Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Consistency Analysis of Domain-Specific Languages

TDK REPORT

Authors:
Ágnes Barta and Oszkár Semeráth

Advisors:

Zoltán Szatmári	Dr. Ákos Horváth	Dr. Dániel Varró
Research Associate	Research Fellow	Associate Professor

October 25, 2013

Kivonat

Modellvezérelt tervezés során az alkalmazási terület fogalmainak és összefüggéseinek leírására széles körben használnak szakterület-specifikus nyelveket (Domain-Specific Language, DSL). A DSL-ek segítségével automatikusan származtathatunk egy ellenőrzött rendszermodellből teszteseteket, vagy bizonyíthatóan helyes forráskódot. Azonban maguk a DSL nyelvek is tartalmazhatnak tervezési hibákat, melyek érvényteleníthetik a rendszermodellen végzett vizsgálatok eredményeit. A dolgozat fő célja, hogy olyan eszközt biztosítsunk, amellyel formális analízist végezhetünk szakterület-specifikus nyelveken, fényt derítve a DSL specifikációk ellentmondásaira és többértelműségére.

A szakterület-specifikus nyelvek konzisztencia-vizsgálata komoly kutatási kihívást jelent, mert (i) az összetett DSL-eken történő logikai következtetés algoritmikusan eldönthetetlen probléma, (ii) további elméleti nehézségei vannak a hozzáadott jólformáltsági kényszerek és a származtatott értékek kezelésének, és (iii) olyan eszköz fejlesztésére van szükség, amit a következtetési eljárás ismerete nélkül is használhat a nyelv tervezője.

A TDK dolgozatunkban egy egységes keretrendszert javasolunk a szakterület-specifikus nyelvek konzisztenciavizsgálatára a következő módon: (i) A jólformáltsági kényszereket és származtatott érték definícióját egységesen elsőrendű logikai kifejezésekké fordítjuk, amelyeken SMT megoldókkal végzünk következtetéseket. (ii) Approximációs technikákat alkalmazva egy hatékonyan elemezhető logikai fregmensbe képezzük az komplexebb nyelvi elemeket. (iii) A validációs eszközünket ipari modellező eszközhöz integráltuk, amely az ellentmondásokat a nyelv szabványos példánymodelljeiként állítja elő.

Módszerünk magja egy olyan leképezésen alapszik, amely egy származtatott attribútumokkal és relációkkal gazdagított EMF metamodellt, OCL vagy EMF-IncQuery nyelven definiált jólformáltsági kényszereket és egy hiányos kezdeti példánymodellt vár bemenetül. Az eszköz a kezdeti modellt kiegészíti új elemek felvételével a generált axiómák és a Z3 SMT megoldó által ismert elméletek alapján, úgy, hogy az eredmény megfeleljen a nyelv specifikációjának.

Az eszközünket két ipari követelményekkel rendelkező esettanulmányon is sikerrel alkalmaztuk. Egy brazil repülőgépgyártóval közös projektben EMF-IncQuery gráfmintákkal megfogalmazott származtatott értékekkel és jólformáltsági kényszerekkel gazdagított EMF metamodell konzisztencia vizsgálata volt a cél, hogy a fejlesztés korai szakaszában detektáljuk a nyelv hibáit. Az R3COP ARTEMIS esettanulmányban biztonságkritikus autonóm rendszerek (pl. ipari robotok) tesztelésének támogatása a cél, ahol az eszközünk feladata a konkrét tesztesetek előállítása volt az OCL kényszerekkel meghatározott absztrakt tesztleírások alapján.

Abstract

Complex design environments based on Domain-Specific Languages (DSLs) are widely used in various phases of model driven development from specification to testing in order to capture the main concepts and relations in the application domain. A precise system model captured in a DSL enables formal analysis and automated code or test generation of proven quality. Unfortunately, the specification of DSL may itself contain conceptual flaws, which invalidates the results of subsequent formal analysis of the system model. The main objective of the current report is to provide formal analysis of a DSL itself to highlight inconsistency, incompleteness or ambiguity in DSL specifications.

However, the consistency analysis of DSLs is a difficult task due to (i) decidability problems of handling complex DSLs, (ii) theoretical challenges of supporting well-formedness constraints and derived features, and (iii) the engineering problem of providing a DSL validation tool that is operable by the DSL developer without any extra validation skills.

In this report, we address these challenges by providing (i) a mapping of well-formedness rules and derived features formulated in different constraint languages into first-order logic theories processed by SMT-solvers, (ii) powerful approximations to map complex structures into an efficiently analyzable fragment of first order logic, and (iii) a DSL validation tool seamlessly integrated into industrial modeling frameworks (EMF) where inconsistencies retrieved by SMT-solvers are available as regular DSL instance models.

Our DSL validation framework is based on a mapping, which takes an EMF metamodel with derived features, a set of well-formedness constraints (captured in OCL or graph patterns of EMF-IncQuery) and a partial model as input. This partial model is completed by introducing new model elements to it which are compliant with the DSL specification using the generated axioms and underlying theories of the Z3 SMT-solver in the background.

We report on successful use of our validation framework in two complex case studies with industrial requirements. In a collaborative project with a Brazilian airframer, the consistency of EMF metamodels augmented with well-formedness constraints and derived features defined by IncQuery graph patterns is checked to detect design flaws in the early phase of the DSL development. The case study of the R3COP ARTEMIS project that aims to develop safety critical autonomous systems like industrial robots. Our validation framework supported the automatic generation of concrete test cases from abstract test properties defined in standard OCL.

Contents

Chapter 1

Introduction

The design of integrated development environments (IDEs) for complex domain-specific languages (DSL) is still a challenging task nowadays. Generative environments like the Eclipse Modeling Framework (EMF) [50], Xtext or the Graphical Modeling Framework (GMF) significantly improve productivity by automating the production of rich editor features (e.g. syntax highlighting, auto-completion, etc.) to enhance modeling for domain experts. Furthermore, there is efficient tool support for validating well-formedness constraints and design rules over large model instances of the DSL using tools like Eclipse OCL [57] or EMF-IncQuery [9]. As a result, Eclipse-based IDEs are widely used in the industry in various domains including business modeling, avionics or automotive.

1.1 Problem statement

However, in case of complex, standardized industrial domains (like ARINC 653 [6] for avionics or AUTOSAR [7] in automotive), the sheer complexity of the DSL is a major challenge itself. (1) First, there are hundreds of well-formedness constraints and design rules defined by those standards, and due to the lack of validation, there is no guarantee for their consistency or unambiguity. (2) Moreover, domain metamodels are frequently extended by derived features, which serve as automatically calculated shortcuts for accessing or navigating models in a more straightforward way. In many practical cases, these features are not defined by the underlying standards but introduced during the construction of the DSL environment for efficiency reasons. Anyhow, the specification of derived features can also be inconsistent, ambiguous or incomplete. (3) In general, a reusable method for validating different requirements of complex domain specific languages in a mathematically precise way.

1.2 Research Context

As model-driven tools are frequently used in critical systems design to detect conceptual flaws of the system model early in the development process to decrease verification and validation (V&V) costs, those tools should be validated with the same level of scrutiny as the underlying system tools as part of a software tool qualification process issues in order to provide trust in their output. Therefore software tool qualification raises several challenges for building trusted DSL tools in a specific domain.

1.3 Objectives

The main objective of this work is to create an automated framework to formalize DSL modeling artifacts (including meta- and instance models, constraints and derived feature definitions) by logic descriptions in order to be able to execute wide range of validation task by automated theorem proving on it. Afterwards we would like to improve the quality of the developed DSL by validating different requirements of the domain specific language such as consistency, ambiguity and completeness. Additionally we aim to decrease the development time and cost by detecting design flaws in the early phase of DSL development and highlight reason of failure to the developer. Finally we plan to automate other development activities by generating instance models with required features. (Like automated test case generation.)

1.4 Contribution

We propose an approach for the validation of domain specific languages which covers the handling of metamodels, well-formedness constraints and derived features captured as model queries. The essence of the approach is to prove consistency and completeness of language specifications by mapping it preferably to an efficiently analyzable fragment of first order logic formulae processed by state-of-the-art SMT solvers. We also propose powerful approximation techniques to handle complex language constructs. It is carried out by completing a prototypical initial instance models (called partial snapshots) in accordance with the DSL specification.

We also developed a research prototype tool to demonstrate the practical feasibility of our approach. Our tool takes DSL specifications int 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) and declarative graph patterns as available in the EMF-IncQuery framework. We integrated the Z3 SMT solver, which is considered to be the most powerful theorem prover built on high-level decision procedures. The validation results are back-annotated to the source DSL specification and to the initial partial model therefore language engineers and domain experts may inspect those results directly in existing model editors as a regular instance model.

Our tool has been successfully applied in case studies of two ongoing industrial projects taken from the avionics and autonomous and cooperative robot system domain. We have carried out initial experiments to assess the performance characteristics of our validation tool. The first functional tests have been executed to assure the correct behavior of the core mapping and also the back-annotation process.

1.5 Structure of the Report

The rest of the report is structured as follows. First the motivating scenarios will be presented in Chapter 2. In Chapter 3 we summarize the theoretical and technical background of this work. Afterwards in Chapter 4 we give a brief overview of the proposed DSL validation approach. As a follow-up the case studies are detailed in Chapters 5 and 6. Chapter 7 presents the novel features of the mapping, while the implementation details, validation and testing aspects of our work are shown in Chapter 8. Finally we conclude the report in Chapter 9.

Chapter 2

Motivating Scenarios and Requirements

In this chapter two different motivating scenarios are presented: (i) the Trans-IMA an MDE based HW-SW allocation project within the avionics domain, and the (ii) the Artemis R3Cop European research project that defines an automated test-case generation for autonomous robots. Common in these two examples that a satisfiability check of the DSL can uniformly provide valuable result on their metamodels such as (a) the unsatisfiability of their language features as it demonstrates inconsistency in their metamodels, and (b) their satisfiability that provides example instance models which can be used as executable test cases.

2.1 DSL Development of Trans-IMA

Trans-IMA aims at defining a model-driven approach for the synthesis of complex, integrated Matlab Simulink models capable of simulating the software and hardware architecture of an airplane. The project aims to: (i) define a model-driven development process for allocating software functions captured as Simulink models[36] over different hardware architectures and (ii) develop an MDE based tooling platform for supporting the definition of the allocation process.

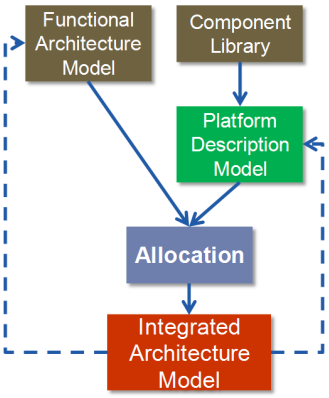


Figure 2.1: *High-Level Overview of the Trans-IMA project*

The high-level overview of the Trans-IMA development process is illustrated in Figure 2.1. The input artifacts for the process are the Functional Architecture Model (for capturing the functional description of different systems) and the Component Library (that defines the available hardware elements).

First the system architect specifies the Platform Architecture Model from the elements of the Component library. Based on the hardware design in the next step the system architect allocates

the functions from the Functional Architecture Model. The allocation itself includes two major parts: (i) the mapping of functions defined in the FAM to their underlying execution elements within the PDM and (ii) the automated discovery of available communication paths for the various information links defined between the allocated FAM elements.

Finally, when the allocation is complete and fulfils all safety and design requirements the Integrated Architecture Model is automatically synthesised and ready to be simulated in Simulink.

This development environment is defined by eight large metamodels (more than 200 elements), where complex EMF-IncQuery patterns are extensively used. The definition of such large DSLs is a very challenging task not only due to their size (and thus complexity) but also to precisely understand their interaction defined using a large number of derived features and also the relation of these derived features relation to the specific safety related well-formedness constraints.

The DSL validation approach is illustrated in Chapter 5 on the simplified metamodel of the Functional Architectural Model.

2.2 Test Generation for R3-Cop

One of the most important industrial related motivation is the Robust & Safe Mobile Co-operative Autonomous Systems (R3Cop) European Union project. The aim of the project is a model based test generation for autonomous agents, based on the information about their context and the described requirements.

Figure 2.2

A high-level overview of the test data generation is depicted in Figure 2.3. The approach uses the context model of the system under test constructed by domain experts, and represents test data as model instances conforming to this metamodel. The test data generation algorithm is based on search-based software engineering and uses search technique to find relevant and high quality test data. The test strategy is used as input for the test data generation, that specifies the required test data (e.g. specifies coverage criteria or prescibes requirements for robustness testing).

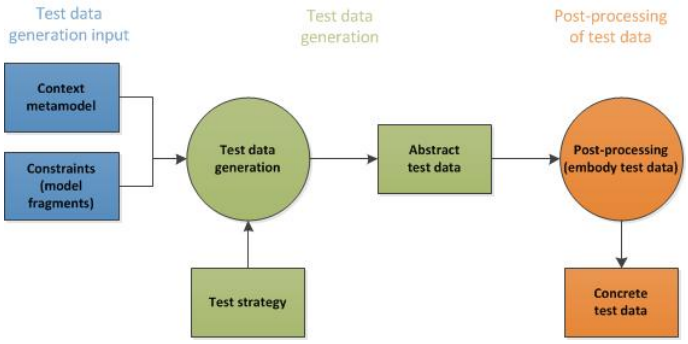


Figure 2.3: *Test generation scenario*

In order to deal with the size of the model-space during the search-based generation and ensure the flexibility of the requirement specification abstraction is used on the metamodel. First an "abstract" test data is constructed, that specifies only "abstract" level attributes or relation (e.g. big, near, after) instead of concrete values. Finally, a post-processing step replaces abstract

elements in the model with actual elements from an object library, and assigns real values to the attributes based on the abstract relations.

This report is motivated by the previously mentioned post-processing step, where a model is given including predifining constraints for different model elements. This model should be transformed to a new model, that is a valid model and fulfills all the requirements of the metamodel and defined constraints.

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, 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, which metamodel is presented in the next section.

A generated test case is represented using an instance model of this metamodel, that fulfills the requirements described by the metamodel and also the OCL constraints. Due to the two phased test generation, first an abstract test data is constructed, where some OCL constraints can be violated or abstract model elements can be used. The goal of our work is, to replace this abstract elements, specify the attributes using concrete values and produce a concrete test data (instance model) that fulfills all the requirements.

Chapter 3

Preliminaries

In this chapter the most important theoretical concepts are presented which are necessary for understanding. First the definitions of modeling, the attributes of the metamodels and instance models are introduced. Afterwards the well-formedness constraints and components (Object Constraint Language, Graph Pattern and Derived Feature) are presented which can be formulated extra rules on the model. Finally the mathematical problem solvers, the problem classes (SAT, CSP, SMT) shown and the related work.

3.1 Modeling, Models

In this section the core concept of model-based development will be introduced.

3.1.1 Metamodel

Metamodels are the models of the modeling languages. They are used to collect the concepts. their attributes and relations in the target domain. For example in the R3Cop project the metamodel defines the structure of the warehouse, the type of elements, their relations and attributes.

In this project, the Eclipse Modeling Framework (EMF) [50] tools are used for domain specific modeling and implementation framework. The most important components of the EMF are the metamodels, that are used to define the elements of the target domain. First, classes are defined, with their attributes, which are mentioned the property of the class. Between classes inheritance and references can be defined. The types of the reference can be association and composition and their multiplicity can also be defined which should be between 0 and infinite. References can be directed or bidirectional, which is defined using 2 references.

Figure 3.1 shows the element of the metamodel of LGV. In this example the most important elements of the metamodeling technique are represented: a world contains layouts and placed objects (relation), the layout has a right bottom and a left up position and contains stations (relation). All stations have one position and a placed object, which are placed on it (relations), the placed objects have one or more neighbours which are symbolized by the near reference (relation). The placed object has position (relations) and the position has an x and an y coordinates (attributes).

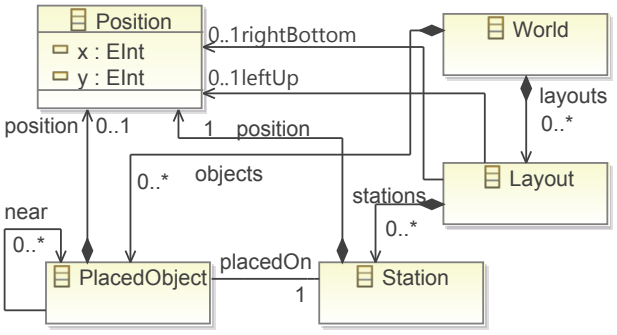


Figure 3.1: *Element of the metamodel of LGV*

3.1.2 Instance Model

The instance model is an instance of the metamodel, it is a specific realization of the defined concepts. During modeling instance objects are created specified the instance of the classes defined in the metamodel, they are named and their attributes are. The template of instance models are metamodels. The components are same as the elements of the metamodel, but the attributes are specified, the objects are concrete and the references link objects. A model is valid if a world is created, it contains a placed object and a layout, the layout has positions and a station, it has a placed object and a position and the placed object has a position. The coordinates of the position are specified.

3.2 Model Query Languages

In this part the languages of well-formedness constraints are introduced. To formulated the constraints the Object Constraint Language, Derived Features and Graph Patterns are used.

3.2.1 Object Constraint Language

The Object Constraint Language (OCL) [22] [21] [38] is a declarative language, that extends the structural metamodels using extra constraints. OCL refers to the models and it defines extra constraints, rules and validates the metamodels.

Two types of OCL constraint can be distinguished. There are logical expression, which returns with true or false. The other types is an expression which can be evaluated to with a number, or collection of model elements. The others are constraints of method calls. Every OCL constraint defines a context which points at the container class of the expression and defines the limited situation in which the statement is valid.

There are many language elements, with them various constraints can be prepared. The same constraint can also be formulated in several ways. The elements of the language are various, containing collections, iterators, variables, functions, logical formulas, etc.

Example 1 The two types of the OCL expressions:

- Constraints of a method

```
context Person::birthday() post: self.age=self.age@pre + 1
```

- Invariants

```
context Vechile, invariant numberOfPassengers:
  self.maxPassengers >= self.traveller->size()
```

The first example expresses that the age is increased when the birthday() method is called. The second means that the capacity of the vehicle must be bigger than the number of passengers.

The OCL language has a well-defined syntax. Each expression can be represented using its Abstract Syntax Tree which can be traversed. Each AST node represents an OCL element and the edges define the relations between them. The AST representation of the OCL expression is unambiguous. The AST of the `A.allInstances()->forAll(a: A | a.time<10)` OCL expression is shown as an example in Figure 3.2.

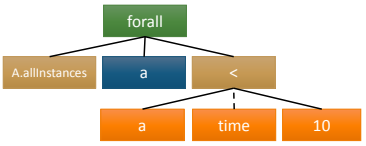


Figure 3.2: *OCL Abstract Syntax Tree example*

In this report we deal with the subset of invariants, which are logical expressions that need to hold at anytime during the lifecycle of an object. These expressions are usually used to add extra constraints to the metamodels and based on the semantic, unspecified missing attributes could be filled out, relations or objects can be added to the model. The queries and the other types of the invariants are not suitable for this, because they can not be used to validate models and the attribute filling is also impossible.

Example 2 In the following examples of the different elements of the OCL are presented. The examples are derived from the Elettric80 example.

Expression	Description
`p`	`p` is a variable.
`null, 5, 10.2, etc.`	Different type of literals.
`p.oclIsTypeOf(Truck)`	If `p` is a Truck then it return true.
`Position.allInstances()`	It is return the collection of Positions.
`p.position`	The `position` is a reference.
`p.position.y`	The `y` is an attribute.
`p.y=q.y`	The "=" is an operation.
`Event.allInstances()->forAll(..)`	The `forAll` is called on a collection.
`forAll(p: Positon| ..)`	The `p` is a variable
`forAll(p: Positon| p.y>=0)`	The body of the `forAll`.
`exists(p: Positon| p.y=10)`	The body of an `exists`.

3.2.2 EMF-IncQuery Graph Patterns

Graph patterns [55] are an expressive formalism used for various purposes in model-driven development, such as defining declarative model transformation rules, capturing general-purpose model queries including model validation constraints, or defining the behavioral semantics of dynamic domain-specific languages. A graph pattern (GP) represents conditions (or constraints) that have to be fulfilled by a part of the instance model. A basic graph pattern consists of structural constraints prescribing the existence of nodes and edges of a given type, as well as expressions to define attribute constraints. A negative application condition (NAC) defines cases

when the original pattern is not valid (even if all other constraints are met), in the form of a negative sub-pattern. A match of a graph pattern is a group of model elements that have the exact same configuration as the pattern, satisfying all the constraints (except for NACs, which must not be satisfied). The complete query language of the EMF-IncQuery framework is described in [10], while several examples will be given below [28].

3.2.3 Derived Feature

Derived features (DF) are often essential extensions of metamodels to improve navigation, provide path compression or compute derived attributes. The value of these features can be computed from other parts of the model by a model query [44, 38]. Such queries have two parameters, in case of (i) derived EReferences one parameter represents the source and another the target EObjects of the reference while in case of (ii) derived EAttributes one parameter represents the container EObject while the other one the computed value of its attribute.

3.3 Mathematical Logic

In this part the First Order Logic, the classes of problem and their provers are resented.

3.3.1 First Order Logic

First-order logic(FOL) is a formal language used in mathematics, philosophy and computer science. In the 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: (i.)constant symbol which symbolize the object, (ii.)predicate symbol which sign the relation and (iii.)function symbol which refer 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 need which link the real objects and the symbols.

The syntax of the FOL in Backus-Naur Form:

Sentence	$\rightarrow$	Atomic Sentence
	|	(Sentence Connector Sentence)
	|	Quantifier Variable, ... Sentence
	|	$\neg$Sentence
Atomic Sentence	$\rightarrow$	Predicate(Term) | Term=Term
Term	$\rightarrow$	Function(Term, ...)
	|	Constant
	|	Variable
Connector	$\rightarrow$	$\Longrightarrow$ | $\bigwedge$ | $\bigvee$ | $\Leftrightarrow$
Quantifier	$\rightarrow$	$\forall$ | $\exists$
Constant	$\rightarrow$	A | X
Variable	$\rightarrow$	a | x
Predicate	$\rightarrow$	Before | Raining | TypeOf | ...
Function	$\rightarrow$	TimeStamp() | Next() | ...

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 component 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 the quantifiers, which symbolize the objects. The $\forall$ x P(x) means that P is true of every object x. The $\exists$ x P(x) means that exists x which make P true. In the FOL equality symbol can be used to create statement. The sentences are added to the knowledge base with assertion.

Example 3 In the following, examples of the different type of syntax element are shown, which derive from the mapping of the Elettric80 example.

Example	Meaning
`myTruck`	Constant, the name of a truck.
`isType!Truck(myTruck)`	If myTruck is a truck it returns true.
`PlacedObject!placedOn(myT, myS)`	If myS is the station of the myT it returns true.
$\forall$ `x isType!Truck(x)`	If x is a truck it returns true.
$\forall$ `t isType!Truck(t)` $\Leftrightarrow \neg$ `isType!Human(t)`	It represents if t is a truck then t is not human.

3.3.2 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 ($x_1, x_2, \ldots x_n$), their negated expressions, the $\bigwedge$ ("and") and the $\bigvee$ ("or") operands. 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 has an evaluation where the value of the formula is 1. The SAT is the language of the satisfiable Boole-formulas. The SAT language is in the NP class: a good evaluation is the witness of satisfiability of the formula. The SAT has subsections which the conclusion is effective e.g. the 2-SAT is polinomial. 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 [37].

Constraint Satisfaction Problem (CSP)

The formal definition of the CSP problem are formulated by the set of variables (X_1, $X_2 \ldots X_n$) and constraints (C_1, C_2, $\ldots C_m$). All X_k variables have a D_i domain which define the possible values. Every C_l constraint restrict the subset of variables, it 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. A 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 differences from the SAT because predicates over suitable set of non-binary variables are used to. SMT formulas provide richer language than is possible with the SAT formulas.

SMT is used to software verification, planning, model checking and automated test generation. The interest theories in these applications include formalizations of arithmetic, arrays, algebraic datatypes, functions. The SMT solvers use the standard SMT-LIB [3] language. SMT solvers are e.g the Alt-Ergo [34], Barcelogic [49], Beaver [31], CVC4 [1], Mistral [54], SONOLAR [23], Yices [4], Z3 [20], from them we use the Z3.

Microsoft Z3 is a theorem prover from Microsoft Research. It can be used to check the satisfiability of logical formulas. Z3 is a good match for software analysis and verification tools because common software constructs map directly into supported theories. Built-in theories are the linear arithmetic, nonlinear arithmetic, bitvectors, arrays, datatypes and quantifiers.

3.4 Related work

There are several approaches and tools aiming to validate UML models enriched with OCL constraints [26] relying upon different logic formalisms such as constraint logic programming [16, 17, 12], SAT-based model finders (like Alloy) [5, 14, 33, 48], first-order logic [8, 19], constructive query containment [43], higher-order logic [11, 27], or rewriting logics [18]. Some of these approaches (like e.g. [17, 14, 33]) offer bounded validation (where the user needs to explicitly restrict the search space), others (like [19, 11, 8]) allows unbounded verification (which normally results in increased level of user interaction and decidability issues).

SMT-solvers have also been used to verify declarative ATL transformations [13] allowing the use of an efficiently analyzable fragment of OCL [19]. The FORMULA tool also uses the Z3 SMT-solver as underlying engine, e.g. to reason about metamodeling frameworks [29] where proof goals are encoded as CLP satisfiability problem. The main advantage of using SMT solvers is that it is refutationally complete for quantified formulas of uninterpreted and almost uninterpreted functions and efficiently solvable for a rich subset of logic. Our approach uses SMT-solvers both in a constructive way to find counter examples (model finding) as well as for proving theorems. In case of using approximations for rich query features, our approach converges to bounded verification techniques.

One of the most relevant mapping from a subset of OCL into first order logic is presented in [19], that proposes an approach using theorem provers and SMT solvers to automatically check the unsatisfiability of non-trivial sets of OCL constraints without generating the SMT code.

Graph constraints are used in [58] as means to formalize a restricted class of OCL constraints in order to find valid model instances by graph grammars. An inverse approach is taken in [15] to formalize graph transformation rules by OCL constraints as an intermediate language and carry out verification of transformations in UML-to-CSP tool. These approaches mainly focus on mapping core graph transformation semantics, but does not cover many rich query features of the EMF-IncQuery language (such as transitive closure and recursive pattern calls). Many ideas are shared with approaches aiming to verify model transformations [15, 35, 13], as they built upon the semantics of source and target languages to prove or refute properties of the model transformation.

The idea of using *partial models*, which are extended to valid models during verification also appears in [47, 29, 32]. These initial hints are provided manually to the verification process, while in our approach, these models are assembled from a previous (failed) verification run in an iterative way (and not fully manually). *Approximations* are used in [30] to propose a type system and type inference algorithm for assigning semantic types to constraint variables to detect specification errors in declarative languages with constraints.

Our approach is different from existing approaches as it can use different approaches (is implemented with graph based query language and also OCL) for capturing derived features and well-formedness constraints. Up to our best knowledge, this is the first approach aiming to validate queries captured within the EMF-IncQuery framework, and the handling of derived features is rarely considered. Furthermore, we sketch an iterative validation process how DSL specifications can be carried out. Finally, we also cover the validation of rich language features (such as recursive patterns or transitive closure) which is not covered by existing (OCL-based) approaches.

Chapter 4

Overview of the Approach

4.1 Functional View of the Approach

Our approach (as illustrated in Figure 4.1) aims to analyse DSL artifacts of modelling tools by mapping them into first order logic formulae that can be processed by advanced reasoning applications. The results of the reasoning is traced back and interpreted in modelling terms as attributes of the DSLs. Linking the independent reasoning tool to the modelling one allows the DSL developer to make mathematically precise deductions over the developed models including different validation techniques and example generations.

DSL development tools like EMF usually specify strictly two meta-levels: a language level that defines the abstract syntax of the DSL and an instance level where concrete instance models can be defined. To define the valid models more precisely the language model can be supplemented with derived features and some ill- or well-formedness constraints that forbids or requires some kind of structure. (see in Chapter 3)

Similarly in the terminology of the the reasoning tools (like Z3) SMT solver this two levels can be defined too: the specification of the system creates the axioms of the in language level where the consistency of the language can be checked, or different properties of the language can be proved as a theorem proving problem. By definition, consistent logic systems have logic model and failed theorems have counterexamples. Those logic structures can be recovered and represented as a standard instance model of the DSL.

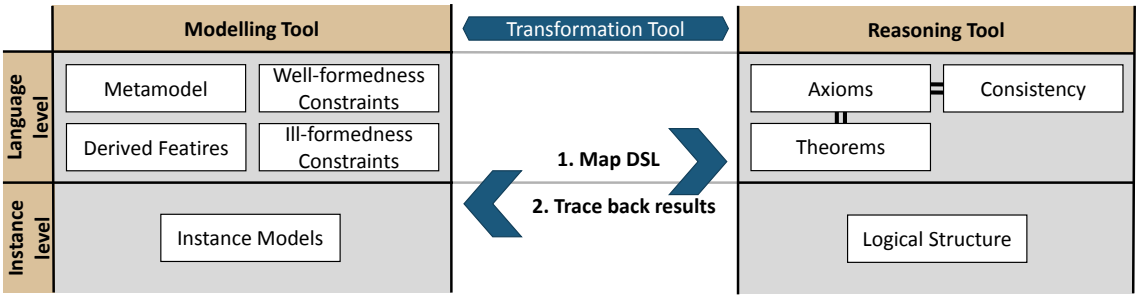


Figure 4.1: *Functional overview of the approach*

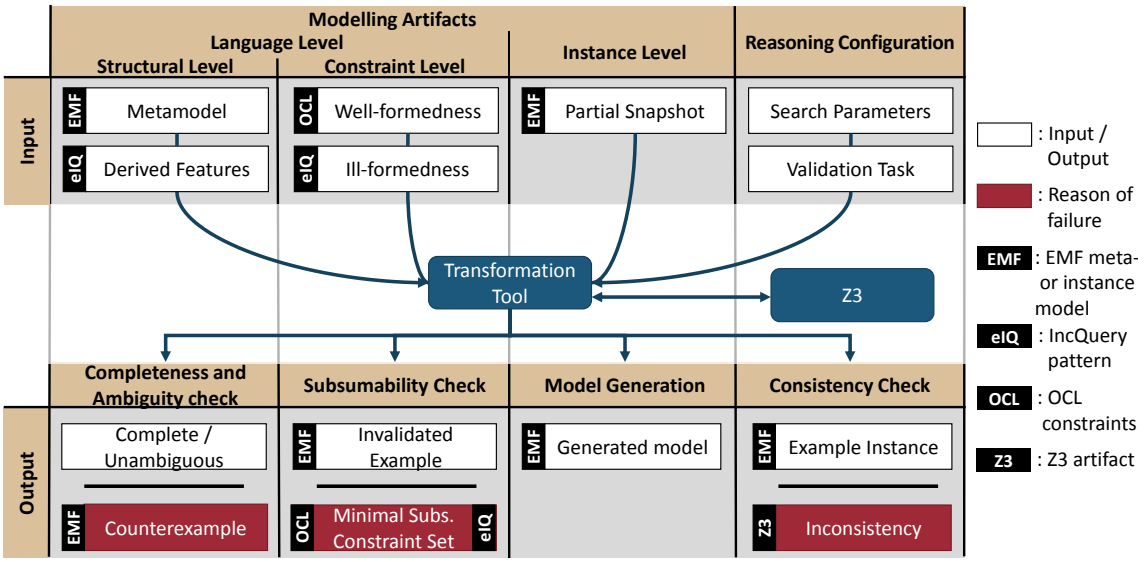


Figure 4.2: *Prototype tool features*

4.2 Input Configuration

Figure 4.2 shows a more detailed figure about the input parametrization of our tool (upper part), the implemented tasks and possible outputs (lower part) of our tool. The language level is divided to a Structural Level that define the language in a constructive way, and a Constraint Level that restricts it. In the following those will be introduced.

Parameters in the Structural Level refer to the actual target of the reasoning process. It is important to note that the input elements are fully functional standard artifacts of the modelling tool.

Metamodel The metamodel contains the main concepts and relations of the DSL 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, the some structural constraint might be specified with the elements like multiplicity.

Derived Features The classes of the metamodel may contain some derived features: attributes or references that can not be edited but automatically calculated from the rest of the model. The model query frameworks (like EMF-IncQuery) can be used to specify evaluate the values of the derived features by declarative queries. Those queries can be translated to logic formulae too so the reasoning tool would handle them as the modelling tool.

To more precisely specify the range of valid instance models different constraints might be added to the DSL. Those constraints can be included to the Constraint Level of the reasoning phase to make formal analysis over them.

Well-formedness The goal of the well-formedness constraints is to define rules that have to be satisfied in a valid model.

Ill-formedness Ill-formedness constraints can be defined to specify faulty model structure. A valid model is free from those fault-patterns.

Analysing purely the language level might be insufficient in some cases: (i) theorem proving problems might derive spurious false positives and (ii) featureless examples might be generated. The search should be controlled by some practical preconditions.

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

Our 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.

The parameters in the Reasoning Level allows to customise the reasoning process. Beside the few technical details like approximation level time limits the following parameters are the most important:

Search Parameters To more precisely control the reasoning 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 (like overapproximation level).

Validation Task The tool capable of multiple reasoning task including different validations, theorem proving or model generation. Those task can be selected and parametrized here. Those tasks are described in the following sections.

4.3 Validation Tasks

4.3.1 General Reasoning

Generally, our tool works as follows. The tool searches for an instance model which:

- Instance of the Metamodel and satisfies every structural constraints including the Derived Features
- Satisfies all the Constraints
- Contains the Partial Snapshot
- Satisfies every Search Parameter

If our tool finds such a model, then it will be demonstrated to the user. If the input is inconsistent, the tool should prove that those requirements are unsatisfiable. Because the validation task is undecidable it is also possible that the tool results with “unknown” or “timeout”.

This general process is applied in each reasoning task with some modifications.

4.3.2 Completeness and Ambiguity Check of Derived Features

Derived features specified by EMF-IncQuery patterns are integrated part of the DSL. By formalising the definition of the patterns some well-behaving property can be proved. In addition, a failed validation attempts will reveal a design flaw of the language, currently we check the completeness and unambiguousness of the DFs.

We understand completeness as follows:

Definition 1 (Completeness of Derived Features) *A derived feature is* ***complete*** *if it evaluates to at least one value for every occurrence of the derived feature.*
Conditional completeness *is when the derived feature requires some additional condition to be complete.*
A derived feature is ***incomplete*** *if there is a valid model there where no values can be assigned to an occurrence of the derived feature.*

The completeness requirement of a derived attribute or a reference is usually indicated with a 1..? multiplicity.

We define the unambiguity similarly:

Definition 2 (Unambiguity of Derived Features) *A derived feature is a* ***unambiguous*** *if it evaluates at most one value for every occurrence of the derived feature.*
Conditional unambiguity *is when the derived feature requires some additional condition to be unambiguous.*
A derived feature is ***ambiguous*** *if there is a valid model there where multiple values can be assigned to an occurrence of the derived feature.*

The unambiguity requirement of a derived attribute or a reference is usually indicated with a ?..1 multiplicity.

Our tool can check the previous properties. Figure 4.3 shows the setup of the DF validation. The validation uses the general setup with the following exceptions:

- Instance of the Metamodel and satisfies every structural constraints including the inspected DF (and the other DFs) but excluding the multiplicity requirement[1]
- There is an instance of the source that violates the multiplicity constraint

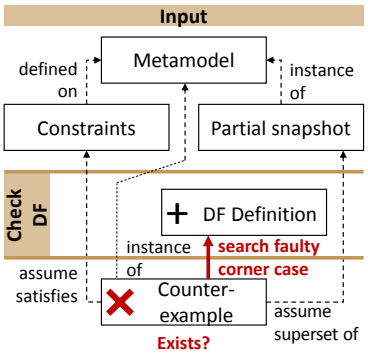


Figure 4.3: *Derived feature validation setup*

The result of the validation task could be the proof of conditional completeness / unambiguity of the checked DF with respect to the Partial Snapshot and the Search Parameters, or a valid counterexample that shows the failed instance model.

If the Partial Snapshot and the Search Parameters do not limit the search (like empty PS) then the full completeness / unambiguity is proved. By setting the PS or the Search Parameters tool would generate various counterexamples.

[1] Not excluding the structural constraint from the multiplicity would cause "short circuit" with the last point: multiplicity satisfied ↔ multiplicity violated

4.4 Subsumability Check

A complex DSL may contains several independent well- or ill-fordmedness constraints that globally restrict the developed language. It would be very profitable if their interaction would be analysable. The two basic invalid interaction is where a constraint contradicts to the DSL and where it is subsumable from the DSL. This section focuses on subsumability, the contradiction is discussed in Section 4.6.

It is important to note that this is the traditional theorem proving scenario, where the theorem is defined by a new constraint and the task is to deduce that the new constraint is implication of the DSL specification.

We define subsumability as follows:

Definition 3 (Subsumability of a Constraint) *A constraint is **subsumable** by a DSL specification if every valid model that would satisfy the DSL specification satisfy this constraint too.*
***Conditional subsubtion** is when the constraint is subsumable if additional condition holds.*
*A constraint is **not subsumed** by a DSL specification if there is a valid instance model that does not satisfy the constraint.*

A subsumable constraint does not express any additional restriction over the DSL therefore it can be removed without any change. A subsumable constraint is considered superfluous.

Our tool can perform subsumption checks for a target constraint in the setup that Figure 4.4 shows. Basically it follows the general setup with the following addition:

- The instance model does not satisfy the target constraint.

The result shows that the target is conditionally subsumable if the Partial snapshot and the Search Parameters holds. In case of valid constraint it also give example that shows that the target is not subsumable. Global subsumality check can be performed if the PS and the Search parameters does not limit the search.

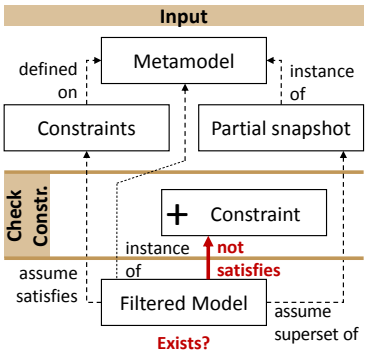


Figure 4.4: *Subsumability check setup*

4.5 Model Generation

Our tool can be specialised to generate instance models of the chosen DSL. The Partial Snapshots and the flexible model size limit makes model generation be highly customisable. Additionally the case study described in Chapter 6 introduces even more advanced methods.

The setup for model generation is the same as the general reasoning task represented in Figure 4.5.

The result is a valid instance model that satisfies the hints of the user drafted in the Partial Snapshot and the Search Parameters. It is possible that those requirements are unrealizable, in that case the failure is communicated to the user.

4.6 Consistency Check

The final validation scenario is the consistency check. Consistency is a property of the whole DSL that means that there is not any contradiction in its specification. Conflicting constraints may break this property, so they can be detected by a consistency check.

The other use-case of the consistency check is the following: the inconsistency invalidates the result of any language check based on theorem proving (like completeness, ambiguity and subsumability checks).

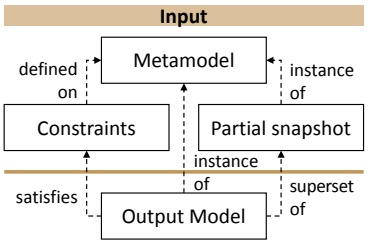


Figure 4.5: *Model Generation setup*

We understand the basic inconsistency as follows:

Definition 4 (Consistency of a DSL) *A DSL is **consistent** if it has a valid instance model.*
*A DSL is **inconsistent** if it is not consistent (so it does not has any valid instance model).*

The setup for model generation is the same as the general reasoning task represented in Figure 4.6.

If there is a result instance model then the DSL is proved to be consistent. If there is not, it shows that the requirements in the Partial Snapshot and the Search Parameters are infeasible. If the Partial Snapshot and the Search Parameters are not limits the search and the tool returns with unsatisfiability than the DSL is proved to be inconsistent.

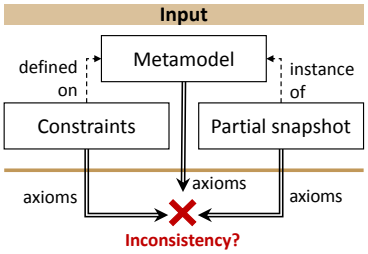


Figure 4.6: *Setup of Consistency check*

It should be noted that the consistency is a minimal property of the language. Harder consistency requirements also can be defined, like every class can be instantiated, or every reference can be used. Those examples also can be checked with our tool using the appropriate Partial Snapshots.

4.7 Partial Snapshot

Unfortunately, a standard EMF instance model is inadequate to act as an initial hint or counterexample because it can not represent incomplete or incorrect initial cases well. To overcome this limitation we created an formalism called Partial Snapshots capable for this role.

The partial snapshot (PS) is a more general instance model than the standard EMF framework allows. To ease the edition, every object is uniquely named. Figure 4.7 presents a Partial snapshot example and a possible completion called Completed model.

1. Undefined attributes: In normal EMF instance object every attribute has a value (or presented default value)[2]. Many use-case need the option to let some of them undefined, so our tool can evaluate them freely. Point 1. shows in Figure 4.7 that the object named function1 has an undefined type attribute that can be filled with the Root literal.

[2] There is an 'unsettable' option in EMF that enables to unset a structural feature. Note that the unset is concrete value opposed to undefined.

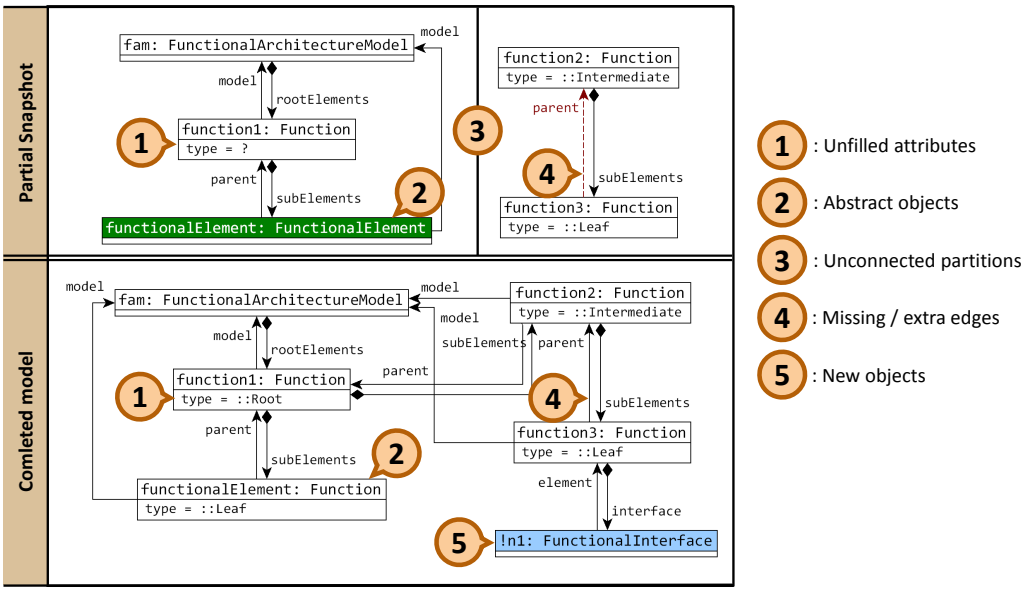


Figure 4.7: *Extra options available in Partial Snapshot*

2. Abstract objects: Partial snapshots allows to instantiate abstract or interface EClasses. They are handled similarly as concrete object like they have attributes and references. The type of an non-concrete object have to be refined in the validation process to a concrete subtype. Point 2. refers to an element with an abstract FunctionalElement type that is refined to the concrete Function.

3. Unconnected partitions: Every EMF instance model is arranged in a strict containment hierarchy. Our approach allows to define instance models that can be unconnected to specify multiple fragments of the model. Point 3. in Figure 4.7 shows an example where there are functions (function2 and function3) that are not yet connected to the FunctionalArchitectureModel. Our tool will complete this model by linking the partitions to be a well-formed connected graph.

4. Missing / extra edges: The Partial Snapshot editor does not automatically manage inverse edges, so it is possible that there is a reference without the inverse one (like in point 4. where the missing reference is indicated with dashed line). In Partial Snapshots the number of references can exceed the bound limit of the edge multiplicities. In that case the PS can not be completed to a valid model.

5. New objects: The Partial Snapshot act as a submodel which can be extended. It can describe models with more object than the object in the PS. For example in point 5. in Figure 4.7 a new FunctionalInterface is added.

4.8 Search Parameters

Model size It is possible to explicitly define the size of the checked models. As Figure 4.8 shows, the range of the checked models can be set to:

1. Initial only: Only the objects in the Partial Snapshot can be used; new objects cannot be created. This option is ideal for simple model-completion tasks.

2. Limited to size: Models with a fixed amount of objects are checked. This option is essential for finding minimal examples.

3. Unlimited: Every model should be checked without having regard to its size. With this option on the prover checks all of the the possible models.

Approximation level Some DSL element (such as the acyclicity of the containment hierarchy) is unrepresentable in the language of the first order logic. To tackle this insufficiency we provided a method to approximate them to some limit called approximation level. 7

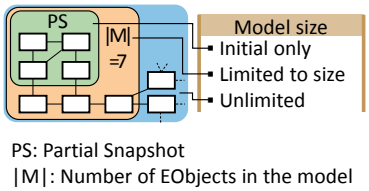


Figure 4.8: *Model Size*

Chapter 5

DSL validation Case Study in Avionics Domain

To illustrate the proposed V&V technique, this report elaborates a case study from DSL tool development for avionics systems. To create an advanced modeling environment, we augment the metamodel with query-based derived features and well-formedness validation rules. Both of these advanced features are defined using model queries. For this purpose we use the language of the EMF-IncQuery framework to define these queries over EMF metamodels.

5.1 DSL Validation Workflow

A DSL usually specifies a quite complex system that may contain multiple design flaw. To assist the developer to find those errors we propose an iterative workflow that defines the practical order of the validation steps. By following this workflow our tool will reveal the design flaws one by one so with the help of the counter examples the source of the error can be easily detected. The iterative steps can be applied on the currently developed language elements as an integrated development task to detect the design errors immediately. Additionally, the workflow can guide the developer through a complete language check.

The workflow illustrated in Figure 5.1 assumes the existence of the metamodel (captured in EMF), its derived features (captured as graph queries) and well-formedness constraints (captured as graph queries or OCL constraints). Basically, the validation process looks like this: first, each DF is investigated by adding them to the formal DSL specification (extending it with one new DF at a time in a predefined order), and then by validating this specification in Z3. Then, WF constraints are validated similarly, by incrementally adding a single WF constraint at each validation step. If one of these step fails then the user have to manually correct the the DSL artifact and continue from the validation of the modified element.

The separation to start the iterative validation process with the derived features and then continue with the WF constraints is based on the observation that each derived feature eliminates a large set of trivial, non-conforming instance models (which are not valid instances of the DSL). Adding a single constraint at a time to the validation problem helps identify the location of errors the solver provides only very restricted traceability information. This eases the refinement in case of an erroneous DF or WF is added in the actual step based on the proof provided by the solver.

The validation fails, if the compiled set of formulas are inconsistent (formally, no models can be constructed within a given search limit). In such a case, the designer needs to either (i) fine-tune

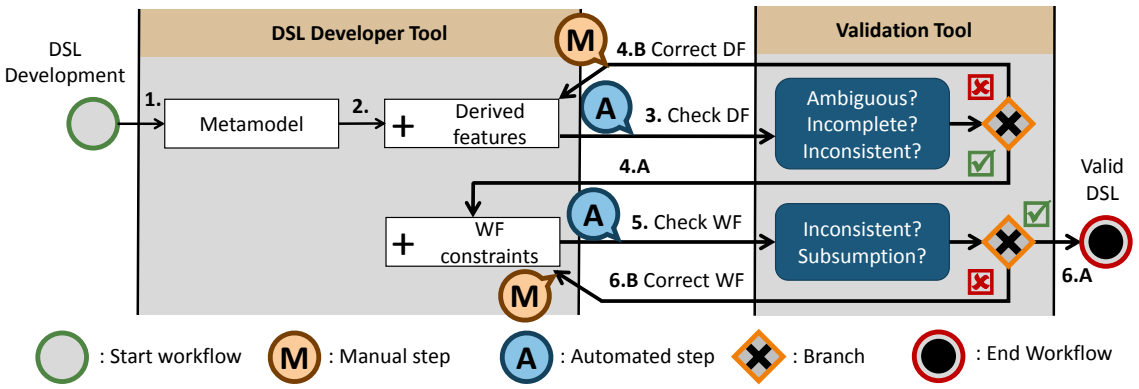


Figure 5.1: *DSL validation workflow*

the search parameters, (ii) provide a new partial snapshot or (iii) modify the DSL specification itself based on the proof outcome. If the formal DSL specification with all DF and WF constraints is validated, then it is valid under the assumptions imposed by the search parameters and the partial snapshot.

The validation process in introduced in details:

1 A metamodel is added to the validation process. A well-formed metamodel is always consistent.

2 Derived features are iteratively added.

3 The ambiguity and the completeness of the DF is automatically checked by our tool. The consistency of the supplemented system is checked too.

4.A When every DF is checked the validation of the WF constraints proceeds. In this phase new constraints are added to the specification iteratively.

4.B If the validation fails, the newly added DF should be corrected based on the counterexamples. In case of false positives or the parametrisation of the tool should be refined.

5 The effect of the constraint to the specification is automatically inspected by our tool.

6.A If every constraint is correct the validation process successfully terminates.

6.B If the validation fails, the newly added constraint should be corrected. In case of false positives or the parametrisation of the tool should be refined.

The rest of this chapter demonstrates how this workflow can be applied on an industrial case study from the avionics domain.

5.2 Introduction to the Domain

In model-driven development of avionics systems, the functional architecture and the platform description of the system are often developed separately to increase reusability. The former defines the services performed by the system and links between functions to indicate dependencies and communication, while the latter describes platform-specific hardware and software components and their interactions. The functional architecture is usually partially imported from industry accepted tools and languages like AADL [45] or Matlab Simulink [36].

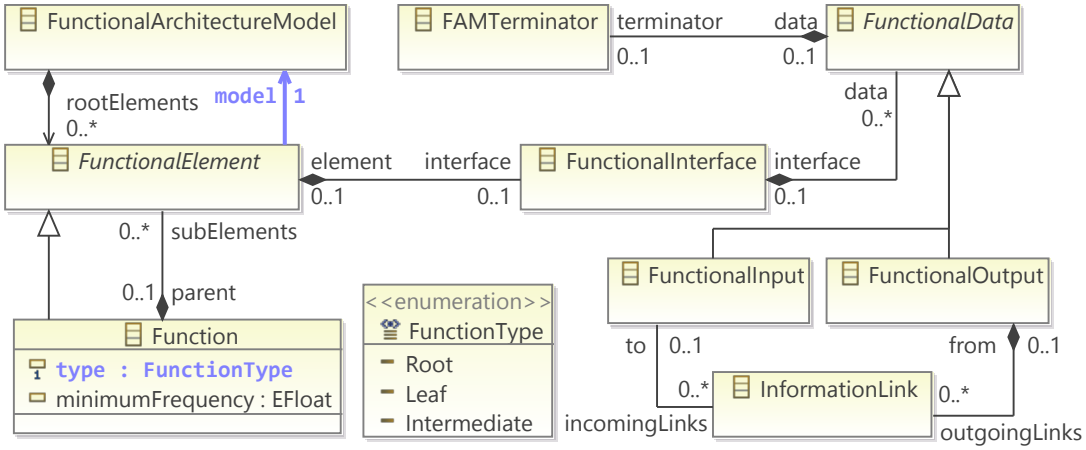


Figure 5.2: *Metamodel for functional architecture of avionics systems*

A simplified metamodel for functional architecture is shown in Figure 5.2. The FunctionalArchitectureModel element represents the root of a model, which contains each Function (subtype of the FunctionalElement). Functions have a minimumFrequency, a type attribute and multiple FunctionalInterfaces, where each functional data is either an FunctionalOutput (for invoking other functions) or an FunctionalInput (for accepting invocations). An output can be connected to an input through an InformationLink.

Additionally two derived feature is added to the DSL (highlighted in blue in Figure 5.2):

- For the type EAttribute of the Function EObject a derived attribute is defined, which takes a value from the enumeration literals: Leaf, Root, Intermediate based on the role of the function in the composition hierarchy.
- FunctionalElements are augmented with the model derived EReference that represents a reference to the container FunctionalArchitctureModel EObject from any FunctionalElement within the containment hierarchy.

Finally, a design constraint is added:

- If an input or output is not connected to an other Function then they must be terminated in a FAMTerminator.

In the following we show how can those rules be validated by our tool.

5.3 Derived Type Validation

The pattern defining the type attribute is illustrated in the right side of Figure 5.3. In Figure 5.3 we use a custom graphical and the EMF-IncQuery textual notation [9] to illustrate the queries defined for these derived features. On the graphical notation each rectangle is a named variable with a declared type, e.g. the variable _Par is as spurious Function, while arrows represent references of the given EReference between the variables, e.g. the function This has the _Par function as its parent. Negative application conditions are illustrated with red rectangles. The

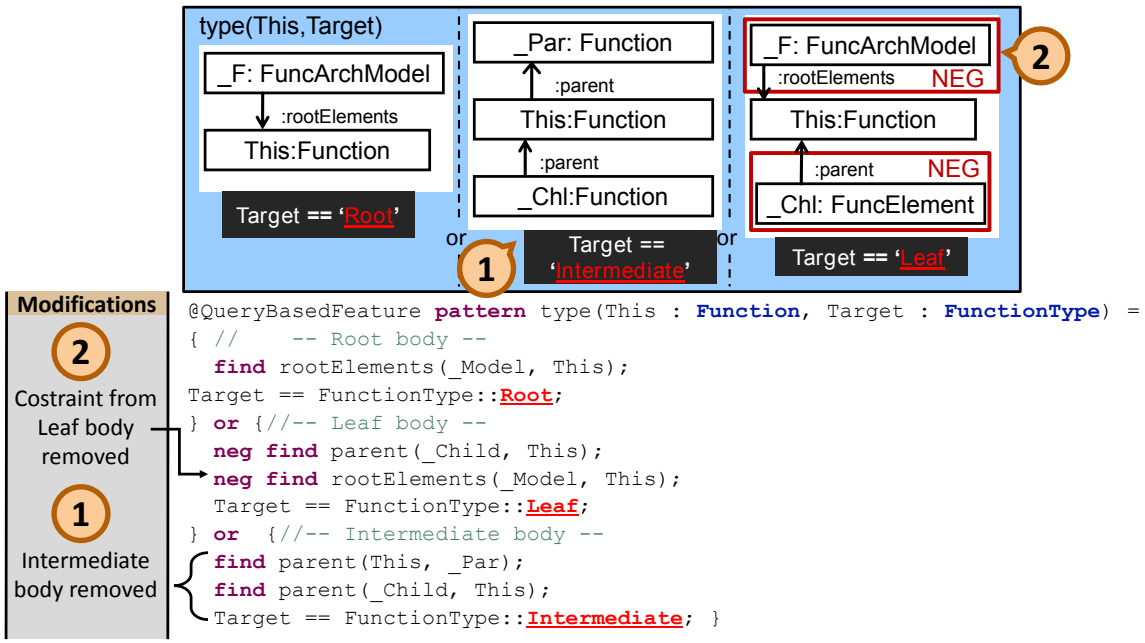


Figure 5.3: *Definition of the type pattern (right) and the illustration of two modifications (left)*

OR pattern bodies represent that the matches of the query is the union of the matches of its or bodies.

Based on these the definition the type query has three OR pattern bodies each defining the value for the corresponding enum literal of the type attribute:

- Leaf if the container EObject does not have a child function along the subFunctions EReference and it is not under the FunctionalArchitectureModel along the rootElements EReference, where both of these constraints are defined using negative application conditions (NEG).
- Root if container EObject is directly under the FunctionalArchitectureModel connected by the rootElements EReference.
- Intermediate if container EObject has both parent and child functions.

To demonstrate our validation tool two modifications had performed on the pattern (also illustrated on the left side of Figure 5.3) to inject hypothetical conceptual flaws into the queries:

1 The pattern body representing the intermediate case has been temporally removed. This will make the derived feature incomplete.

2 The constraint defines that the leaf elements cannot be referred with the rootElements reference is also removed. This will lead to an ambiguity by making the body representing the leaf case more permissive.

The validation process is illustrated in Figure 5.4. First (Step 1) we add the `type` DF to the formal specification and its consistency has been successfully validated.

Then (Step 2), the completeness of the `type` DF is checked resulting in a failure illustrated by the counter example showing three functions without type creating a circle in the containment hierarchy. Our tool visualise the counter example 1 as seen in Figure 5.5, where the invalid elements illustrated by red notation, and the containment references with diamonds.

Validation step	Outcome	Action
1. Consistency: **type**	☑	
2. Completeness: **type**	☒ → CE1	Set acyclicity approximation to 2
3. Completeness: **type**	☒ → CE2	Add missing body to **type** query
4. Completeness: **type**	☑	
5. Unambiguity: **type**	☒ → CE3	Add missing constraint to type query
6. Unambiguity: **type**	☑	

Figure 5.4: *Validation scenario of the type pattern*

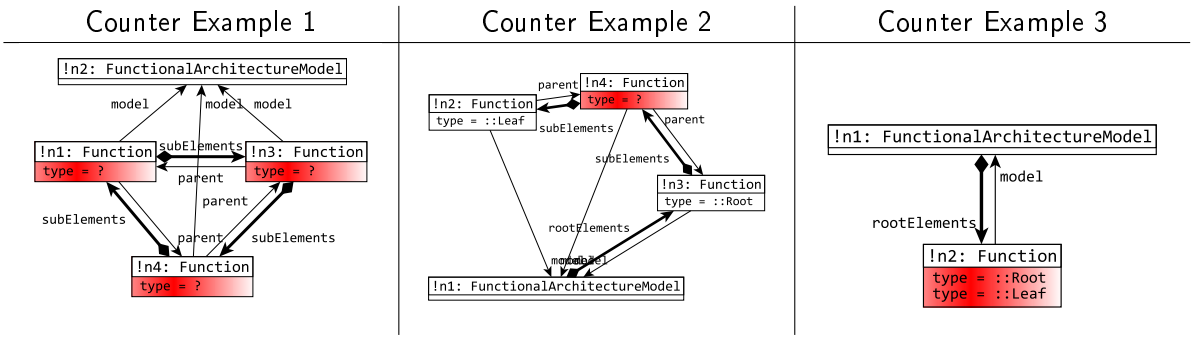


Figure 5.5: *Counter Examples of the type validation*

It is visible (and our tool detects it too) that almost every properties of the instance model is correct but the containment hierarchy is unfortunately violated (`n1-n3-n4` circle), so the example is invalid. It may happen because the acyclicity of the containment hierarchy can only be approximated in first order logic. In our tool this problem can be easily solved by simply raising the level of the transitive acyclicity approximation.

In Step 3 our tool shows a valid counterexample (2nd in Figure 5.5) where an intermediate function (named `n4`) does not have type attribute. This is fixed by adding back the second pattern body with the `Intermediate` definition to the `type` pattern. By correcting it, the validation is successfully executed in Step 4.

After this the ambiguity of the attribute is tested (Step 5), which fails again with a single function node that is both a `Leaf` and a `Root` as a counter example. This counter example is also visible in Figure 5.5. This is fixed by adding the missing NAC condition on the `rootElements` to the third pattern body of `type` in Step 6.

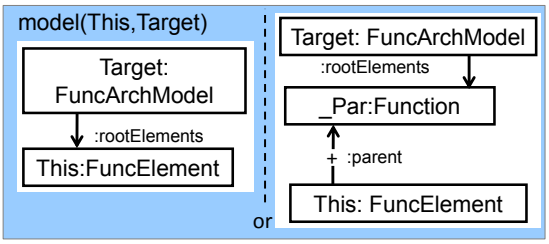
5.4 Derived Reference Validation

This section presents the validation process (visible in Figure 5.6) of the derived feature model that defines a reference to the container FunctionalArchitctureModel from a FunctionalElement. The definition if the pattern visible on Figure 5.7. Transitive closure depicted by an arrow with a + symbol, e.g., the parent reference between the This and _Par.

The validation scenario is illustrated in Figure 5.6. Step 7 adds the `model` DF to the specification, the consistency check executed successfully. Followed in Step 8 with its completeness validation, which fails as pointed out in counter example 4 in Figure 5.8 since a model with a single Function element does not even have anything to refer to with the `model` EReference.

Validation step	Outcome	Action
7. Consistency: **model**	☑	
8. Completeness: **model**	☒ → CE4	Set partial snapshot to PS1
9. Completeness: **model**	Timeout	Checked in boundend size
10. Unambiguity: **model**	☑	

Figure 5.6: *Validation scenario of the model pattern*




```
@QueryBasedFeature pattern model(
  This : FunctionalElement,
  Target : FunctionalArchitectureModel) = {
  find parent+(This, Parent);
  find rootElements(Target, Parent);
} or {
  find rootElements(Target, This); }
```

Figure 5.7: *Definition of the model pattern*

This result represents a spurious counter example, because Functions used only with the context of a `FunctionalArchitectureModel`. For this purpose a partial snapshot is defined with a `FunctionalArchitectureModel` object to prune the search space and avoid such counter examples (Figure 5.8). However, its revalidation (Step 9) ends in a `Timeout` (more than 2 minutes) and thus this feature can only be validated on a concrete bounded domain of a maximum of 5 model objects in Step 9.

Finally in Step 10, the unambiguity of the `model` DF is validated without a problem.

5.5 Constraint Check

In our running example, a design rule captures that a FunctionalData EObject with a FAMterminator cannot also be connected to an InformationLink. It is specified by the terminatorandInformationLink query (see in Figure 5.9) that has two OR pattern bodies, one for the FunctionalInputs and one for the FunctionalOutputs with their corresponding incomingLinks and outgoingLinks, respectively. This rule is visible in the top part of Figure 5.9.

To demonstrate our tool another WF constraint is added to the DSL specification expressed by the `informationAndTermintator` query (Figure 5.9, bottom part), which prohibits that an `InformationLink` is connected to a `FAMTerminator`. This constraint only differs from the first body of the original WF constraint that it uses the inverse edges and thus it is a redundant.

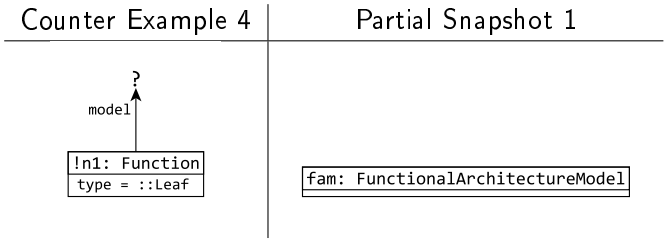


Figure 5.8: *Counter Examlpe and Partial Snapshot of the model validation*

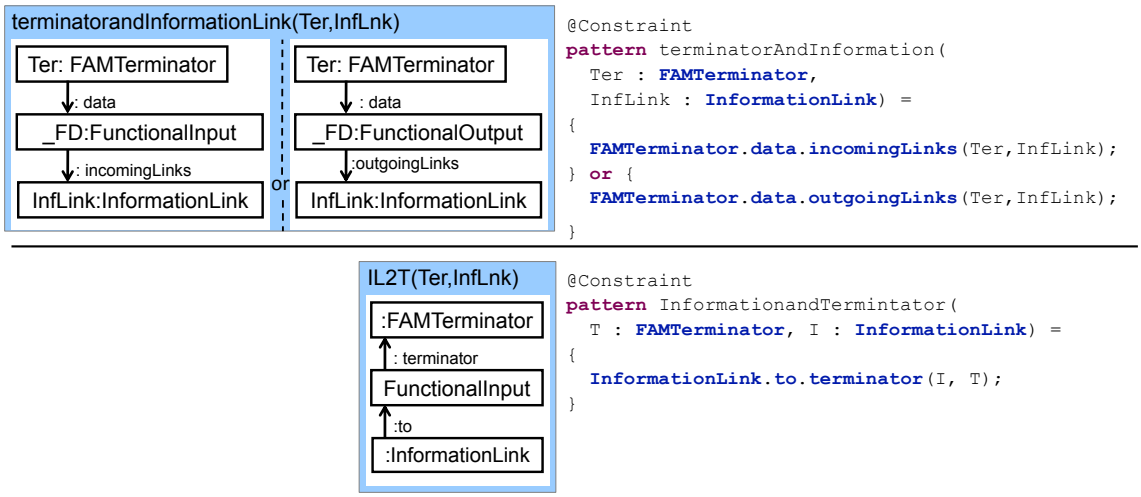


Figure 5.9: *Definition of the terminatorAndInformationLink pattern (top) and informationAndTermintator pattern (bottom)*

Validation step	Outcome	Action
11. Consistency: **T&IL**	☑	
12. Consistency: **IL2T**	☑	
13. Subsumability: **IL2T**	☒	Remove WF: **IL2T**

Figure 5.10: *Validation scenarios for well-formedness constraints*

The validation process of the WF constraints illustrated on Figure 5.10. At first, the consistency validation of the WF constraint terminatorandInformationLink (Step 11) is executed with a success. After this, the redundant informationAndTermintator is added to the system which remains consistent (Step 12). Finally the last constraint is checked for subsumption (Step 13) and found positive; thus it is already expressed by the DSL specification and thus it can be deleted from the set of WF constraints.

Chapter 6

Model Generation Case Study in Laser Guided Vehicle Domain

In this chapter the motivating scenarios of R3Cop project are demonstrated. The different cases and the proposed solutions are presented in three case studies. Each scenario contains the requirements, the reduced size instances models and the given solutions.

6.1 Model Generation Workflow

Our tool is able to generate instance models for a given DSL. It is possible to add extra predefining elements (like extra edges) to the DSL which are relevant only in the context of reasoning. Those predefining language elements are used to mark special relations in the partial snapshot where the semantic of the relation is defined by a constraint. For example in a domain about creating a layout for placed object a predefining relation called "`near`" would express a relation between the coordinates of two objects. The predefining constraint would look like this:

$$near(\mathtt{a}, \mathtt{b}) \Rightarrow \text{The difference between the positions of } \mathtt{a} \text{ and } \mathtt{b} \text{ is smaller than a limit}$$

So if there are two objects that are linked with a `near` reference their positions have to be filled in a way that satisfies the right side of the constraint.

There are many customisable elements in this process that allows the user to specify the requirements of the output model. that are illustrated with the general steps of the model generation workflow (visible in Figure 6.1).

1. The model generation takes a valid DSL as an input. Adding predefining elements to the DSL to define relations over the objects of a partial snapshot. Specification of the predefining constraints.
2. Constructing an initial model by a partial snapshot. The required feature can be denoted by the predefining references.
3. In the reification phase a concrete instance of the DSL will be generated that also satisfies the constraints from the predefining references.

4.A In case of unsatisfiable requirements the generation phase fails. The developer might reconsider the partial snapshot.

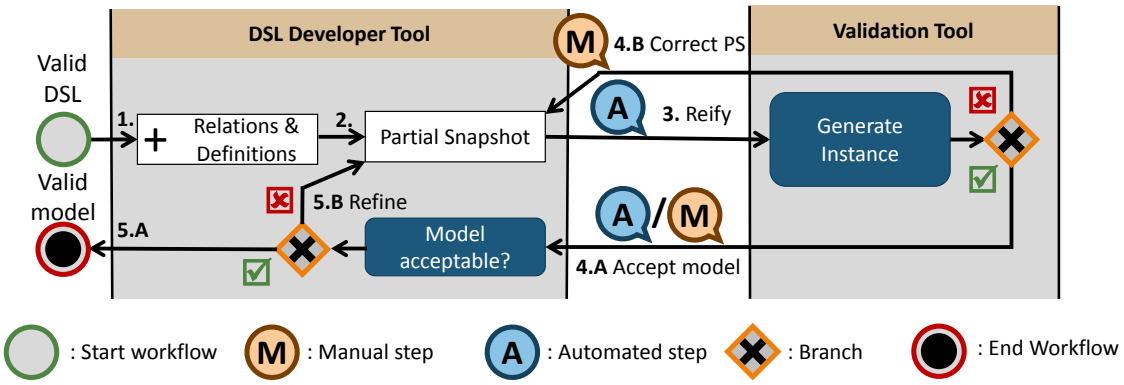


Figure 6.1: *Workflow of the Model Generation*

4.B If the model generation succeed the result will be submitted for acceptance. The acceptance process can be the review of the developer or a an automated process.

5.A If the result is a suitable model it will be the output of the model generation.

5.B if the result has failed to accomplish the acceptance it can be used in the refinement of the requirements.

6.2 Introduction

This section shows the basic scenarios used in the automated test-case generation task in the R3Cop project. In those cases the goal is to specify the values of unspecified attributes of objects in the abstract test properties. In order to specify the concrete test cases, extra references are added to the language to declare different relations between the model elements. Those relations are like "x is dangerously close to y" or "e event has happened before f".

The semantics of the extra references is defined by using OCL rules that restricts the values of the unspecified attributes. In the case study the required structure of the environment is specified by semantic of those relations and a concrete model is needed to be reified. Due to get better presentation output we sliced the problem in 3 loosely coupled scenarios, but in the case study all of the presented features are used together.

6.3 The Model

The presented metamodel is a reduced version, where only the important classes are shown. The metamodel models the environment of autonomous agent. In the world there are layouts, placed objects and events. A layout contains stations and segments which are connected to each other and compose the fixed infrastructure. A position object defines a concrete position given by an X and Y coordinates. The layout has a leftUp and a rightBottom references, which are used to reference two positions, and defines the size of the layout. The segments are parts of the layouts defined with their size. The placed object and the various inherited types of it represents the movable (dynamic objects) of the environment (e.g. pallets, boxes etc.). The moving objects are the human, truck, LGV and SUT, which are able to move alone. The LGV represents the different forklifts in the warehouse and the SUT is an LGV, which is the device under test. The stationary objects are fixed in the environment, e.g. a rack. They are placed on the stations which

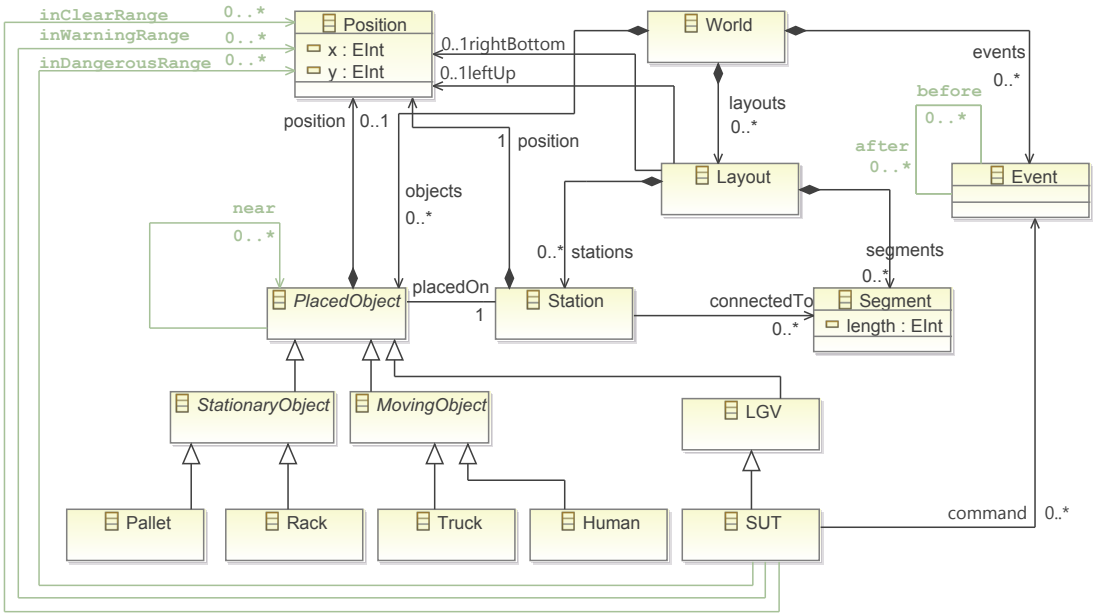


Figure 6.2: *Metamodel of the environment of the Elettric 80 case study*

is defined using 2D coordinates. Events can be defined which represents the dynamic changes of the environment during the time, e.g some movements or activities. Each event has a time stamp, which defines the order of execution.

Using the metamodel some constraints can be expressed (e.g. type or numeric constrains), but for complex requirements the metamodel in itself is insufficient. Due to the limitations of the metamodeling, OCL expressions are used to formulate these extension rules and these expressions are added to the extend metamodel.

6.4 Scenario 1: Events

6.4.1 Description

In this basic scenario the goal is to automatically define a concrete time stamp for all the events based on casual ordering. The logical ordering is defined by two additional references:

- `before`: The referred event has to happen before than this.
- `after`: The referred event has to happen later than this.

The absence of the references does not mean that the events can not be in that order. Naturally those references are opposites, so x `before` $y \Leftrightarrow y$ `after` x is structurally enforced by the EMF metamodel (using the inverse relations).

Concrete timestamps should be defined for the events based on the ordering relations. Only one event can happen at the same time, the time stamp of first one is 0. This can be formalised by those OCL constraints:

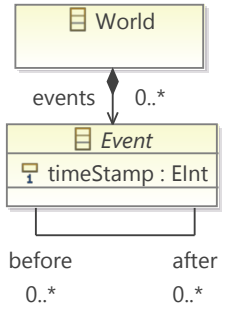


Figure 6.3: *Metamodel relevant to the Event scenario.*

I. If an event is after of an other event then the time stamp of this event should be greater than the time stamp of the other.

```
Event: self.after -> forAll(v: Event | v.timeStamp >self.timeStamp)
```

II. A time stamp exists which value is 0. It is necessary because the partial snapshot should contain more smaller separated graph and only one root of them can be 0.

```
World: Event.allInstances()->exists(e: Event | e.timeStamp=0)
```

III. The time stamp of the events are greater than or equal to 0.

```
World: Event.allInstances()->forAll(e: Event | e.timeStamp>=0)
```

IV. All events should have different timestamps.

```
World: Event.allInstances() ->
        forAll(e, v: Event |e.timeStamp = v.timeStamp implies e = v)
```

Partial Snapshot

The initial partial snapshot is illustrated using a graph. The ordering of events are specified, the attributes are not specified.

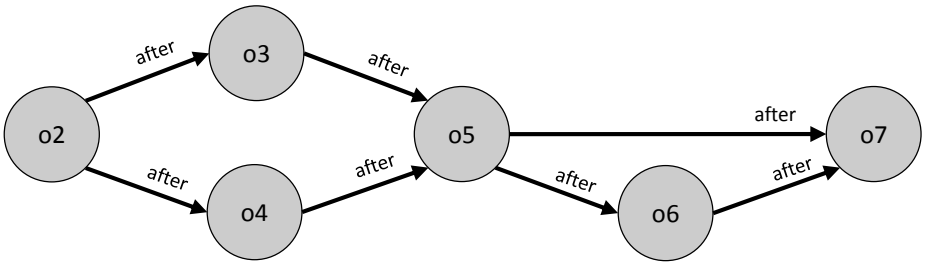


Figure 6.4: *The proposed event time stamp filling problem*

6.4.2 Completed Instance Model

The introduced problem is solved by the tool. The output completed instance model is shown in Figure 6.5. The tool is fill the attributes by using the OCL constraints.

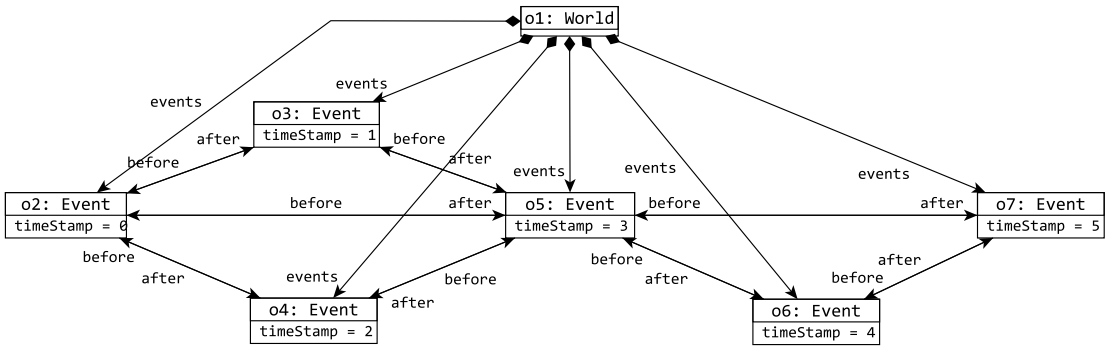


Figure 6.5: *The completed instance model*

6.5 Scenario 2: Layout

6.5.1 Description

In this world there are layouts which contain stations. The layout has a size which is defined by coordinates of the bottom right and the top left corners. The stations of the layout should be placed within this rectangular area, and these items should have different coordinates. The stations contain placed objects, which. placed objects should have the same coordinates as their stations. A placed object can have a near reference which means that the referenced objects are neighbours so difference of their or x or y coordinates should be less than a fixed constant.

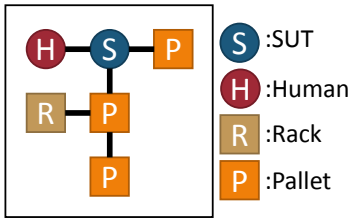


Figure 6.6: *Map of the demonstrated layout*

This can be formalised by those OCL constraints:

I. The position of the station is same as the position of its placed object.

```
World: Station.allInstances()->forAll(s: Station | s.placed<>null implies
  s.position.x = s.placed.position.x and
  s.position.y = s.placed.position.y)
```

II. All stations have different coordinates.

```
World: Station.allInstances()->forAll(e, v: Station |
  e.position.x = v.position.x and e.position.y = v.position.y implies
    e = v)
```

III. All stations must be placed within the area of the layout which contains then.

```
World: Layout.allInstances() -> forAll(l: Layout | l.stations ->
  forAll(s: Station | (l.leftUp <> null and l.rightBottom <> null)
    implies l.leftUp.x < s.position.x and
            l.leftUp.y > s.position.y and
            l.rightBottom.x > s.position.x and
            l.rightBottom.y < s.position.y))
```

IV. The near reference of the placed object means that the coordinates of referenced placed objects are neighbouring.

```
World: PlacedObject.allInstances() -> forAll(p: PlacedObject |
  p.near -> forAll(m: PlacedObject |
    (p.position <> null and m.position <> null) implies
      (p.position.x  =m.position.x and p.position.y+1=m.position.y) or
      (p.position.x-1=m.position.x and p.position.y  =m.position.y) or
      (p.position.x  =m.position.x and p.position.y-1=m.position.y) or
      (p.position.x+1=m.position.x and p.position.y  =m.position.y)))
```

6.5.2 Partial Snapshot

The initial partial snapshot is illustrated using a map in Figure 6.6. On the map different objects are placed: there are 3 pallets, a rack, a human and a SUT. The neighbours are linked if they must be neighbours so the lines symbolize the near references. In the initial partial snapshot the objects and the size of the layout are defined and the near reference are specified. The coordinates of the objects are unspecified.

6.5.3 Completed Instance Model

The completed instance model is shown in Figure 6.7. The positions of the placed objects and the stations are filled. In Figure 6.8 the positions of the object are illustrated on a map of the warehouse.

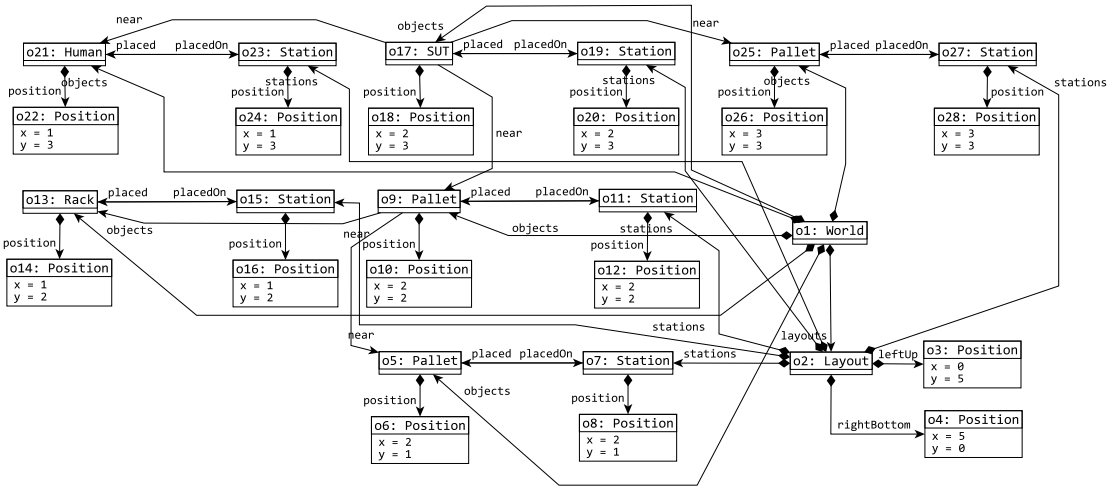


Figure 6.7: *The completed instance model*

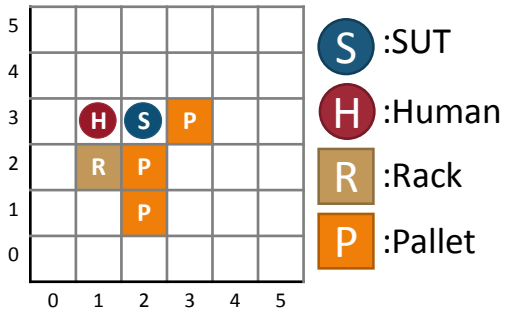


Figure 6.8: *Map of the solution*

6.6 Scenario 3: The Distance Zones

6.6.1 Description

In this scenario a SUT is created which represents the tested LGV. The SUT can have three predefining relations that are defined as follows:

I. inClearRange: the distance of two objects connected with this relation is minimum 5 meters

II. inWarningRange: the distance of two objects connected with this relation is minimum 2, maximum 5 meters

III. inDangerousRange: the distance of two objects connected with this relation is less than 2 meters

The usage of this relation is motivated by the case study defined in the R3Cop project, because the LGVs have to follow different behaviour and should produce different signals in different distance zones. In this scenario, three objects are created which are placed in the mentioned zones, all object in different zone.

The requirements can be formalised by those OCL constraints:

I. They are specified mentioned the constraints of the predefining relations.

```
SUT: self.inClearRange -> forAll(p: Position | self.position<>null implies
  (self.position.x-p.x) * (self.position.x-p.x) +
  (self.position.y-p.y) * (self.position.y-p.y) > 25)

SUT: self.inWarningRange->forAll(p: Position | self.position<>null implies
  (self.position.x - p.x) * (self.position.x - p.x) +
  (self.position.y - p.y) * (self.position.y - p.y) < 25 and
  (self.position.x - p.x) * (self.position.x - p.x) +
  (self.position.y - p.y) * (self.position.y - p.y) > 4)

SUT: self.inDangerousRange->forAll(p: Position| self.position<>null implies
  (self.position.x - p.x) * (self.position.x - p.x) +
  (self.position.y - p.y) * (self.position.y - p.y) < 4)
```

II. The stations must be placed on same coordinates as its placed object.

```
World: Station.allInstances() -> forAll(s: Station | s.placed<>null implies
 s.position.x = s.placed.position.x and s.position.y = s.placed.position.y)
```

III. All PlacedObject have different coordinates.

```
World: PlacedObject.allInstances() -> forAll(e, v: PlacedObject |
  (e.position <> null and v.position <> null) implies
  (e.position.x = v.position.x and e.position.y = v.position.y implies
    e=v))
```

IV. All stations are placed within the area of the layout.

```
World: Layout.allInstances()->forAll(l: Layout |
  l.stations->forAll(s: Station |
    (l.leftUp<>null and l.rightBottom<>null) implies
      l.leftUp.x      < s.position.x and l.leftUp.y      > s.position.y and
      l.rightBottom.x > s.position.x and l.rightBottom.y < s.position.y))
```

6.6.2 Partial Snapshot

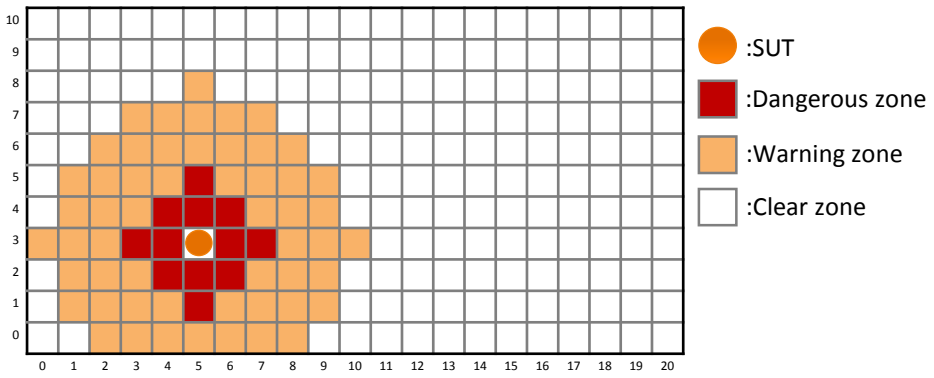


Figure 6.9: *Map of the layout with the defined zones*

The initial partial snapshot is illustrated by a map of the layout in Figure 6.9. The orange circle illustrates the SUT. The red zone is the dangerous the orange is the warning and the white is the clear zone. A Rack has to be placed in the dangerous zone, a LGV has to be placed in the warning zone and an other Rack has to be place in the clear zone. The connections of the instance objects are shown in Figure 6.10.

Figure 6.10: *The relationship of the defined objects*

6.6.3 Completed Instance Model

The diagram of the completed instance model is shown in Figure 6.11. The placed objects are placed, their attributes are filled. Figure 6.12 shows the map of the warehouse with the placed objects.

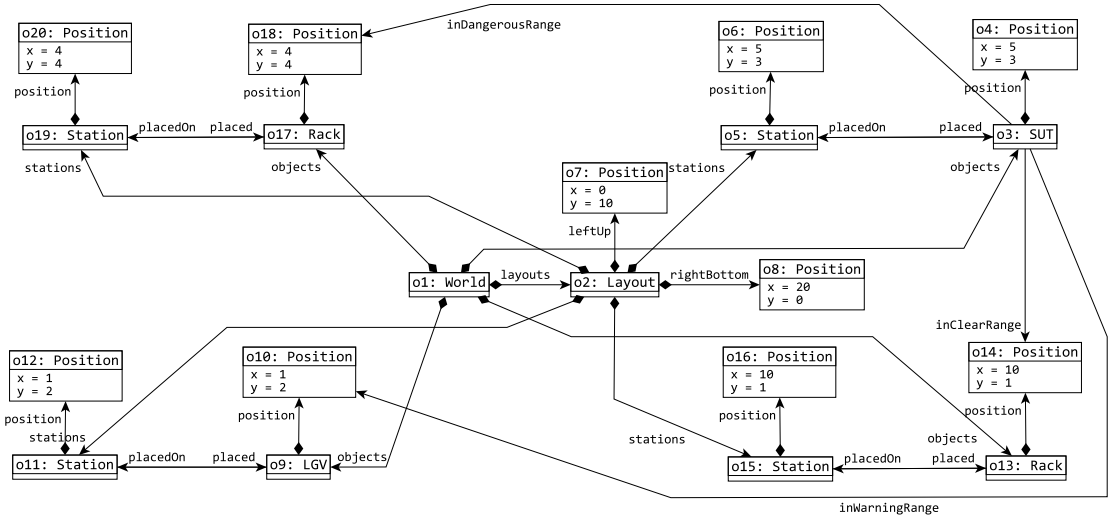


Figure 6.11: *Complete instance model*

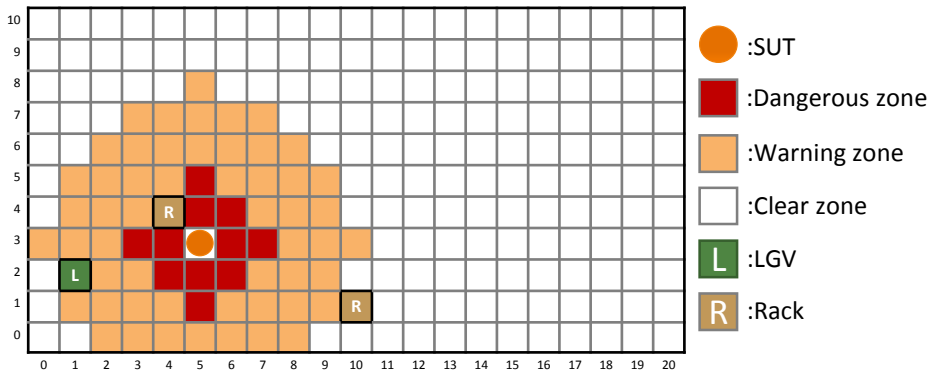


Figure 6.12: *Map of the solution*

Chapter 7

Mapping DSLs to FOL Formulae

This chapter discusses the transformation process of the DSL artifact to SMT language in details. The metamodel, instance model, OCL and GP are transformed to FOL formulae to be able to perform the reasoning tasks and to solve the different problems risen in the case studies.

7.1 Strategy of the Transformation

The current section introduces the main concepts used in our approach: structured transformation process enchanted with robust approximation methods.

7.1.1 Structure of the Transformation

The goal of the transformation of the DSL is to create an axiom system called DSL_F (where F note that this is a set of logic formulae), which is satisfiable only if the original DSL was consistent. If the DSL_F is satisfiable then by definition there is an interpretation M_F that satisfies DSL_F. Additionally, to back annotate the result defined by the M_F logic structures to an actual instance of the DSL, formally:

$$transformation(DSL) = DSL_F \text{ and } backannotated(M_F) = \text{M then}$$

1. $DSL_F \models M_F \quad \Leftrightarrow \quad$ M instanceof DSL
2. DSL_F does not have model $\quad \Leftrightarrow \quad$ DSL does not have instance

The *transformation* function consists of multiple different transformation steps that independently transforms the input modelling artifacts to the logic axiom system:

- The metamodel transformation creates the formulae called $META_F$ from the metamodel that maps the main structural features of the DSL (detailed in Section 7.2).
- The instance model transformation maps the formulae PS_F from the optional partial snapshot (discussed in Section 7.3).
- The OCL transformation generates the formulae called WF_F from the well-formedness constraints defined as OCL invariants (highlighted in Section 7.4).

- Finally, the Pattern transformation creates the formula set called GP_F from the definitions of the EMF-IncQuery graph patterns and link them to their corresponding ill-formedness (IF_F) or derived feature formulae (DF_F) (explained in Section 7.5).

So the transformed DSL is partitioned in the following way:

$$DSL_F = META_F \cup WF_F \cup GP_F \cup IF_F \cup DF_F \cup PS_F$$

The Search Parameters can directly customise the mapping process, their effect will be detailed in each transformation step. Our tool can execute different reasoning task on DSL_F. The Reasoning task transformation prepares DSL_F to create the actual input for our reasoning tool based on the method described in Section 7.6. The reasoning tool then makes the satisfiability check on the input formulae and provides the result, which will be interpreted in the context of the reasoning task.

If the reasoning tool finds the axiom system satisfiable an example interpretation will be created that explicitly defines every uninterpreted features of the axiom system (like how many objects in the model, which ones are linked with a reference or what are the matches of the graph patterns). By querying the metamodel specific attributes of this logic model an EMF instance model will be created.

7.1.2 Approximation techniques

The main advantage of the SMT solvers to the normal theorem provers is that the SMT solvers can use combinations of multiple elaborated background theorems, therefore it can effectively reason over a certain set of logic problems [25]. Our choice of background theorem was the effectively propositional logic (EPR)[42] as its provides logical formulae that can cover the largest set of DSL language features.

Definition 5 (Effectively propositional logic) *The effectively propositional logic is a fragment of the first order logic, which contains constant variables, relations and the statements in prenex form build from some existentially quantified variables, then some universally quantified variables, then the predicates and logical connectives.*

However, expressive power of the EMF-IncQuery or the OCL language is even larger than the SMT language itself. Some constraints such as recursively called patterns, transitive closures, set cardinalities and check expressions can not be fully compiled into it.

$$\text{EPR} < \text{SMT} < \text{EMF-IncQuery, OCL}$$

To tackles this problem and represent problems in the required logic fragment some approximation techniques have to be deployed:

Definition 6 (Approximations of Predicates) *The P^U predicate is underapproximate (P^O overapproximate) the P constraint if it satisfies the following implications for every parametrisation:*

$$P^U \Rightarrow P \quad (P \Rightarrow P^O)$$

As a trivial example the constant *true* predicate is always a good overapproximation, and $false$ approximates every predicate under. A statement also approximates itself.

An axiom system can be also approximated if every statement are approximated in it. The approximations of the formulae in DSL_F define languages with more or less instances than the unapproximated one, as the following implications show:

$$DSL_F^U \models M \Rightarrow DSL_F \models M \text{ and } DSL_F^O \not\models M \Rightarrow DSL_F \not\models M$$

This allows to validate properties of the DSL_F by proving the same properties on its under- or overapproximations.

$$DSL_F^U \text{ satisfiable } \Rightarrow DSL_F \text{ satisfiable and } DSL_F^O \text{ unsatisfiable } \Rightarrow DSL_F \text{ unsatisfiable}$$

This means that the consistency check of a domain specific language can be done by verifying a more general logical structure what more efficient to reason over.

Example 4 The objects of an EMF model are arranged in a tree hierarchy. This defines that the containment graph satisfies the following properties (as described in details in Section 7.2):

- Every object is contained by an other with the exception of the root element. (This is expressible in SMT but not in EPR.)
- The containment is acyclic. (This is not expressible in SMT.)

To express containment hierarchy in SMT the second rule have to be overapproximated like this:

- the containment graph is free from circles of maximum five length. (This is SMT and EPR.)

To express it containment hierarchy in EPR the first rule have to be omitted.

A consistency consistency check can be efficiently executed on a problem in EPR the class, and if the reasoning tool finds the DLS with more general containment rules unsatisfiable then the original problem have to be unsatisfiable too.

7.2 EMF metamodel transformation

Table 7.2 summarises the transformed features of the metamodel. It also presents which property is expressible in FOL or EPR.

7.2.1 Objects

The models of the EMF framework are graph based models, where the EObjects are the nodes and the EReferences are the edges. In Z3 models the type of EObjects is mapped to Object: `(declare-sort Object)`. If the number of objects is bounded then a fix-sized enum is declared.

Example 5 A model with exactly four elements can be defined in the following way:
`(declare-datatypes () ((Object element1 element2 element3 element4)))`

where `element1`, `element2`, `element3` and `element4` are the objects in a four element literals.

Features of the metamodel	
EClasses	E +
Class hierarchy	E +
EEnums	E +
EReferences	E +
EAttributes	E +
Multiplicity upper bound	E +
Multiplicity lower bound	E −
Inverse edges	E +
Containment hierarchy	A −

E: Expressible A: Approximable X: Inexpressible +: in EPR −: not in EPR

Table 7.1: *Expressing Ecore features in Z3*

7.2.2 Types

The possible types in the instance models are the classes of the metamodel. These classes are transformed to the type indicator predicate. If an "`o`" object is an instance of the type Station then the `isType!Station` expression is true, else it should be false.

Example 6 The declaration of the Station type:

```
(declare-fun isType!Station (Object) Bool)
```

7.2.3 Type hierarchy

The Z3 does not support inheritance between the types so the class hierarchy has to be defined in an other way. The simple way to define the type hierarchy of the EMF is to enumerate the possible type cases of the type predicate combination. This is defined by a table, where the columns represent types and the rows the concrete types(not abstract, not interface). The cell represents a predicate, which is positive if the type of the row is compatible with the type of column, and negated if it is not.

Example 7 The transformation of the previously presented metamodel is shown in the next table. The PlacedObject is an abstract class and the inherited types can be the Rack and the Station, where the Rack is an PlacedObject:

Figure 7.1: *Type hierarchy example*

```
(assert (forall ((o Object))( or
```

		PlacedObject	Rack	Station	
Rack	`(and`	`(isType!PlacedObject o)`	`(isType!Rack o)`	`(not (isType!Station o))`	`)`
Station	`(and`	`(not(isType!PlacedObject o))`	`(not (isType!Rack o))`	`(isType!Station o)`	`)`

```
)))
```

7.2.4 Reference

The references of the metamodels are the edges between the objects. Those edges are directed and also loop edges are allowed. The references are transformed to relations. The types of the objects on the end of the relations has to be defined. The definition of the relation is an assertion: if the `(o, t)` pair satisfies the relation then the "`o`" is the instance of the source of the relation and "`t`"is the instance of the target.

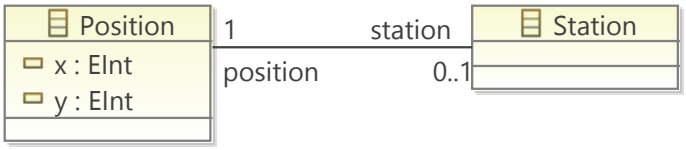


Figure 7.2: *Reference example*

Example 8 The definition of the position reference of the Station:

```
(declare-fun Station!position (Object Object) Bool)
```

The limitations of the ends of the edge:

```
(assert (forall ((o Object) (t Object)) (=> (Station!position o t) (and
(isType!Station o) (isType!Position t)))))
```

7.2.5 Multiplicity

A relation with 0..* multiplicity is the default, In other cases multiplicity assertions might be necessary. In the n..m relation the n lower bound means that every object is in relation with n different one. The upper bound m means that there is not at most m different target elements that is in relation with the object. The lower bound which value is different from 0, is transformed into existential quantifier, which is surrounded by an universal quantifier which provide the type of the source object.

Example 9 The transformation of position edge of the Station, which lower bound is 1:

```
(assert (forall ((src Object)) (=> (isType!Station src) (exists ((trg0 Object))
(Station!position src trg0)))))
```

Example 10 If the upper bound is different from infinite, it is transformed to an universal quantifier. This example presents when the upper bound is 1:

```
(assert (forall ((src Object) (trg0 Object) (trg1 Object)) (=> (and
(Station!position src trg0) (Station!position src trg1)) (= trg0 trg1))))
```

7.2.6 Inverse edges

The inverse of a reference can also be defined as demonstrated in the following example.

Example 11 In the previous example the station and the position edges are inverse:

```
(assert (forall ((o Object) (t Object)) (=> (Station!position o t)
(Position!station t o))))}
(assert (forall ((o Object) (t Object)) (=> (Position!station o t)
(Station!position t o)))
```

7.2.7 Containment

The objects of an EMF model are arranged in a directed tree hierarchy by the containment edges. The acyclicity means that any object is unreachable from itself by the path of the containment edges. If the previous statement is satisfied, the composition graph is DAG. There should be a root element which is represented by the root constant in Z3. Every object of the model has exactly one parent with the only exception of the root element.

Example 12 There is containment relation between two object if there is a containment type edge between them:

```
(declare-fun root () Object)
(declare-fun contains (Object Object) Bool)
```

The roots does not have any parent, and every non-root element have exactly one:

```
(assert (forall ((parent Object)) (not (contains parent root))))
(assert (forall ((o Object)) (or (= o root) (exists ((parent Object)) (and (not
(= parent o)) (contains  parent o))))))
```

The acyclicity of the containment hierarchy is inexpressible in the SMT language so some kind of approximation is needed. For example statement of "the containment graph of the is free from C_3 (three length circle)" overapproximate the acyclicity requirement. Increasing the size of the forbidden circles converges to the acyclicity, and we can deal any kind of containment inconsistency using an appropriate approximation level.

Example 13 The following SMT assertion detect the three length circles.

```
(assert (forall (
  (circleElement1 Object)
  (circleElement2 Object)
  (circleElement3 Object)) (not (and
    (contains circleElement1 circleElement2)
    (contains circleElement2 circleElement3)
    (contains circleElement3 circleElement1)))))
```

7.2.8 Attributes

The attributes of the metamodel are the properties of the classes. In this status of the work, only attributes of type EInt are supported. The EMF EInt attribute is mapped to the Z3 Int type. The attributes are transformed to relations which return the value which belong to the input object.

Features of the metamodel	
Instance Objects	E +
Types	E +
Abstract Types	E +
Filled References	E +
Filled Attributes	E +

E: Expressible A: Approximable X: Inexpressible +: in EPR −: not in EPR

Table 7.2: *Expressing Ecore features in Z3*

Example 14 The declaration of the x coordinate attribute of the position class is the following:

```
(declare-fun Position!x (Object) Int)
```

If the type of object is not position then the `Position!x` relation must return with 0:

```
(assert (forall ((o Object)) (=> (not (isType!Position o))(= (Position!x o)
0))))
```

7.3 EMF instance model transformation

This section defines how instance models and partial snapshots are transformed. Table 7.2 summarises the transformed features.

The analysis can be parametrized by an initial instance model of the metamodel. This initial model can be inserted to the axiom system of the input of the Z3.

7.3.1 Instance object

The instance objects are the instances of the classes of the metamodel. They are transformed to a function, which return with an Object. The transformed instance object must be different.

Example 15 Definition of the "o1" instance object is:

```
(declare-fun InstanceObject!o1 () Object)
```

Three instance objects are defined and they must be different:

```
(assert (distinct InstanceObject!o1 InstanceObject!o2 InstanceObject!o3))
```

7.3.2 Type

The type of the instances must also be specified.

Example 16 The type of `InstanceObject!o1` is Station:

```
(assert (isType!Station InstanceObject!o1))
```

Features of the OCL	
self	E +
allInstances	E +
Iterator expressions	E −
Attributes	E +
References	E +
Condition	E −
notEmpty	E −
isEmpty	E +
Logical operations	E +
Mathematical operations	E +
oclIsTypeOf	E +
Transitive closure	A +
let	X
Ordered set	X

E: Expressible A: Approximable X: Inexpressible +: in EPR −: not in EPR

Table 7.3: *Expressing OCL features in Z3*

7.3.3 Reference

The references should be defined in the following way: the reference type, the target object and the source object must be defined.

Example 17 Between the `InstanceObject!o1` and the `InstanceObject!o2` there is an position reference:

```
(assert (Station!position InstanceObject!o1 InstanceObject!o2))
```

7.3.4 Attributes

If the attribute is filled then the value of attribute has to transform and set.

Example 18 The x coordinates of `InstanceObject!o2` is 3:

```
(assert (= (Position!x InstanceObject!o2) 3))
```

7.4 OCL constraint transformation

In this section the OCL mapping is demonstrated. In the project we deal with the OCL invariants. We formulate constraints which should be satisfied and that way they define attribute values or relations of attributes. The nature of these constraints is exploited and with them we can fill out these attributes or we show the contradiction between model and attributes.

During the mapping, at first we have to select a subset of the OCL elements which we want to deal with. The transformed OCL expressions have to be selected because the number of elements are huge. The selected elements give a wide range of practical languages. Those elements are chosen which are very common or facilitates the construction of rules. For example: iterators, attributes, references, boolean or mathematical expressions.

OCL rules represented as their AST can be visited by the AST traversal algorithm. Generally, the mapping can be solved recursively, but there are some unusual cases which should treated as a special case. Always with the AST recursively does not give the correct results, sometimes the expression semantic is not equal to the technical construct. A good example is the `reference=null` expression. If we resolve it recursively, then the two sides of the equal sign will be comparable expressions, but in the first order logic we can't treat the null expression properly. These type of phrase should be treated as exception and the mapping must eliminate the usage of null expression.

The list of the supported language elements is visible in Table 7.3.

7.4.1 Mapping

In this part the most important theoretical considerations of the mapping are presented. The OCL mapping is greatly influenced by the mappings of metamodel and instance model, because the modeling dependencies between them. Therefore, constraints are transformed to SMT assertion.

Sets

The constraints usually are evaluated on a set. In this case we have the following two options: One of them is when the `Class.allInstances()` formula is used. In this case the mentioned formula is transformed to a predicate, which reflects which type of instances have to be true the OCL expression.

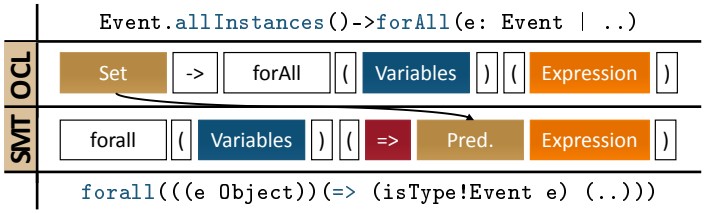


Figure 7.3: *Transformation of the expression which contains a set*

The second of them is when we define the context and in the constraint refer to it with the self element. In this case an extra forall expression should be added to the expression which contains the translated rule. The variable of this forall is the `self` and the forall expression contains a predicate which formulate type constraint.

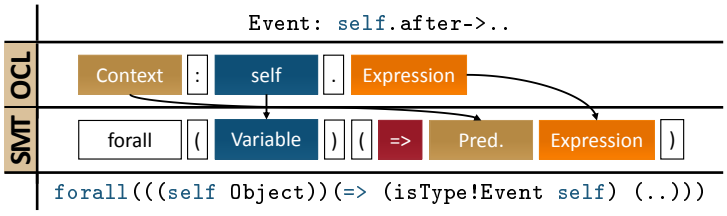


Figure 7.4: *Transformation of the expression which contains "self" variable*

Iterator expressions

The second group which is presented is the iterator expression, for example: forall, select and exists. Except the exists, the other iterators are equivalent of the universal quantifier in the

first order logic, which is equivalent of the forall SMT expression. The exists is equivalent of existential quantifier. The example shows that the variables and the expressions of iterator can be formalized using SMT variables and expressions.

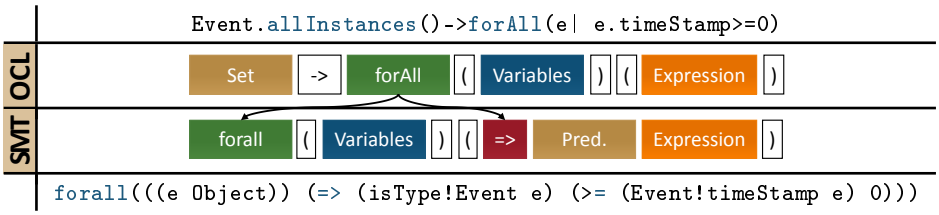


Figure 7.5: *Transformation of the forall*

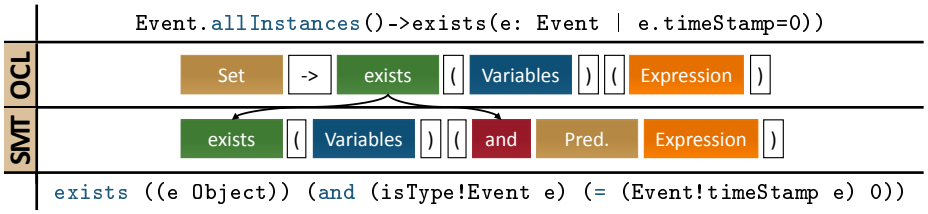
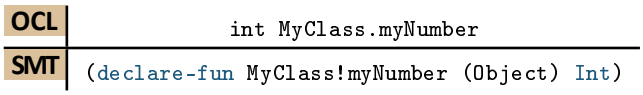


Figure 7.6: *Transformation of the exists*

Attributes and References

The mapping of the attributes and references are not trivial. The attribute should be translated to a function.



Transformation of references is complex. We distinguish based on the multiplicity: 0..1, 1 or infinity multiplicity. We distinguished based on the number of references: single or multiple path. These type of references are different. Now the transformation of types are introduced.

At first references are introduced infinite multiplicity. This type of reference or in multiple path the last of this type of reference imply an iterator expression, so in many cases the iterator expressions have to be completed. The single path which multiplicity is infinite is transformed to a function, which is a predicate. This predicate is contained by an iterator expression which follows the reference.

The multiple path with infinite multiplicity is transformed to a special structure. Except the last reference, all references attracted to a forall and their expressions contain a function which maps the references.

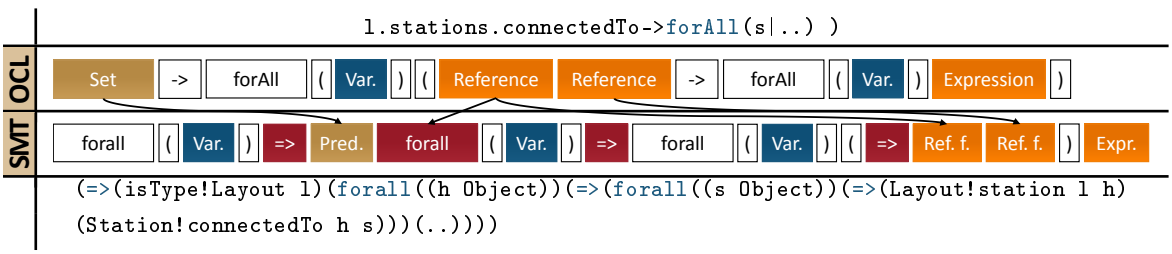


Figure 7.7: *Transformation of the expression which contains infinity multiplicity, multiple path*

The 0..1 or 1 multiplicity references rarely stand alone, usually they are followed by an attribute. The single path which multiplicity is 1 is transformed to an exists expression which contains a predicate which is transformed to the reference. If it is followed by an attribute then we have 2 functions: one of them symbolizes the attribute and the other of them symbolizes the references. In this case helper variables are had to define. This variables give the parameters of the functions. The multiple path which multiplicity is 1 is transformed same as the previous case, but more variables and functions which is symbolized the references are defined.

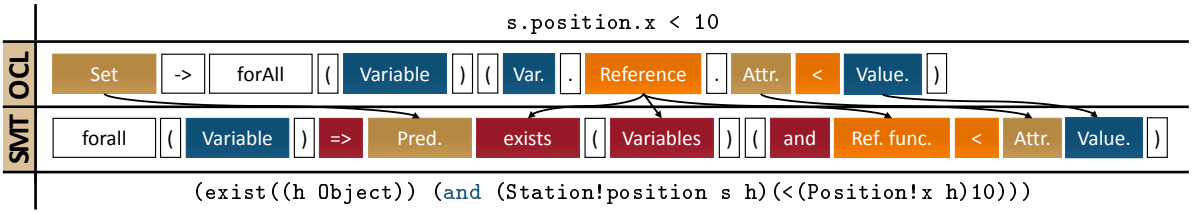


Figure 7.8: *Transformation of the expression which contains 1 multiplicity, single path*

The single navigation with multiplicity 0..1 is transformed to an implication. The left side of it is an exists quantifier construct which expression says that the reference is not null. It is translated to an expression which states that a variable exists, which satisfies the function which is transformed from the reference. The right side is an exists which contains the predicate which is transformed from the reference and the other elements of the expression. Usually a reference is followed by an attribute. In this case helper variables are defined which are contained by the exists expression. The right side of the implication is same as the mapping of the references which multiplicity is 1. The multiple navigation which multiplicity is 0..1 is same as the previous case.

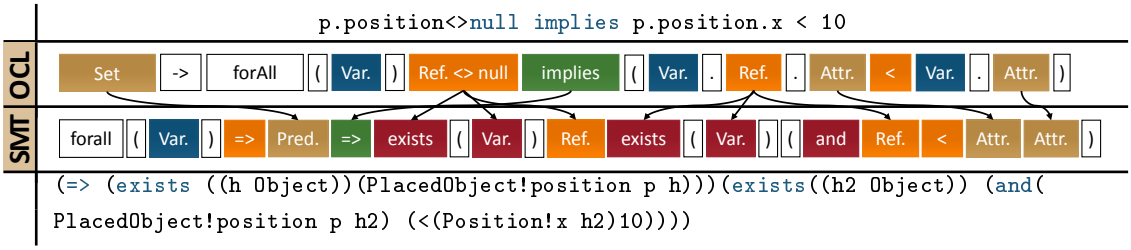


Figure 7.9: *0..1 multiplicity, single path transformation*

Condition

The structure of the OCL condition is "`if`" expression "`then`" expression "`else`" expression "`endif`". It is transformed to an "`if`" expression which exists in the SMT. The terms between the keywords are unlocked by using the other cases. From the condition expression element is not missed.

Built-in Functions

In the OCL, there are many built-in functions, which facilitate the constraints formulation. Some of them are transformed. One of them is the `oclIsTypeOf()`. It can be used in this form: `variable.oclIsTypeOf(MyType)`. It returns true, if the type of variable is `MyType`. It is translated to a function, which returns true if the type of the received parameter is equals with the type which is referenced with the function.

OCL	`variable.oclIsTypeOf(MyType)`
SMT	`(isType!MyType variable)`

The `notEmpty()` function is called on a set or collection which is available through references. It returns true if the set or collection is not empty. This is transformed to an exists expression which means that an object exists which is placed in the given set.

OCL	`variable.reference->notEmpty()`
SMT	`exists((t Object)) (reference v t)`

The `isEmpty()` using is same as the `notEmpty()`, but the meaning of it is different. It returns true if the set or collection which is called is empty. It is transformed to a "not exists" expression. An exists is created which means that exists an object which is in the set. The created exists is denied, and the meaning of it equals with the meaning of `isEmpty`.

OCL	`variable.reference->isEmpty()`
SMT	`not(exists((t Object)) (reference v t)`

Logical and mathematical operations

The logical and mathematical operations have equivalent in the first order logic. For example the + is transformed to +, the implies is transformed to $\Longrightarrow$ and the `and` is transformed to `and`. Except `not`, all operations have a left and a right side. Both sides contain an expression. These expression are resolved by using the other cases which deal with the particular type.

Comparison

The comparison is happened with the "=" and "<>" signals. If boolean formulas and numbers are compared then the transformation is same as described in the previous paragraph. There are some special cases, one of them is the `v.reference=null` expression. It is transformed to an forall expression which means that there is not two objects which is available with the reference.

OCL	`v.reference=null`
SMT	`forall ((null Object)) (not (Reference v null))`

The `v.reference<>null` is a special case. It is transformed to an exists expression which means that exists an object which is available through the v.reference.

OCL	`v.reference<>null`
SMT	`exists((null Object)) (Reference v null)`

The `v.reference=variable` transformation is same as the previous case.

OCL	`v.reference=variable`
SMT	`exists((variable Object)) (Reference v variable)`

DF	Features of model query	WF
E +	Classifier constraint	E +
E −	EReference constraint	E +
E −	Acyclic pattern call	E +
E −	Negative pattern call	E −
A −	Transitive closure	A +
A −	(Positive) pattern call recursion	A +
A −	Arbitrary call graph	A −
X	Aggregate (eg. Count, Sum)	X
X	Check expressions	X

E: Expressible A: Approximable X: Inexpressible +: in EPR −: not in EPR

Table 7.4: *Expressing Ecore and* EMF-IncQuery *language features in Z3*

7.5 EMF-IncQuery Graph Pattern Transformation

This section describes how EMF-IncQuery patterns can be transformed to first order formulae. Table 7.4 shows which feature can be translated to SMT and EPR whether they used as well-formedness constraints or derived features.

7.5.1 Structure of the Patterns

An IncQuery pattern consists of a parameter list and a definition that specify a condition over the parameters. The parameter list is a fix sized vector of variables over the model, let us denote it as *Params*. The condition is defined by pattern bodies that consist of constraints.

The match-set of a pattern is a relation which is explicitly transformed to SMT relations. The satisfaction of the relation is specified by the pattern definition that express a $pattern(Params)$ condition over the parameters.

$$Params \in patternMatch \Leftrightarrow pattern(Params)$$

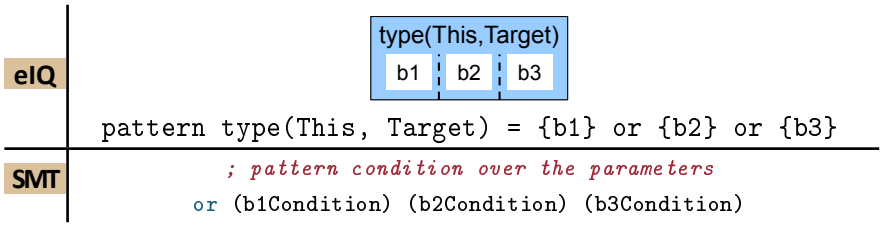
Example 19 Let us take a two parameter pattern called type with the parameter list This: Function and Target: FunctionType. The matches of this pattern are defined by the following predicate:

```
(declare-fun pattern!type (Object enumType!FunctionType) Bool)
```

And the condition that defines the relation:

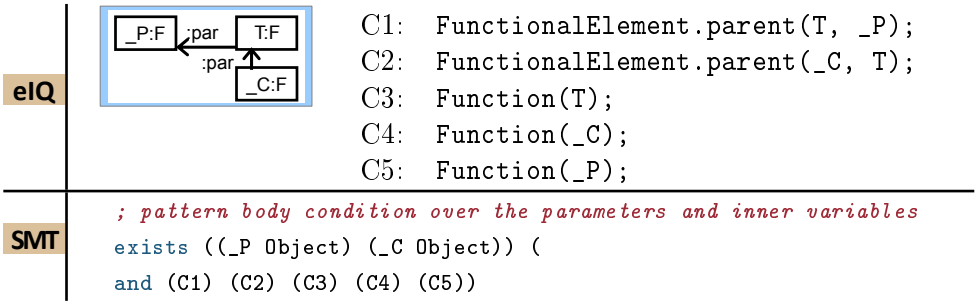
```
(assert (forall (
  (parameter!This Object)
  (parameter!Target enumType!FunctionType)) (iff
    (pattern!type parameter!This parameter!Target)
    (
    ; pattern condition over the parameters
    ))))
```

An individum vector is element of the match-set if and only if the vector satisfies one of the pattern body. So The pattern condition is defined as the disjunction of the pattern body conditions.



Transformation of the pattern condition

The pattern body condition is defined by the constraints of the body, where the condition is the conjunction of the constraints. A patten body may introduce additional existentially quantified inner variables called *Vars*. For example the following body of the type pattern contains two path and three classifier constraints:



Transformation of the pattern body

So the pattern condition is structured as follows:

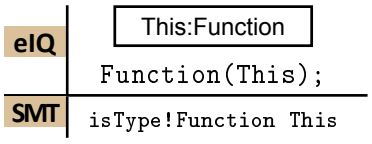
$$pattern(Params) = \bigvee_{\substack{body \in \\ pattern.\texttt{bodies}}} \exists Vars \bigwedge_{\substack{constraint \in \\ body.\texttt{constraints}}} constraint(Params, Vars)$$

The following section defines the transformation method for each supported the constraint.

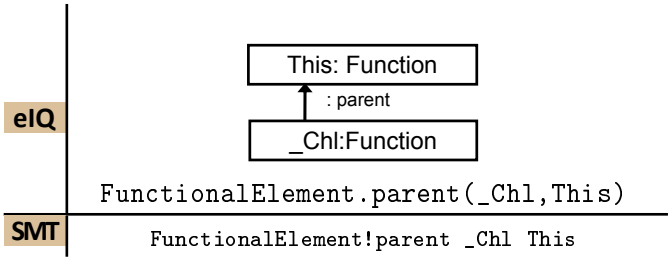
7.5.2 Constraint Transformation

This section provides the translation of the simple constrains of the IncQuery language to a Z3 expression.

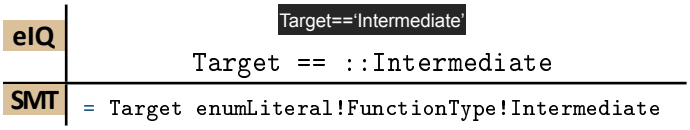
Classifier constraint defines the type of the objects that are binded to the variable. The EMF-IncQuery constraint can be easily compiled to type predicate. This can be transformed to the satisfaction of an `isType!xxx` predicate:



Path constrains in IncQuery defines that there is a path consists of sequence of references from the defined type that leads from a variable to another. By introducing the implicit object variables as the inner nodes of the path, the expression can be compiled into simple reference requirements. For example The path expression constraint `FunctionalElement.parent(_Chl,This)` defines that there is a path that starts from `_Chl`, ends in the `_This` object. If The path touches some further object that should be referred by existentially quantified implicit inner variables.



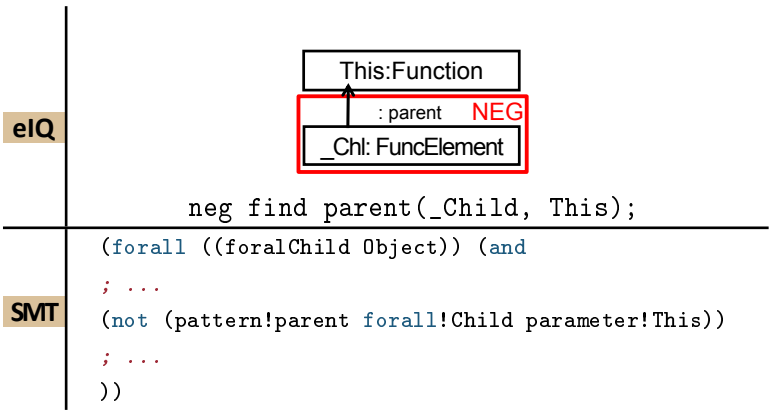
Equivalence and unequivalence of two individual can be simply defined as with SMT equivalence relation:



Pattern Call Constraints The pattern call constraints makes it possible to compose more complex patterns that referring to others.

- A positive call defines that the substituted parameters have to create a match of the referred pattern.
- Negative calls may introduce new negatively referenced variables. A negative pattern call defines that the target pattern does not have match for the substituted old variables with for any possible substitution of the negatively referenced parameters.

For example there is a negative pattern call from the `type` pattern:



Transitive closure approximation is an advanced lanugage element of the EMF-IncQuery pattern language. The transitive closure of a two-parametrezed pattern matches on the e_1 e_n pair if there is a $e_1, e_2, \ldots e_n$ sequence of model elements where the pattern is matches every e_i e_{i+1} pair. The transitive closure of a pattern can only be approximated in first order logic. The detailed process of the approximation is available in [41].

For example, predicate $parent(This, P) \Rightarrow parent2Match(This, P)$ defines an overapproximation of length 2 for the transitive closure of the `parent` EReference in the second pattern body of the `model` query, in the following way:

2: $parent2Match(This, P) \Rightarrow parent(This, P) \vee \exists m1 : parent(This, m1) \wedge parent1Match(m1, P, This)$
1: $parent1Match(This, P, d1) \Rightarrow parent(This, P) \vee \exists m2(m2 \neq d1) : parent(This, m2) \wedge parent0Match(m2, P, d1, This)$
0: $parent0Match(This, P, d1, d2) \Rightarrow parent(This, P) \vee \exists m3(m3 \neq d1, m3 \neq d2) : parent(This, m3) \wedge \mathbf{true}$

7.5.3 Patterns as DSL elements

Model query patterns are used to specify the restrictions on the structure of the DSL. The patterns defined as constraints and derived features are transformed in the following way:

- Ill-formedness constraints are defined as a statement that the model is free from matches of this pattern. For example in case of the pattern `terminatorAndInformation` the statement looks like this:

eIQ
```
@Constraint pattern terminatorAndInformation(T, I)
```
SMT
```
(assert (forall ((T Object) (I Object)) (
(not (pattern!terminatorAndInformation T I))))
```

- Derived features states that the features evaluate exactly when the specifying pattern matches the class and the value. The transformed DF `type` pattern looks like this:

eIQ
```
@QueryBasedFeature pattern type(This, Target)
```
SMT
```
(assert (forall ((This Object) (Target enumType!FunctionType)) (iff
(Function!type This Target)
(pattern!type This Target))))
```

7.6 Transformation of the Reasoning Task

This section describes the way how the result formulae are modified to express the different validation problems. Generally, the main goal is execute the proving of theorem T over the axiom system of DSL_F. Formally:

$$DSL_F \models T$$

To prove this property the consistency of $DSL_F \cup \neg T$ is checked:

$$
\begin{array}{rcl}
DSL_F \cup \neg T \text{ unsatisfiable} & \rightarrow & DSL_F \models T \\
\text{exists a model } M\text{: } DSL_F \cup \{\neg T\} \models M & \rightarrow & DSL_F \not\models T, \text{ and } M \text{ is a counterexample}
\end{array}
$$

- Subsumability check: T states that the target constraint is satisfied.
- Completeness check: T states that every occurrence of the derived feature has al least one value. For example if the completeness of the model reference of the Function class is checked then $\neg T$ is the following assertion:

```
(assert (exists ((incomplete Object)) (forall ((target Object))
  (and (not (Function!model incomplete target))
            (isType!Function incomplete)))))
```

- Ambiguity check: T states that every occurrence of the derived feature has at most one value. In case of unambiguity of the model reference of the Function $\neg T$ looks like this:

```
(assert (exists ((ambiguous Object)
                 (target1 Object)(target2 Object)) (
  (and (Function!model ambiguous target1)
       (Function!model ambiguous target2)))))
```

Chapter 8

Implementation

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

8.1 Architecture

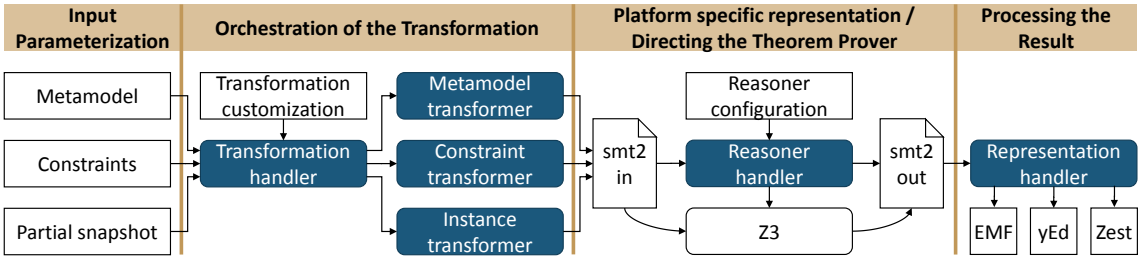


Figure 8.1: *Architecture of the tool*

The architecture of the tool is presented in Figure 8.1. The architecture and the processing of the statements are divided four section. First the input parametrization is introduced that:

I. The metamodel is given as eCore model by DSL specification.

II. The constraints add extra rules to the problem, they can be OCL or GP constraints.

III. The partial snapshot is a special instance model which is more permissive than the EMF instance model, it permits:

   i. unfilled attributes

   ii. abstract objects

   iii. unconnected partitions

   iv. missing or extra edges

   v. new objects

The partial snapshot is created by the reflective editor of the EMF instance model, which is shown in Figure 8.2.

The tool is executed using the defined parametrization. The transformation handler gets the input, which calls the transformation components: first the metamodel transformer, then the instance transformer finally the constraint transformer. If the transformation is successful then the SMT file is generated which contain the collected outputs of the previous components which is shown in Figure 8.3.

The SMT generated input file is passed to the reasoner handler. The reasoner handler is parametrized using a reasoner customization which defines the parametrization of the solver. This component calls the Z3, and after successfully execution its output is appended to the input file. The reasoner handler parses this output and based on the results builds the completed partial snapshot.

The result is a valid partial snapshot in the case, when the output of the Z3 is not a counterexample. The result is a counterexample in the following cases: (i.) inconsistency of the instance model or (ii.) inconsistency of the constraints or (iii.) one of the constraints can not be satisfied. The results can represented by different visualization tool. The representation handler represents the result, if the instance model is valid it can visualize the result in EMF, yED [59], Zest [53], in other cases the yEd or the Zest can be used.

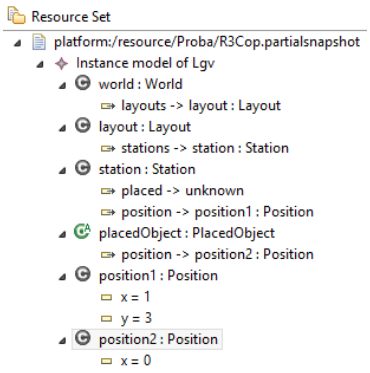


Figure 8.2: *The created editor of the partial snapshot*

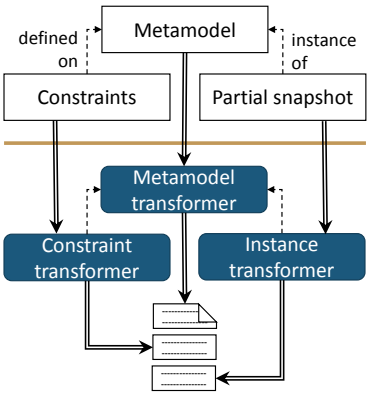


Figure 8.3: *The production of input SMT code*

8.1.1 Details

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

Grammar

SMT code can be constructed and passed easily, using the constructed SMT grammar. Xtext [52] is used. The language is defined by a generative grammar, which means a list of applicable rules that are produce all elements of the language. The SMT grammar is based on and extends the FOL grammar (defined in Chapter 3). Using the defined grammar the input and output SMT files can be parsed easily. During the transformation the defined symbols are used, text is not generated so the produced SMT input is a graph of defined elements.

Transformation

The instance and metamodel transformation is implemented using the Xtend [51] Eclipse plugin. The Xtend is a statically-typed programming language which is translated to Java source code. Syntactically and semantically the root of Xtend is the Java language but is extended by extra features, e.g. lamda expressions, powerful switch expressions and properties. With Xtend, the model elements can be visited easily, the writing of different filters is easier when using the lambda expressions. The code is more readable and less complex.

OCL transformation

The OCL rules and their contexts are given as a string. They must be read, parsed and finally their AST is built [24]. 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 (i.) parses the string, (ii.) connects the elements of the constraints to the metamodel, (iii.) defines the type of the OCL elements, (iv.) creates the AST and (v.) checks the syntax of the expression. The built AST has to be visited recursively the EcoreSwich class is used. It is a switch which can manage the inheritance hierarchy of the model. Every OCL element is associated with a method that can be overridden by the code of mapping. The doSwitch can be called recursively so the expression could be translated by it and the result is the SMT code of expression.

Resolution of the SMT Output

The interpretation of the generated SMT output is difficult because the structure of the output code is complex. The Z3 produces function compositions, so the `and` connection of the generated functions is given. Every function can contain other function definitions, branches, logical or numeric values and objects. These structure has to be resolved recursively and the solutions should be evaluated. The functions are resolved and the new partial snapshot is built during the resolving.

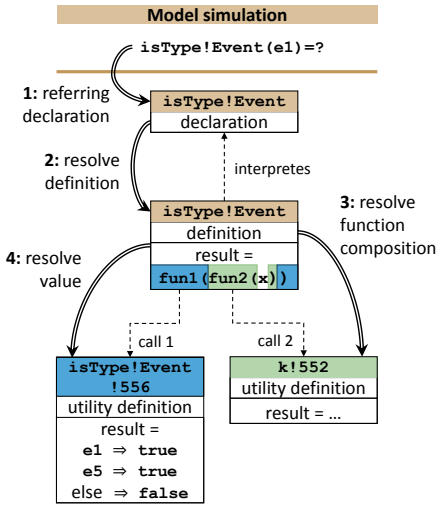


Figure 8.4: *Example of the structure of the SMT output*

Example 20 In Figure 8.4 an example is shown. The value of the `isType!Event(e1)` is searched. The steps of the resolvation are:

1. The referring declaration of the function has to be searched.
2. The declaration has a definition which have to be searched and resolved.
3. The definition contains a function composition which has to resolved. In the example function `k!552` has to be resolved first.
4. With the result of the `k!552` function must be parametrized the `isTypeEvent!556` function and it has to resolve the value, which returns with the value of the `isType!Event(e1)`. It can be true or false.

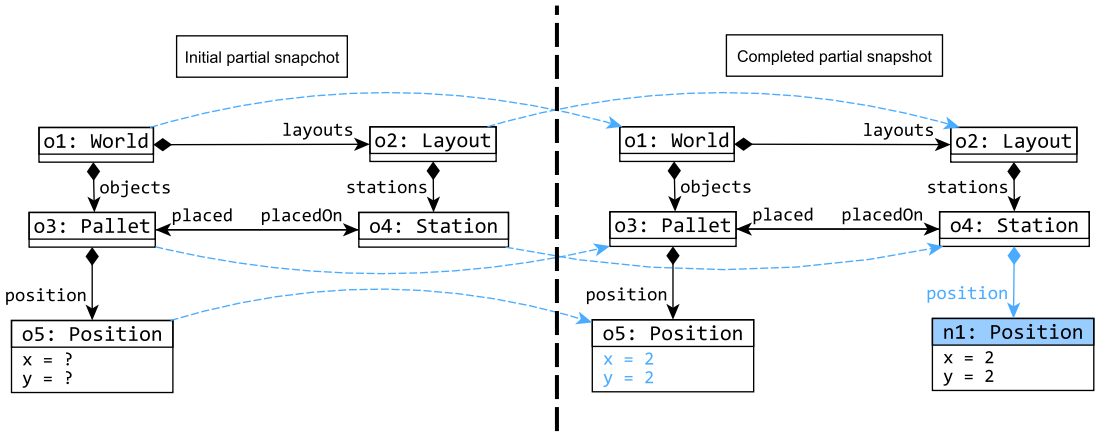


Figure 8.5: *Traceability example*

8.1.2 Traceability

The traceability is the ability to follow the lifecycle of elements, models and requirements in both a backward and forward direction.

During the mapping process, the constraints and the metamodel do not change. Only the objects of instance model are able to change, which changes are influenced by the constraints. So the traceability of the instance model have to be solved. The possible changes are the following:

- The missing attribute value of the objects are filled out.
- The objects which type is abstract or interface are concretised to real type.
- New objects are added to the model.
- New references are added.

The tool gives name to the instance objects automatically when the object is created. The mapping and the Z3 do not change these names. This can be exploited when the mapping of instance model is completed by the Z3 from the resolved output SMT code, the initial instance model is not completed, the new one is built. This solution is usable because the existing objects can be identified with their names. Every objects which exists in the initial model has different name is granted by the tool. The new objects get name during the process of creating a new partial snapshot which names are unique too. The attributes are identified with their container objects and the references can be identified with the objects of their ends.

Example 21 In Figure 8.5 a mini initial and completed partial snapshot of the LGV case study is presented. The compliance of the objects are symbolized by arrows. There are three new piece of information are defined by OCL constraints and multiplicities of references:

- Position of pallet are filled out. (OCL constraint)
- The position of the station is added. (multiplicity of the position reference) It is given an individual name.
- The new position attributes are filled. (OCL constraint)

Output instance model

The output partial snapshot is a special instance model, which is more permissive than the EMF instance model. Cases which is irregular and invalid for EMF can be created so the user can see it. If the instance model is complete and regular it can be transformed to EMF instance model. The produced output can be the input of other programs and components.

Visualization

The visualization is very important because the result can be validated by the user. Different type of representations is realized because they can have different goals. Zest can be used if during the work feedback would be seen. yEd give a nicer 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 the Eclipse modelling edition contains it. The advantage of the tool is to treat of visualisation components are isolated, only the output is transformed to the language of the installed visualization plugins.

8.2 Validation of Approach

8.2.1 Experiments and runtime performance

At the end of the work we evaluated the runtime performance of the presented implementation and we also tried to identify the practical boundaries of the approach.

The runtime tasks of the framework can be separated into the following four group:

- the transformation to the SMT language,
- the execution of the Z3 tool,
- the resolution of the output SMT code and
- the visualisation.

Our analysis shows that the runtime of transformation is proportional with the size of models and the number of constraints, the resolving of the output SMT code is proportional with the size of the parsed SMT output model and finally the runtime of the visualization code generator is proportional with the size of output partial snapshot. We can conclude, that the runtime of these components is predictable, only the time of the execution of the Z3 step cannot be approximated.

We tried to analyse the influence of input specification changes to the runtime performance of Z3. We concluded that the complexity (and also the execution time) cannot have a lower bound, because also small input models can be transformed to a very complex, difficult axiom set on which the theorem prover fails to execute in a reasonable time.

Using an average personal computer, we identified, that the maximum number of nodes in the model can be around 160 000, but using more model elements results in unreasonable execution time.

We executed iterative tests using input instance models, which size was increased in every step. In this test the number of elements is grown by 1 from 1 to 100. The test sequence was executed three times and an average was calculated based on the results.

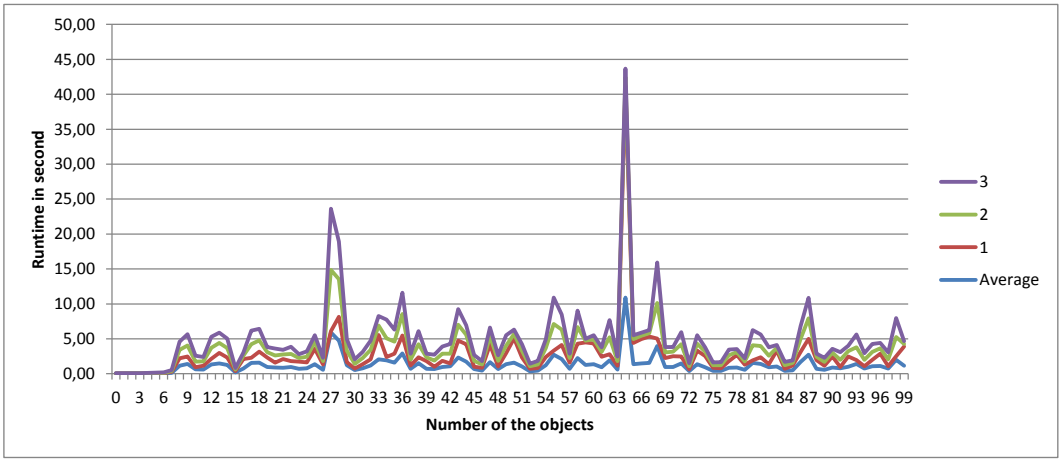


Figure 8.6: *Performance test with models which contains maximum 100 elements*

In Figure 8.6 the results of the test execution are presented. On the individual sequences there are some peaks which could be caused by different external effect, e.g. the processor of the computer is busy, but the average runtimes are under 10 seconds.

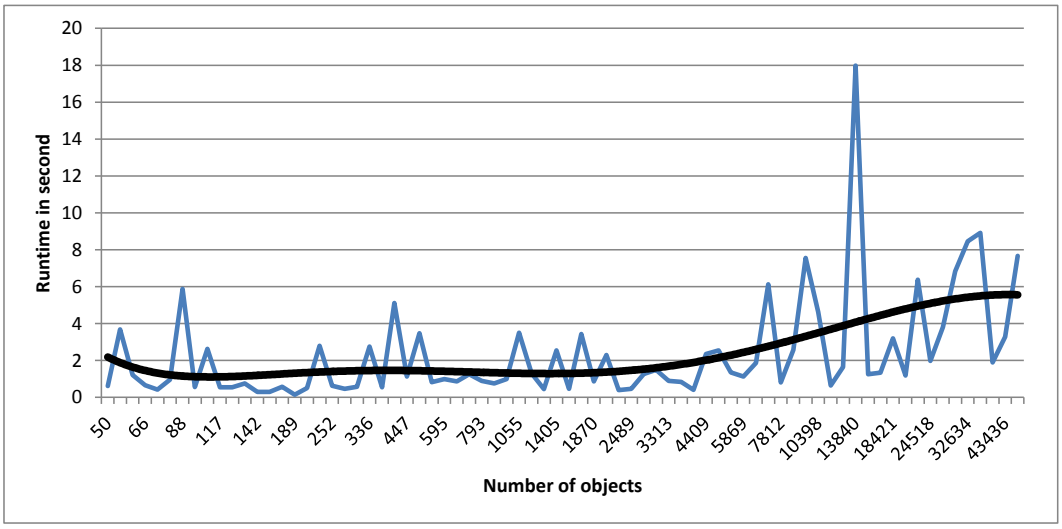


Figure 8.7: *Performance test with models which contains maximum 45000 elements*

The second test is presented in Figure 8.7. Here the maximum number of objects is grown to about 45000 and the number of the objects was increased exponentially. The diagram shows (blue line) that bigger number of elements do not cause significant performance degradation.

Our conclusion is that the runtime performance of the tool is influenced by the size of metamodel, the number of constraints and the complexity of the solved problem. The performance test of Z3 shows that the runtime is influenced mostly by the complexity of the problem because the growing of the number of elements does not cause significant performance degradation.

8.2.2 Testing

Test method

We executed functional test on the implemented tool and performed systematic tests. The compiled set of test cases covers every implemented function.

The test can be divided in three phases. In the 1st phase the consistency of the model, model query languages and the generated SMT code are examined, in the 2nd phase the consistency of the SMT code and the generated model are checked and finally in the 3rd phase the consistency of the initial and completed partial snapshot is elaborated.

The steps of the mapping can not be fully isolated, there are dependencies between them, so the test cases are not fully independent. For example the mapping of a class can be tested alone but classes have to created in order to test the mapping of references or attributes and also the mapping of the model query languages can not be done without the mapping of the entire metamodel. The mapping of the base elements are tested isolated and afterwards we systematically executed even more complex test along the dependency hierarchy.

Test case example

Due to the large number of the test cases only one test case is introduced. The selected case is the testing of the multiplicity of references, accurately the upper bound of the reference is tested which value is 1. In the Figure8.8 the example model is shown: there are a PlacedObject and a Station class which is connected by the placedOn reference which upper bound is 1.

Figure 8.8: *Upper bound mulitlicity test case*

The elements of the expected output:

- Definition of the Object type
- Definition of the PlacedObject class
- Definition of the Station class
- Type hierarchy of the classes
- Definition of the PlacedOn reference
- Declaration of the PlacedOn reference
- Mapping of the upper bound of the reference

In the following table the generated and expected results are compared. There is no difference between them.

Expected code	Generated code
`(declare-sort Object)`	`(declare-sort Object)`
`(declare-fun isType!PlacedObject (Object) Bool)`	`(declare-fun isType!PlacedObject (Object) Bool)`
`(declare-fun isType!Station (Object) Bool)`	`(declare-fun isType!Station (Object) Bool)`
`(assert (forall ((o Object)) (or(and (isType! PlacedObject o)(not(isType!Station o))) (and( not(isType!PlacedObject o))(isType!Station o) ))))`	`(assert (forall ((o Object)) (or(and (isType! PlacedObject o)(not(isType!Station o))) (and( not(isType!PlacedObject o))(isType!Station o) ))))`
`(declare-fun PlacedObject!placedOn (Object Object) Bool)`	`(declare-fun PlacedObject!placedOn (Object Object) Bool)`
`(assert (forall ((o Object) (t Object)) (=> ( PlacedObject!placedOn o t) (and (isType! PlacedObject o) (isType!Station t)))))`	`(assert (forall ((o Object) (t Object)) (=> ( PlacedObject!placedOn o t) (and (isType! PlacedObject o) (isType!Station t)))))`
`(assert (forall ((src Object) (trg0 Object) ( trg1 Object)) (=> (and (PlacedObject!placedOn src trg0) (PlacedObject!placedOn src trg1)) (= trg0 trg1))))`	`(assert (forall ((src Object) (trg0 Object) ( trg1 Object)) (=> (and (PlacedObject!placedOn src trg0) (PlacedObject!placedOn src trg1)) (= trg0 trg1))))`

To test the second phase an PlacedObject and a Station is created which are connected by the placedOn reference which model is shown in Figure 8.9. The created instance model is transformed to SMT, then the SMT code is solved and finally the result is parsed and the new instance model is created has the same structure than the initial model.

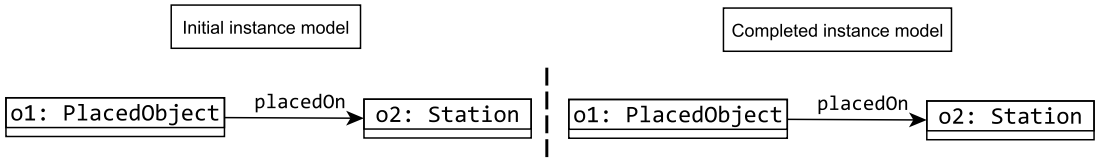


Figure 8.9: *The second phase of the testing*

Experiences

The test cases are run repeatedly in the different phases of the implementation process. During the last running all the test cases are run correctly. The implemented tool was demonstrated successfully in different industrial projects and it performed well in the different scenarios.

Chapter 9

Conclusions and Future Work

Scientific results This report summarise a new **DSL validation approach** where we proposed logic based validation method. This approach was applied on EMF formalism and can be generalised to other metamodeling techniques (like Meta-Object Facility (MOF)[40]). In the case of failed validation a **counterexample** is generated to show the reason of the inconsistency. In addition to EMF metamodel we presented the **mapping of OCL constraints EMF-IncQuery graph patterns** to first order logic formulae. To handle complex expressions of the model query languages we used sophisticated **approximation techniques**. In order ensure the decidability of the generated problem we used further approximations to map the input into **effectively propositional logic** which is a decidable fragment of first order logic. We proposed a workflow to define a **validation process** based on the independent validation tasks to ensure coverage of the whole DSL. The development of the framework **supports the ongoing research** in the industrial projects, and we adapted the approach to solve different validation tasks proposed in the case studies. Finally we developed the **back annotation** technology to support valid instance model generation.

Engineering results Our validation approach was implemented in a framework that covers the whole validation process. This framework was built on **extendable transformation modules**. Each module responsible for mapping of one DSL artifact, and additional modules can be added to the framework. This framework was **integrated into the Eclipse** which is one of the most popular of industrial relevant DSL development tools. To deal with standardised SMT language an **XText-based API** was constructed which is able to parse and query the logic structures of the Z3. We have created a **reflective editor for Partial Snapshots** to represent more general instance models. Those partial snapshots are compatible with the EMF instance models because we implemented a **bidirectional transformation** between them so result of the model generation part of our framework is a fully functional standard instance model. In additionally our implementation supports **two visualisation technologies**: (i) Eclipse integrated Zest based model view, (ii) and a yFiles based graph presentation approach. (Actually most of the figures of this report were generated in this way.)

Dissemination results Our mapping method has been successfully demonstrated during the Trans-IMA industrial project in the avionics domain. Part of this contribution was published[46] in the IEEE/ACM 16th International Conference on Model Driven Engineering, Languages and Systems (MODELS 2013) conference which is won Springer Best Paper Award. The framework was also applied in the R3-COP ARTEMIS project where the tool was used for automated model-based test generation and was successfully demonstrated in the final review meeting.

9.1 Future Work

The first advancement option is the development of more sophisticated validation campaigns that consists of multiple validation and model generation executions. Those campaigns can be used to enumerate different model results, or search models that maximize the value of a model metric given as input. The second option can be used to generate models with maximal boundary values for example for robustness tests.

The second goal of our future work is to extend our DSL validation process to further aspects of the language design. For example, to avoid inconsistency in the query-based definition view models (similar concepts as views in relational databases).

It would also be interesting to compare our framework to other methods that executes reasoning tasks or consistency checks over models, in particulary to different ontologies[56].

Finally, to investigate the applicability of my approach in context with the new DO-178C certification standard [39] for civil avionics software development that accepts formal validation as certification artifacts.

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] ARINC - Aeronautical Radio, Incorporated. A653 - Avionics Application Software Standard Interface.

[7] AUTOSAR Consortium. *The AUTOSAR Standard.* http://www.autosar.org/.

[8] B. Beckert, U. Keller, and P. H. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *Proc of the VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002.

[9] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental Evaluation of Model Queries over EMF Models. In *MODELS'10*, volume 6395 of *LNCS*. Springer, 2010.

[10] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for emf models. In Jordi Cabot and Eelco Visser, editors, *Fourth International Conference on Theory and Practice of Model Transformations*, volume 6707 of *LNCS*, pages 167–182. Springer, June 2011.

[11] A. D. Brucker and B. Wolff. The HOL-OCL tool, 2007. http://www.brucker.ch/.

[12] Fabian Büttner and Jordi Cabot. Lightweight string reasoning for OCL. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitrios S. Kolovos, editors, *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Lyngby, Denmark, July 2-5, 2012. Proceedings*, volume 7349 of *LNCS*, pages 244–258. Springer, 2012.

[13] Fabian Büttner, Marina Egea, and Jordi Cabot. On verifying ATL transformations using 'off-the-shelf' SMT solvers. In *Proc. of the 15th Int. Conf. on Model Driven Engineering Languages and Systems*, volume 7590 of *LNCS*, 2012.

[14] 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.

[15] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. A UML/OCL framework for the analysis of graph transformation rules. *Softw. Syst. Model.*, 9(3):335–357, 2010.

[16] Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 547–548, New York, NY, USA, 2007. ACM.

[17] Jordi Cabot, Robert Clarisó, and Daniel Riera. First international conference on software testing verification and validation. In *Verification of UML/OCL Class Diagrams using Constraint Programming*, pages 73–80. IEEE, 2008.

[18] M. Clavel and M. Egea. The ITP/OCL tool, 2008. `http://maude.sip.ucm.es/itp/ocl/`.

[19] Manuel Clavel, Marina Egea, and Miguel Angel García de Dios. Checking unsatisfiability for OCL constraints. *ECEASST*, 24, 2009.

[20] 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.

[21] The Eclipse Project. *MDT OCL*. `http://www.eclipse.org/modeling/mdt/?project=ocl`.

[22] Eclipsepedia. MDT/OCLinEcore, 2013. `http://wiki.eclipse.org/MDT/OCLinEcorel`.

[23] Florian Lapschies. *SONOLAR*. `http://www.informatik.uni-bremen.de/~florian/sonolar/`.

[24] Miguel Garcia. How to process ocl abstract syntax trees. 2007.

[25] Yeting Ge and Leonardo Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer Berlin Heidelberg, 2009.

[26] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Softw. Syst. Model.*, 4(4):386–398, 2005.

[27] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System model-based definition of modeling language semantics. In *Formal Techniques for Distributed Systems*, volume 5522 of *LNCS*, pages 152–166. Springer, 2009.

[28] Ábel Hegedüs, Ákos Horváth, István Ráth, and Dániel Varró. Query-driven soft interconnection of EMF models. In *Proc of the Int. Conf on Model Driven Engineering Languages and Systems*, volume LNCS 7590, pages 134–150, 2012.

[29] 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.

[30] Ethan K. Jackson, Wolfram Schulte, and Nikolaj Bjørner. Detecting specification errors in declarative languages with constraints. In *Proc. of the 15th Int. Conf. on Model Driven Engineering Languages and Systems*, volume 7590 of *LNCS*, pages 399–414, 2012.

[31] 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.

[32] 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.

[33] 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.

[34] Laboratoire de Recherche en Informatique, Inria Saclay Ile-de-France and CNRS. *Alt-Ergo SMT Solver*, October 2013. `http://alt-ergo.ocamlpro.com/`.

[35] Levi Lucio, Bruno Barroca, and Vasco Amaral. A technique for automatic validation of model transformations. In *Proc. of the 13th Int. Conf. on Model Driven Engineering Languages and Systems*, volume 6394 of *LNCS*, pages 136–150, 2010.

[36] Mathworks. *Matlab Simulink - Simulation and Model-Based Design.* `http://www.mathworks.com/products/simulink/`.

[37] Niklas Eén, Niklas Sörensson. *MiniSAT.* `http://minisat.se/`.

[38] The Object Management Group. *Object Constraint Language, v2.0*, May 2006. `http://www.omg.org/spec/OCL/2.0/`.

[39] Special C. of RTCA. DO-178C, software considerations in airborne systems and equipment certification, 2011.

[40] omg. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[41] Oszkár Semeráth. Validation of Domain Specific Languages, 2013. Technical Report, `https://incquery.net/publications/dslvalid`.

[42] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjorner. Deciding effectively propositional logic with equality, 2008. Microsoft Research, MSR-TR-2008-181 Technical Report.

[43] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data Knowl. Eng.*, 73:1–22, 2012.

[44] István Ráth, Ábel Hegedüs, and Dániel Varró. Derived features for EMF by integrating advanced model queries. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitris Kolovos, editors, *Modelling Foundations and Applications*, volume 7349 of *Lecture Notes in Computer Science*, pages 102–117. Springer Berlin / Heidelberg, 2012.

[45] SAE - Radio Technical Commission for Aeronautic. Architecture Analysis & Design Language (AADL) v2, AS-5506A, SAE International, 2009.

[46] Oszkár Semeráth, Ákos Horváth, and Dániel Varró. Validation of derived features and well-formedness constraints in dsls. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 538–554. Springer Berlin Heidelberg, 2013.

[47] 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.

[48] 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.

[49] Technical University of Catalonia. *Barcelogic for SMT*, November 2005. `http://www.lsi.upc.edu/~oliveras/bclt-main.html`.

[50] The Eclipse Project. *Eclipse Modeling Framework*. `http://www.eclipse.org/emf`.

[51] The Eclipse Project. *Xtend*. `http://www.eclipse.org/xtend/`.

[52] The Eclipse Project. *Xtext*. `http://www.eclipse.org/Xtext/`.

[53] The Eclipse Project. *Zest*. `http://www.eclipse.org/gef/zest/`.

[54] Thomas Dillig, Isil Dillig, Ken McMillan, Alex Aiken. *Mistral SMT Solver*, December 2012. `http://www.cs.wm.edu/~tdillig/mistral/index.html`.

[55] Dániel Varró and András Balogh. The Model Transformation Language of the VIATRA2 Framework. *Science of Computer Programming*, 68(3):214–234, October 2007.

[56] Tobias Walter, Fernando Silva Parreiras, and Steffen Staab. Ontodsl: An ontology-based framework for domain-specific languages. In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009*, volume 5795 of *LNCS*, pages 408–422. Springer, 2009.

[57] 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.

[58] Jessica Winkelmann, Gabriele Taentzer, Karsten Ehrig, and Jochen M. Küster. Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. *ENTCS*, 211(0):159 – 170, 2008. Proc. of the 5th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'06).

[59] yEd Graph Editor. *yED*. `http://www.yworks.com/en/products_yed_about.html`.