader.type=
MonitorType::isLeader)
```

IV. The type of the `Monitor` is `isMonitor` if it is not the site leader of the `DataStorageGroup`.

```
Monitor.allInstances()->forAll(m| m.dataStorage->isEmpty() implies
m.type=MonitorType::isMonitor)
```

V. If nodes of the `Component` elements are equivalent then the components are connected by the `connected` reference.

```
Component.allInstances()->forAll(c1, c2| (not(c1=c2) and c1.node=c2.node)
implies c1.connected->includes(c2))
```

VI. The endpoints of `connected` are different.

```
Component.allInstances()->forAll(c1| not(c1.connected->includes(c1)))
```

VII. The `connected` reference is symmetric.

```
Component.allInstances()->forAll(c| c.connected->forAll(c2|
c2.connected->includes(c)))
```

VIII. If the role of the `Database` is `Master` then it must not be connected by the other `Database` whose role is `Master`.

```
Databse.allInstances()->forAll(d1, d2| not(d1.role=DatabaseRole::Master) or
not(d2.role=DatabaseRole::Master) or d1=d2 or d1.connected>closure()->
includes(d2))
```

IX. In every case a `Database` exists whose role is `Master`.

```
Database.allInstances()->exists(d| d.role=DatabaseRole::Master)
```

7.2.2 Initial Partial Snapshot

The initial partial snapshot of the case study is shown in Figure 7.7. The initial snapshot contains three separated components. One of them contains two nodes which are connected by the `connectedNodes` references. Other one contains a separated node. The model contains a `DataStorageGroup` which has 4 components, there are 2 monitors, a database and a component. The `m1` Monitor is the site leader of the `DataStorageGroup`. The role of the database and the type of the monitors are not specified. This can be regarded as the minimum size configuration of a working system, so it is the initial model fragment i.e. partial snapshot of my algorithm.

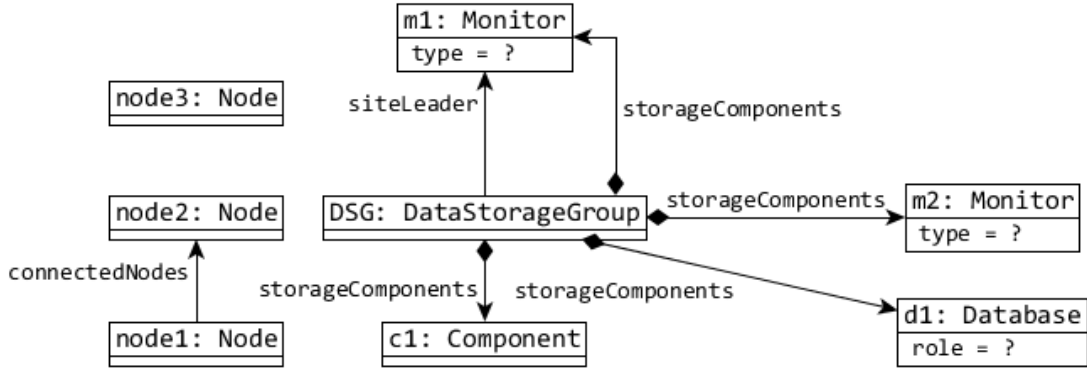


Figure 7.7: Initial partial snapshot

7.2.3 Parametrization

The parametrization of the abstract test data generation process:

- Metamodel: the metamodel of the distributed system domain.
- Constraints: the extra constraints are formalised as well-formedness constraints by using OCL. The OCL constraints are specified in Section 7.2.1.
- Partial Snapshot: An initial partial snapshot is used which is presented in Section 7.2.2.
- Model size: The size of the generated model is 10.
- Structurally different models: I would like to generate 2 different models i.e. test data.
- Abstraction: In this case, structural difference means that the generated `Monitor` object and their connection to the `Node` objects are different. So, the algorithm keeps the `Node` and `Monitor` objects with their references and attributes from the first iteration, and constructs a partial snapshot from them. This PS will then be fed back to the generation algorithm as a negative partial snapshot. Note that other objects, attributes and references other than `Node` and `Monitor` are deleted.

7.2.4 Generated test data

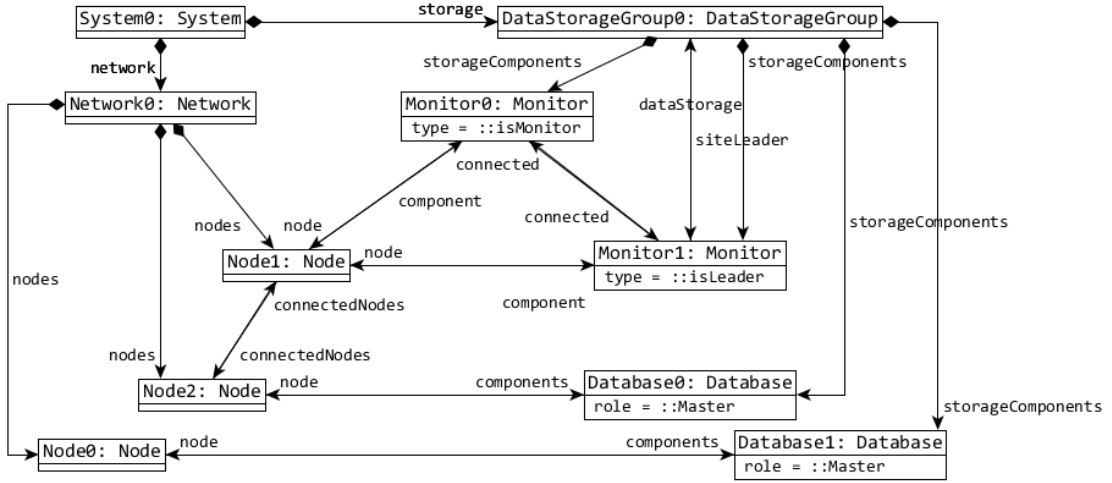


Figure 7.8: Test data generation example

After the parametrization of the test data generator process the generation is started. The first generated model is shown in Figure 7.8. The example shows that the Alloy Analyzer concludes the necessity of the Network component. It adds the missing edges, e.g nodes, connected, node. The missing attributes are filled. The generated instance model satisfies the constraints.

```
task = model generation;
model generation = generate 1 different{
    omitted: "DataStorageGroup";
    omitted: "System";
    omitted: "Network";
    omitted: "Database";
}
```

Figure 7.9: Abstraction description

Before the generation of the second model the abstraction steps are executed on the previous output model. The partial snapshot resulted from the abstraction is shown in Figure 7.10. During the abstraction (see in Section 4.2.3) objects and attributes are removed which is specified with the parametrization language. The concrete abstraction code is shown in Figure 7.9 which produced the mentioned partial snapshot. The model was fed back appropriately, so it is added to the axioms of the input of Alloy.

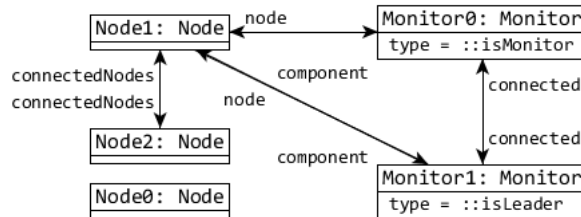


Figure 7.10: Model fragment describing the required structural differences

Then the second test data generation process is executed. The result is shown in Figure 7.11. In the example we can observe that the `connectedNode` edges which connects the `Node1` and `Node2` objects disappeared and the monitors allocated to different nodes (different endpoints of their references).

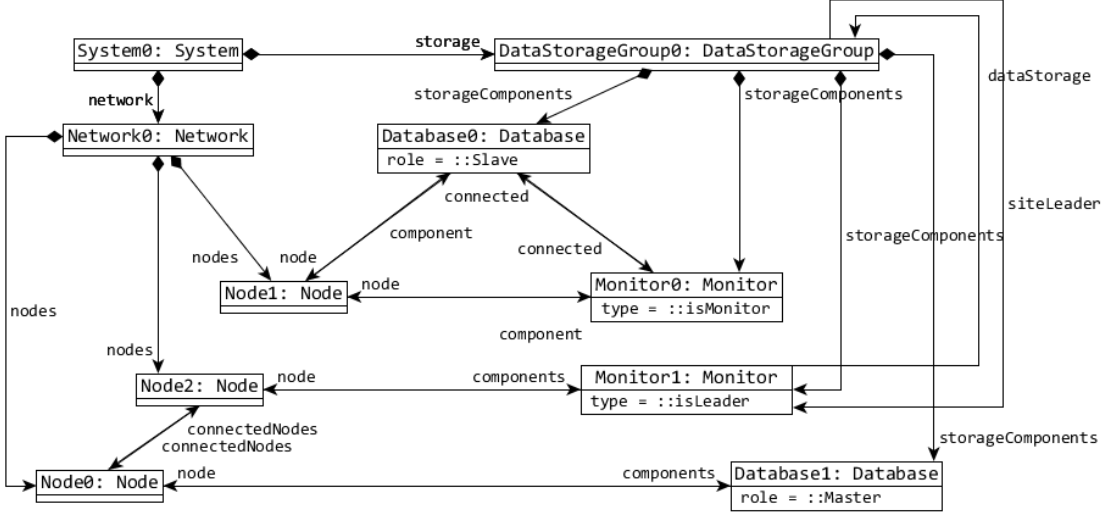
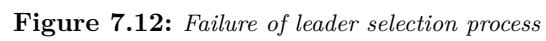


Figure 7.11: Test data generation example - with feedback

7.2.5 The procedure of the leader election

In order to show the usefulness of the TCDL (see in Section 4.2.4) and the potential of the test data generation approach, now I show an example aiming the generation of test data representing the state of the distributed system, where a leader election should be initialized. So after the first test data generation process is finished I show how to use the OCL constraints in order to generate new test data. The used OCL constraint describes the consistent state of the leader selection process (III. constraint in Section 7.2.1). I negate this constraint and I set the number of the structurally different model in the parametrization to 1. With TCDL the requirements can be easily defined, the formal definition of the test case is: $P(Meta) \wedge P(PS) \wedge P(OCL_1) \wedge P(OCL_2) \dots \wedge P(OCL_9) \wedge \neg P(OCL_3)$, where $P(OCL_I)$ is returns true if the generated test data satisfies the constraint which are expressed by the TCDL expression. The defined TCDL expression means that the generated test data have to satisfy the constraints of metamodel, initial partial snapshot, the OCL rules except the III. constraint which must not be satisfied.

Then the test data generation process is executed, the result model is shown in Figure 7.12. The example test data shows that the leader selection is failed which caused by the negated OCL constraint. It is easy to see that the generated configuration is inconsistent: the view of the system is different from the `DataStorageGroup` and the participants point of view. `DataStorageGroup` has the `siteLeader` reference to `Monitor0`, however `Monitor0` is not set to be a `SiteLeader`. This test data can be used to analyse the correctness of the leader election procedure by checking if this situation will initialize a leader election process.



Chapter 8

Experiments and runtime performance

At the end of the work I evaluated the runtime performance of the presented implementation. The runtime tasks of the framework can be separated into the following groups:

- transformation to the Alloy language,
- execution of Alloy Analyzer,
- resolution of the output Alloy Solution,
- visualization.

My analysis shows that the runtime of the transformation is proportional with the size of meta-model, partial snapshots and the number of constraints, the resolving of the Alloy Solution is proportional with the size of the solution and the visualization code generator is proportional with the size of the generated output partial snapshot. I conclude that the runtime of these components is predictable, only the time of the execution of Alloy Analyzer step cannot be approximated.

Using an average personal computer (Core i3 processor, SSD, 8GB RAM) I executed runtime performance measurements of the execution of Alloy Analyzer step, I used the DSL specification and parametrization without feed back of the distributed system domain (see in Section 7.2.1). Before the test I had to set the parameters of Alloy. I provided 4 GB of memory (this is the maximum which the Alloy is able to handle) and 8 GB of stack.

An example result is shown in Figure 8.1. The Alloy Analyzer could generate the models which contains 30 objects less than 1 minute. The generation process was a little bit longer than 1 hour if I would like generate a model which contains 80 objects. The upper bound of the number of generated objects was 80, because later the Alloy Analyzer was run out from the memory.

	10 objects	20 objects	30 objects	50 objects	70 objects	80 objects
1.	1.07	5.68	27.67	307.06	1622.92	3655.95
2.	1.85	5.64	32.46	315.55	1923.22	3587.21
3.	1.27	4.74	34.37	285.87	1821.32	3708.56
Avg.:	1.4	5.4	31.5	302.8	1789.2	3650.6

Figure 8.1: *Performance test*

Chapter 9

Conclusions and future work

The testing of distributed and autonomous systems is a challenging task. According to their complex structure which evolves with time the automatic derivation of the possible test data is a computationally complex task, where traditional approaches can not be used. The focus of my paper is the challenging task of abstract test data generation for such systems. I developed a framework to support the automatic derivation of a diverse set of abstract test data from domain specific specifications. The theoretical results of my paper are the following. I formalized the problem of abstract test data generation in order to be able to use logic solvers. The mapping is defined to work on arbitrary metamodels, partial snapshots and a subset of OCL invariants used for the specification of the test generation problem. They are mapped to the first-order-logic to provide mathematical soundness, and then I defined their mapping into the input language of Alloy. I gave a specification language called “Test Criteria Description Language” to support the test engineer in the precise definition of the required test data. I also show an algorithm which is able to construct structurally different test data from the input discriminator property. This input parameter is also used for multiple test data generation: from the discriminator property the algorithm is able to derive equivalence classes of the test data and the approach can produce different test data regarding these equivalence classes.

I implemented the algorithms in a prototype tool in Eclipse on top of the Eclipse Modeling Framework. I prove the efficiency and usefulness of my approach on a research case study and also an industrial case study from the distributed cloud system domain.

In the future I plan to combine my algorithm with concretization approaches to further refine the generated test data. This way I would expect to get a scalable test generation framework being able to handle arbitrary arithmetical and other constraints in the test data generation. As the basis of the test generation is a model generation framework, I also plan to use the algorithms to support the development of DSLs by providing interesting instance models automatically.

Bibliography

- [1] *CVC4*, May 2013. <http://cvc4.cs.nyu.edu/web/>.
- [2] *Sugar*, October 2013. <http://bach.istc.kobe-u.ac.jp/sugar/>.
- [3] *The Satisfiability Modulo Theories Library*, July 2013. <http://www.smtlib.org/>.
- [4] *The Yices SMT Solver*, January 2013. <http://yices.csl.sri.com/index.shtml>.
- [5] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Softw. Syst. Model.*, 9(1):69–86, 2010.
- [6] Ágnes Barta and Oszkár Semeráth. Consistency analysis of domain-specific languages. *TDK report*, 2013.
- [7] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. Verification of ATL transformations using transformation models and model finders. In *14th International Conference on Formal Engineering Methods, ICFEM'12*, pages 198–213. LNCS 7635, Springer, 2012.
- [8] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'08/ETAPS'08*, pages 337–340. Springer-Verlag, 2008.
- [9] The Eclipse Project. *MDT OCL*. <http://www.eclipse.org/modeling/mdt/?project=ocl>.
- [10] Eclipsepedia. *MDT/OCLinEcore*, 2013. <http://wiki.eclipse.org/MDT/OCLinEcore1>.
- [11] Michalis Famelis, Rick Salay, and Marsha Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pages 573–583, Piscataway, NJ, USA, 2012. IEEE Press.
- [12] Florian Lapschies. *SONOLAR*. <http://www.informatik.uni-bremen.de/~florian/sonolar/>.
- [13] Miguel Garcia. How to process ocl abstract syntax trees. 2007.
- [14] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
- [15] Ethan K. Jackson, Tihamer Levendovszky, and Daniel Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In *Proc. of the 14th Int. Conf. on Model Driven Engineering Languages and Systems*, volume 6981 of *LNCS*, pages 653–667, 2011.

- [16] Susmit Jha, Rhishikesh Limaye, and Sanjit Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In *Computer Aided Verification*, pages 668–674. 2009.
- [17] Sarfraz Khurshid and Darko Marinov. Testera: Specification-based testing of java programs using sat. *Automated Software Engineering*, 11(4):403–434, 2004.
- [18] Mirco Kuhlmann and Martin Gogolla. Strengthening SAT-based validation of UML/OCL models by representing collections as relations. In *European Conf. on Modelling Foundations and Applications*, volume 7349 of *LNCS*, pages 32–48, 2012.
- [19] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive validation of OCL models by integrating SAT solving into use. In *TOOLS’11 - Objects, Models, Components and Patterns*, volume 6705 of *LNCS*, pages 290–306, 2011.
- [20] Laboratoire de Recherche en Informatique, Inria Saclay Ile-de-France and CNRS. *Alt-Ergo SMT Solver*, October 2013. <http://alt-ergo.ocamlpro.com/>.
- [21] Microsoft Research. Pex. <http://research.microsoft.com/projects/pex/>.
- [22] Niklas Eén, Niklas Sörensson. *MiniSAT*. <http://minisat.se/>.
- [23] The Object Management Group. *Object Constraint Language, v2.0*, May 2006. <http://www.omg.org/spec/OCL/2.0/>.
- [24] Lukman Ab Rahim and Jon Whittle. A survey of approaches for verifying model transformations. *Software & Systems Modeling*, pages 1–26, 2013.
- [25] Rick Salay, Michalis Famelis, and Marsha Chechik. Language independent refinement using partial modeling. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212 of *Lecture Notes in Computer Science*, pages 224–239. Springer Berlin Heidelberg, 2012.
- [26] Oszkár Semeráth, Ágnes Barta, Ákos Horváth, Zoltán Szatmári, and Dániel Varró. Formal validation of domain-specific languages. *Software and Systems Modeling*, 2014. The paper was submitted and is under review process. The draft is available for reviewers in the following URL <http://mit.bme.hu/~szatmari/Sosym-DSL-2014-submitted.pdf>.
- [27] Sagar Sen, Jean-Marie Mottu, Massimo Tisi, and Jordi Cabot. Using models of partial knowledge to test model transformations. In *5th Int. Conf. on Theory and Practice of Model Transformations*, volume 7307 of *LNCS*, pages 24–39, 2012.
- [28] Seyyed M. A. Shah, Kyriakos Anastasakis, and Behzad Bordbar. From UML to Alloy and back again. In *MoDeVVa ’09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 1–10. ACM, 2009.
- [29] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL models using boolean satisfiability. In *Design, Automation and Test in Europe, (DATE’10)*, pages 1341–1344. IEEE, 2010.
- [30] Technical University of Catalonia. *Barcelogic for SMT*, November 2005. <http://www.lsi.upc.edu/~oliveras/bclt-main.html>.
- [31] The Eclipse Project. *Eclipse Modeling Framework*. <http://www.eclipse.org/emf>.
- [32] The Eclipse Project. *Xtext*. <http://www.eclipse.org/Xtext/>.

- [33] Thomas Dillig, Isil Dillig, Ken McMillan, Alex Aiken. *Mistral SMT Solver*, December 2012. <http://www.cs.wm.edu/~tdillig/mistral/index.html>.
- [34] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.
- [35] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. 2006.
- [36] Hélène Waeselynck, Zoltán Micskei, Minh Duc Nguyen, and Nicolas Riviere. Mobile systems from a validation perspective: a case study. In *6th International Symposium on Parallel and Distributed Computing (ISPDC 2007)*, Hagenberg, Austria, July 5-8, 2007, pages 85–92, 2007.
- [37] Lora G Weiss. Autonomous robots in the fog of war. *Spectrum, IEEE*, 48(8):30–57, 2011.
- [38] E. D. Willink. An extensible OCL virtual machine and code generator. In *Proc. of the 12th Workshop on OCL and Textual Modelling*, pages 13–18. ACM, 2012.
- [39] yEd Graph Editor. *yED*. http://www.yworks.com/en/products_yed_about.html.