MŰEGYETEM 1782

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék
Department of Measurement and Information Systems

Modelltranszformációk formális helyességellenőrzése

MSc diplomamunka

Szerző:
Semeráth Oszkár

Konzulensek:

Dr. Varró Dániel	Dr. Horváth Ákos	Szatmári Zoltán
egyetemi docens	tudományos munkatárs	tudományos segédmunkatárs

2013. december 13.

DIPLOMATERV-FELADAT

Semeráth Oszkár (J053M6)

szigorló mérnök informatikus hallgató részére

Modelltranszformációk formális helyességellenőrzése

Napjainkban a modellvezérelt tervezési módszertan egyre növekvő térhódítása figyelhető meg a rendszertervezés területén. A modellvezérelt tervezést egy precíz modellezési fázis vezet be, amely során számos nézőpontból vizsgáljuk meg a tervezés alatt álló rendszert, majd az integrált rendszertervet precíz matematikai analízisnek vetjük alá automatikus modelltranszformációk által. A garantált minőségű rendszertervből kiindulva az alkalmazás forráskódját és a telepítési információt automatikus kódgenerálás segítségével származtatjuk, amelyet tipikusan szintén egy speciális modelltranszformációnak tekinthetünk

Automatikus modelltranszformációk alkalmazásakor azonban problémát jelent, hogy vajon mennyire megbízhatóak maguk a modelltranszformációk. A gráftranszformációk paradigmája egy intuitív és egyben precíz formális módszert ad e modelltranszformációk specifikációjára. Amennyiben a transzformációk specifikációi koncepcionális hibákat tartalmaznak, ezek megjelennek a modelltranszformációk végrehajtásakor is, így gyakorta maguk a modelltranszformációk jelentik a minőségi szűk keresztmetszetet. Ezt elkerülendő, egyre inkább elengedhetetlenné válik a modelltranszformációk formális helyességellenőrzése.

A modelltranszformációk helyességellenőrzése során egyrészt garantálni kell, hogy a célnyelv jólformáltsági kritériumai teljesülnek, valamint további, transzformáció-specifikus kritériumok helyességét is igazolnunk kell, általános esetben tetszőleges forrásmodellből kiindulva. Ehhez precíz absztrakciók segítségével elsőrendű logikai formulákat származtatunk, amelyet automatikus tételbizonyítók segítségével dolgozhatunk fel.

A jelölt feladatai a következők:

- Tekintse át a szakterület-specifikus nyelvek és transzformációik specifikációjára az iparban széleskörűen használt Eclipse Modeling Framework (EMF) feletti eszköztárat.
- Dolgozzon ki leképezést a gráfminták nyelvéről elsőrendű logikai formulákba, ügyelve hogy lehetőleg az elsőrendű logika jól analizálható résznyelvét kapjuk.
- Javasoljon módszert a gráfminták által definiált származtatott attribútumok, jólformáltsági kényszerek és absztrakciót végző modelltranszformációk helyességvizsgálatára SMT-megoldók felhasználásával.
- Készítse el az analízis keretrendszer kísérleti implementációját, és egy esettanulmányon igazolja a kidolgozott rendszer gyakorlati alkalmazhatóságát.

Tanszéki konzulens: Dr. Varró Dániel (egyetemi docens)

Budapest, 2013. szeptember 30.

……………………

Dr. Jobbágy Ákos
tanszékvezető

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 diplomamunka célja, hogy olyan eszközt biztosítsak, 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 diplomamunkámban egy egységes keretrendszert javasolok a szakterület-specifikus nyelvek konzisztenciavizsgálatára a következő módon: (i) A jólformáltsági kényszereket és származtatott érték definícióját egységesen elsőrendű logikai kifejezésekké fordítom, amelyeken SMT megoldókkal végzek következtetéseket. (ii) Approximációs technikákat alkalmazva egy hatékonyan elemezhető logikai fregmensbe képzem az komplexebb nyelvi elemeket. (iii) A validációs eszközüt ipari modellező eszközhöz integráltam, amely az ellentmondásokat a nyelv szabványos példánymodelljeiként állítja elő.

Módszerem 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 eszközt két ipari követelményekkel rendelkező esettanulmányon is sikerrel alkalmaztam. 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 eszköz feladata a konkrét tesztesetek előállítása volt.

Köszönetnyilvánítás

Szeretnék köszönetet mondani dr. Varró Dániel konzulensemnek hogy több éves támogatásával lehetővé tette sikereim elérését. További köszönet illeti dr. Horváth Ákost és Szatmári Zoltánt tanácsaikért, iránymutatásukért, valamint Barta Ágnest gyümölcsöző közös munkánkért.

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

Consistency Analysis of Domain-Specific Languages

MSc Thesis

Author:
Oszkár Semeráth

Advisors:

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

December 13, 2013

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 thesis 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, I 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.

The 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.

I 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.

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) [48], 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 [55] 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 objectives of this report are:

- The main objective is to create an **automated framework** to **formalize** DSL modeling artifacts (including meta- and instance models, constraints and derived feature definitions) by logic descriptions.
- Execute wide range of validation task by automated theorem proving on the formalised language.
- To improve the quality of the developed DSL by **validating** different requirements of the domain specific language such as **consistency**, **ambiguity** and **completeness**.
- 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.
- To automate other development activities by **generating multiple 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.

I also developed a research prototype tool to demonstrate the practical feasibility of my approach. My tool takes DSL specifications in the form of EMF models, which is an open source technology widely used in the industry. Model queries are specified using declarative graph patterns as available in the EMF-IncQuery framework. I 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.

The tool has been successfully applied in case studies of two ongoing industrial projects taken from the avionics which is documented in this thesis and autonomous and cooperative robot system domain. Initial experiments have been carried out to assess the performance characteristics of my validation tool.

1.5 Structure of the Report

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

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[34] over different hardware architectures and (ii) develop an MDE based tooling platform for supporting the definition of the allocation process.

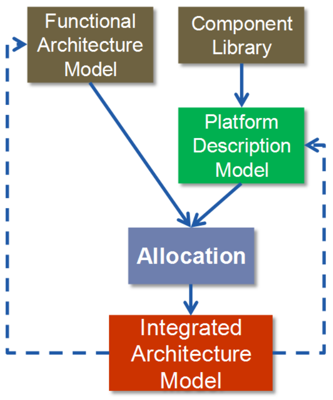

Functional Architecture Model
Component Library
Platform Description Model
Allocation
Integrated Architecture Model

Figure 2.1: *High-level overview of the Trans-IMA project*

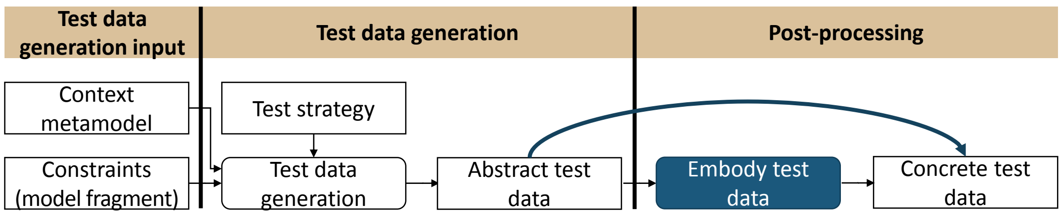


Figure 2.2: *Test generation scenario*

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.

A high-level overview of the test data generation is depicted in Figure 2.2. 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).

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 constraints. Due to the two phased test generation, first an abstract test data is constructed, where some constraints can be violated or abstract model elements can be used. The goal of the 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

This chapter summarises the most important practical and theoretical concepts of this thesis. First the definitions of modeling and metamodeling are introduced, then the well-formedness constraints and derived features are presented which can be formulated as extra rules on the models. After this, the mathematical problem solvers, the problem classes (SAT, CSP, SMT) are shown. Finally, the chapter summarises the related works.

3.1 Domain-specific Modelling

3.1.1 Metamodeling

Metamodels are the models of the modeling languages. They are to collect the concepts, relations and attributes of the target domain, and define the structure of the models. In this thesis, the Eclipse Modeling Framework (EMF) [48] is used for domain specific modeling.

In this formalism the concepts are represented by EClasses (which are often referred simply as classes), which can be instantiated to EObjects (or objects). Between the classes EReferences are defined which models the relations of the domain. When two objects are in relation an EReference is instantiated. EAttributes are added to the EClasses to enrich the expression power of the modelling language with predefined primitive values like integers and strings. Attributes and relations together called structural features. The multiplicity of structural features can be limited with upper and lower bound in lower..upper form. Two parallel but opposite directional reference can be defined as inverse of each other to specify that they always occur in pairs. The instance models are arranged into a directed tree hierarchy by relations marked as a containment.

Generalisation relation can be specified between two classes to express that a class has every structural feature of the more general one. Multiple inheritance is supported. Some classes are Abstract or Interface, those ones do not have direct instances.

The instance model is an instance of the metamodel, it is a specific interpretation of the defined concepts.

3.1.2 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 [42, 36]. Such queries have two parameters, in case of (i) derived references one parameter represents the source and another the target objects of the reference while in case of (ii) derived attributes one parameter represents the container object while the other one the computed value of its attribute.

3.1.3 Well-formedness Constraints

We also define some Structural well-formedness (WF) constraints are usually derived from design rules and guidelines to be validated on functional architecture models. The role of the WF constraints is the same as the OCL[36] invariants. In our current approach WF constraints define ill-formed model structures and thus they cannot have a match in a valid model.

3.1.4 EMF-IncQuery Graph Patterns

Graph patterns [53] 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 [26].

3.2 Mathematical Logic

In this section the first order logic axiom system and its problem classes are introduced.

3.2.1 First-order Logic

First-order logic (FOL) is a formal language widely used in both mathematics and computer science. A logic language is consists of: (i) the collection of domains (types), (ii) function, constant, relation and predicate symbols over the domains and (iii) a collection of formulae, which defines the assertions. Figure 3.1 summarises the syntax of the language of first order logic. It parallelly shows the traditional mathematical notation and the SMT2[3] standard syntax which is used in this thesis.

The central question of logic is to decide whether an axiom system defined in logic language is satisfiable or not. A logic structure of an axiom system is a collection of definition which specifies the elements of the unspecified domains and gives an interpretation of each uninterpreted

Expression		Mathematical	SMT2 standard
		Types	
Expression		Mathematical	SMT2 standard
Boolean	$\rightarrow$	$\{true, false\}$	`Bool`
Integer	$\rightarrow$	$\mathbb{Z}$	`Int`
Real	$\rightarrow$	$\mathbb{R}$	`Real`
Set	$\rightarrow$	S	`(declare-sort S)`
(definition)	$\mid$	$S = \{e_1, \ldots, e_n\}$	`(declare-datatypes () ((S e!1 ... e!n)))`
		Symbolic Value Declaration and Definition	
Expression		Mathematical	SMT2 standard
Function	$\rightarrow$	$f : type_1, \ldots, type_n \mapsto type$	`(declare-fun f (type!1 ... type!n) type)`
(definition)	$\mid$	$f(x_1, \ldots, x_n) = y$	`(define-fun f (type!1 ... type!n) type y)`
Constant	$\rightarrow$	$c : \emptyset \mapsto type$	`(declare-fun c () type)`
Relation	$\rightarrow$	$R : type_1, \ldots, type_n \mapsto \{true, false\}$	`(declare-fun R (type!1 ... type!n) Bool)`
		Terms and Formulae	
Expression		Mathematical	SMT2 standard
Formula	$\rightarrow$	$relation(term_1, \ldots, term_n)$	`(relation term!1 ... term!n)`
	$\mid$	$\neg formula$	`(mot formula)`
	$\mid$	$formula_1 \wedge formula_2$	`(and formula!1 formula2)`
	$\mid$	$formula_1 \vee formula_2$	`(or formula!1 formula2)`
	$\mid$	$formula_1 \Rightarrow formula_2$	`(=> formula!1 formula2)`
	$\mid$	$term_1 = term_2$	`(= term!1 term!2)`
	$\mid$	$term_1 \neq term_2$	`(distinct term!1 term!2)`
	$\mid$	$\exists \quad var_1 \in type_1, \ldots, var_n \in type_n : formula$	`(exists ((var!1 type!1) ... (var!n type!n)) formula)`
	$\mid$	$\forall \quad var_1 \in type_1, \ldots, var_n \in type_n : formula$	`(forall ((var!1 type!1) ... (var!n type!n)) formula)`
Term	$\rightarrow$	$function(term_1, \ldots, term_n)$	`(function term!1 ... term!n)`
	$\mid$	(Variable) var_1	`var!1`
	$\mid$	(Individual) e_1	`e!1`

Figure 3.1: *Mathematical and SMT2 Standard syntax of first order logic*

function. In the terminology of the SMT2 language the structure specifies the elements of each declared type as constants and fully defines the declared functions. A logic structure is a model of the axiom system if it satisfies its every assertions. A Σ axiom system is satisfiable if it has a model ($\Sigma \models M$). Otherwise, it is called unsatisfiable.

In general, the satisfiability of an arbitrary axiom system is undecidable. However, there are many method to reason over special fragments of the first order logic. In the following the most popular reasoning processes will be presented

3.2.2 Satisfiability Problem (SAT)

To define the SAT language the Boole-formula should be introduced. The Boole-formula is built up from Boolean constants constants, their negated expressions, the $\wedge$ ("and") and the $\vee$ ("or") operands. The result of an evaluated formula is true or false. The Boole-formula is satisfied if their variables has an evaluation where the value of the formula is true.

The input of the SAT problem is the set of uninterpreted Boolean constants and the formulae. The SAT solver searches a substitution to the values which makes each formula to be evaluated true.

The SAT problem is decidable, but the expression power of the logic is limited to predicates. An example for the SAT solver is the MiniSat [35].

3.2.3 Constraint Satisfaction Problem (CSP)

The formal definition of the CSP problem is formulated by the set of constants (C_1, $C_2 \ldots C_n$) and the collection of restrictive constraints formulae. Each C_i constant has a finite D_i domain of possible values. The task is to generate a model for the input axiom system which assigns values from the finite domains to the constants.

The CSP is decidable problem, but handles finite domains only. The Boole CSP is the special case of NP-complete problems. A CSP solver is e.g. the Sugar [2].

3.2.4 Satisfiability Modulo Theories (SMT)

The SMT problem is a decision problem for full range of first order logic formulae with the combinations of several background theories. The expression power of the SMT axiom system satisfiability is richer than in CSP because it supports types with undefined and unlimited elements. Therefore it is suited to check potentially infinite possibilities.

In general, the SMT axiom system satisfiability is undecidable. However, many task can be solved with the help of rich set of background theorems which includes reasoning methods for of arithmetic, arrays, algebraic datatypes, function and sets.

Many SMT solvers use the standard SMT-LIB [3] language which is previously presented in Figure 3.1. This allows the user to use different solvers with a unified interface, like the Alt-Ergo [32], Barcelogic [47], Beaver [29], CVC4 [1], Mistral [52], SONOLAR [21], Yices [4], Z3 [20]. From them the state-of-the-art Z3 is used to analyse domain-specific languages.

3.3 Related Work

There are several approaches and tools aiming to validate UML models enriched with OCL constraints [24] relying upon different logic formalisms such as constraint logic programming [16, 17, 12], SAT-based model finders (like Alloy) [5, 14, 31, 46], first-order logic [8, 19], constructive query containment [41], higher-order logic [11, 25], or rewriting logics [18]. Some of these approaches (like e.g. [17, 14, 31]) 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 [27] 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 [56] 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, 33, 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 [45, 27, 30]. 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 [28] 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

This chapter will introduce the functional overview of my DSL validation approach. It gives the precise definition of the implemented reasoning tasks and describes which language development artifact how can be processed with what additional configuration options.

The end of the chapter some EMF specific technical details will be presented.

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 created. 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

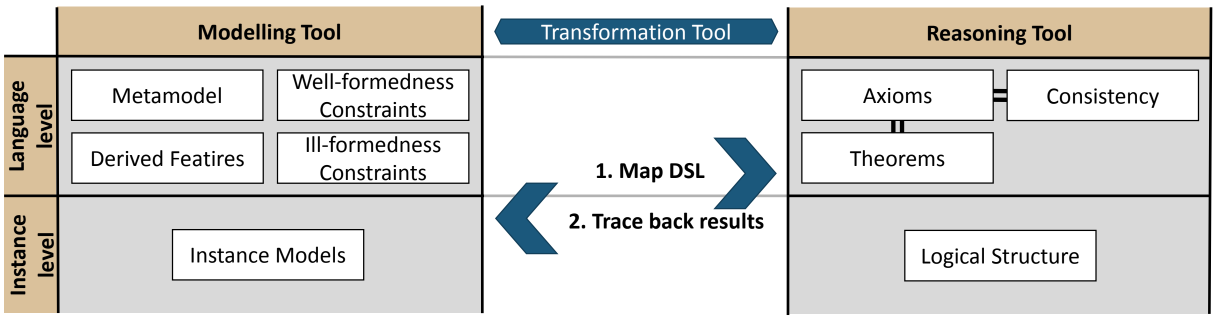


Figure 4.1: *Functional overview of the approach*

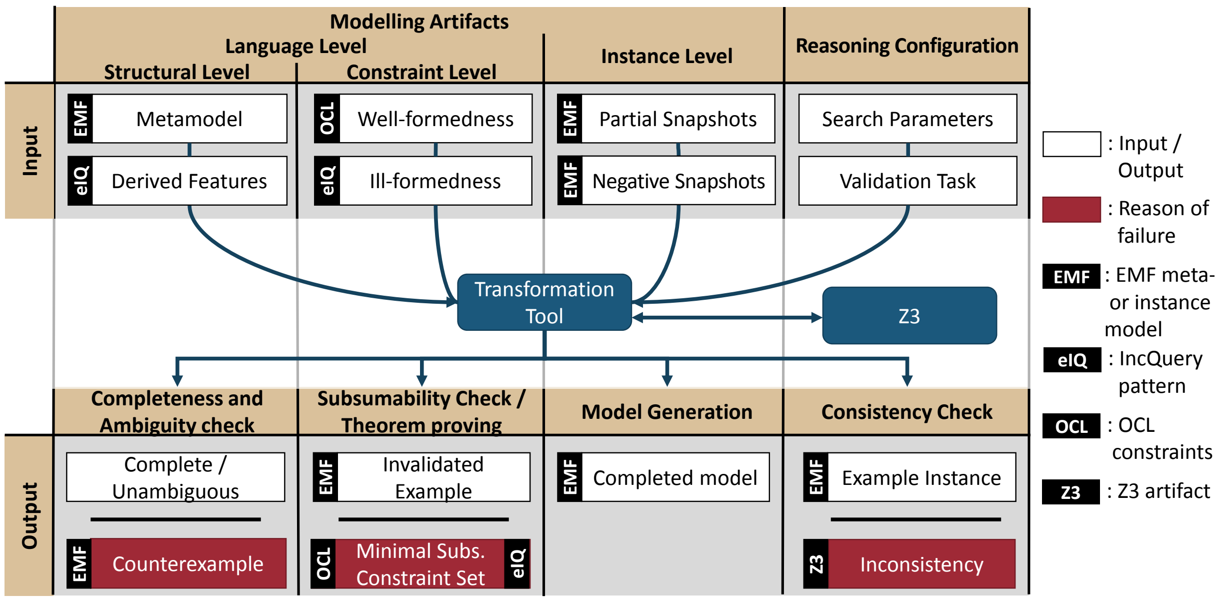


Figure 4.2: *Prototype tool features*

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.

4.2 Validated Language Elements

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 defines the language in a constructive way, and a Constraint Level that restricts it. In the following those ones will be introduced.

Parameters in the Structural Level refer to the defining elements of the target domain-specific language. 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 are 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 and 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 similarly 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 every 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 defined in the level of the instance models.

Partial Snapshots By adding initial structures to the Instance Level the reasoning process will be more directed as the tool checks only the cases that contain those structures as submodels.

Negative Snapshots In the other hand the user might want to make the reasoning process to ignore specific constructs. Instance models can be added to the configuration as Negative Partial Snapshots which defines that the range of to examined models excludes the cases that contains the specific structure as a submodel.

The parameters in the Reasoning Level allows to customise the reasoning process. Beside the few technical details like time limit 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 over-approximation level). Additionally, the amount of the required models can be set, so multiple different instance models can be generated.

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 section.

4.3 Validation Tasks

Many different validation tasks can be executed on the selected domain-specific language. At first, this section describes the basic reasoning method, then describes the differences on each validation task.

4.3.1 General Reasoning over the DSL

Generally, our tool searches for an instance model which:

- Instance of the Metamodel and satisfies every structural constraints including the Derived Features
- Satisfies all the Constraints

The reasoning process can be focused by limiting the search with additional parameters. The tool generates models that:

- are the completion of every Partial Snapshot and does not contain any Negative Snapshot
- satisfy every Search Parameters

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 used in each reasoning task with some task-specific modification. The following subsections details those differences.

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 case where the derived feature is faulty. Currently the completeness and unambiguousness of the DFs are checked.

Completeness is understood 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.

Unambiguity is defined 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 of the feature.
- Moreover, there is an instance of the source type that actually violates the multiplicity constraint.

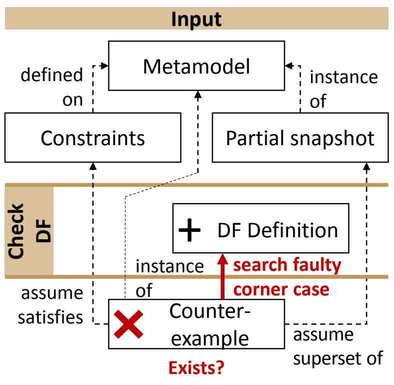


Figure 4.3: *Derived feature validation setup*

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.

4.3.3 Subsumability Check and Theorem Proving

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 4.3.5.

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 decide that the a constraint is derivable from the DSL specification (in mathematical notation: *DSL specification* $\models$ *constraint*). In this way the satisfaction of a certain constraint can be proved in every possible instance model of a language as opposed to the model validator that checks only the edited one.

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, therefore a subsumable constraint is considered unnecessary.

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 do not limit the search.

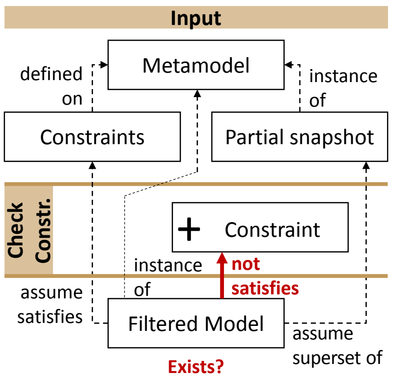


Figure 4.4: *Subsumability check setup*

4.3.4 Model Generation

Our tool can be specialised to generate instance models of the chosen DSL. The Partial Snapshots and the flexible model size limit make model generation be highly customisable.

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 Snapshots, and the Search Parameters. It is possible that those requirements are unrealizable, in that case the failure is communicated to the user.

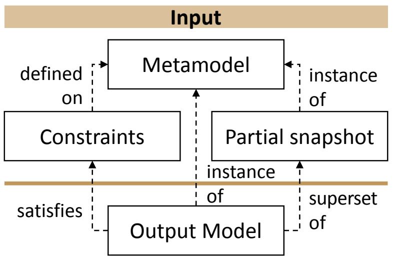


Figure 4.5: *Model Generation setup*

4.3.5 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).

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 have any valid instance model).*

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

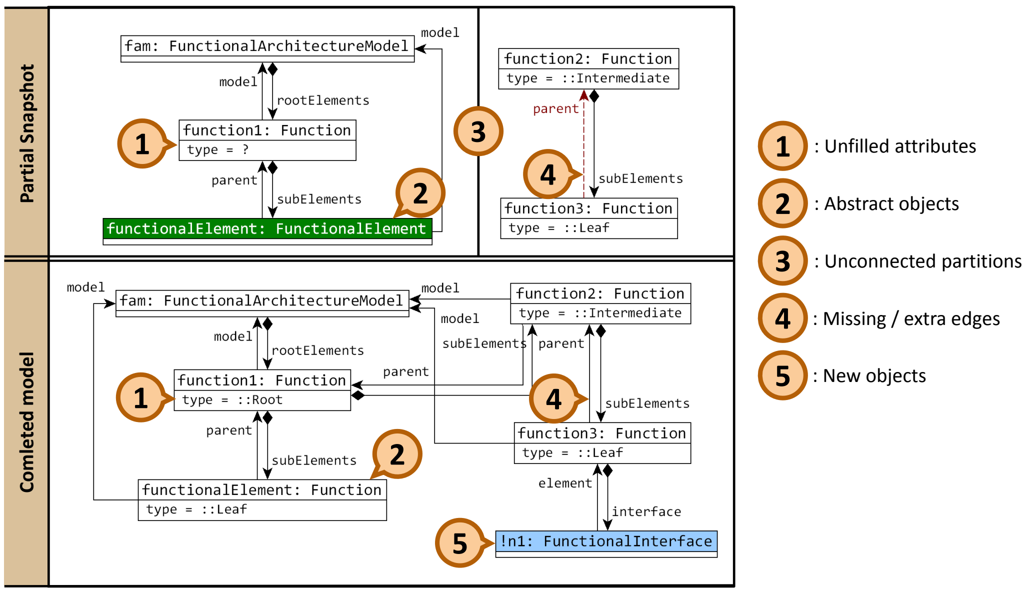


Figure 4.7: *Extra options available in Partial Snapshot*

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 then the DSL is proved to be inconsistent.

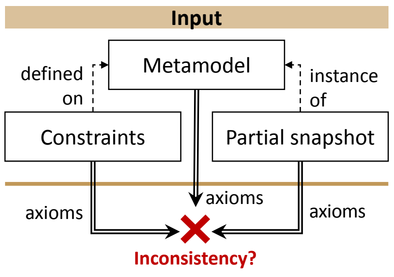


Figure 4.6: *Setup of Consistency check*

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.

4.4 Configuration of the Reasoning Process

The approach of validating in general is discussed in the previous sections, however the current section summarizes the EMF specific configuration.

4.4.1 Partial Snapshot

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. To overcome this limitation, I have created a formalism called Partial Snapshots.

The partial snapshot (PS) is an extended instance model compared to the standard EMF framework allows. Additionally, every object is identified with a unique name, like o1 or PS/o1 where PS is a partial snapshot. Figure 4.7 presents a Partial snapshot example and one of its possible completion called Completed model. The following features are added to the instance models:

1. Undefined attributes: In a normal EMF instance object each attribute has a value (or preseted default value)[1]. Many use-cases 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. Abstract objects: Partial snapshots allow to instantiate abstract or interface EClasses. They are handled similarly as concrete object like as they can have attributes and references. The type of an non-concrete object has 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 hierarchical containment tree.

4. Missing / extra edges: Our Partial Snapshot editor does not automatically manage inverse edges, thus it is possible that there is a reference without the its inverse counterpart (like in point 4. where the missing reference is indicated with dashed line). In Partial Snapshots the number of references can also violate the multiplicities.

5. New objects: The Partial Snapshot defines only an initial structure which can be extended with additional objects, like !n1.

The validation process can be parametrised with multiple PSes, and the reasoning process tries to satisfy each one. Figure 4.8 shows three partial snapshots (top) and an instance model that satisfies them (bottom). The first (PS1) defines the initial structure of a model that contains a Function (r) with two subfunctions (l1 and l2). PS2 requires a Function with two interfaces. Finally, the third defines a three deep containment hierarchy of Functions.

A referred PS can be configured with different semantic options (where the first is the default), as it is labelled in Figure 4.8:

1. Positive / Negative: A Positive partial snapshot is an incomplete instance model that every result have to contain as a submodel. The occurrence of containment can be back annoted in the result model, like the node o3 in the example model of Figure 4.8 created from the function that contains two subfunctions (r object of PS1) and from function that implements two interfaces (f object of PS2). This is marked with the (PS1/r,PS2/f) expression.

   Referring a PS Negatively makes the opposite effect: if a model contains the submodel then it is invalid. For example PS3, that defining three deep containment, is referred as a negative PS, which filters out models with function hierarchy deeper than two levels.

2. Injective / Shareable: In case of an Injective PS the objects of the snapshot have to be mapped different instance objects in the result model. For example, the two subfunctions of PS1/r cannot be mapped to the same function in the generated model.

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

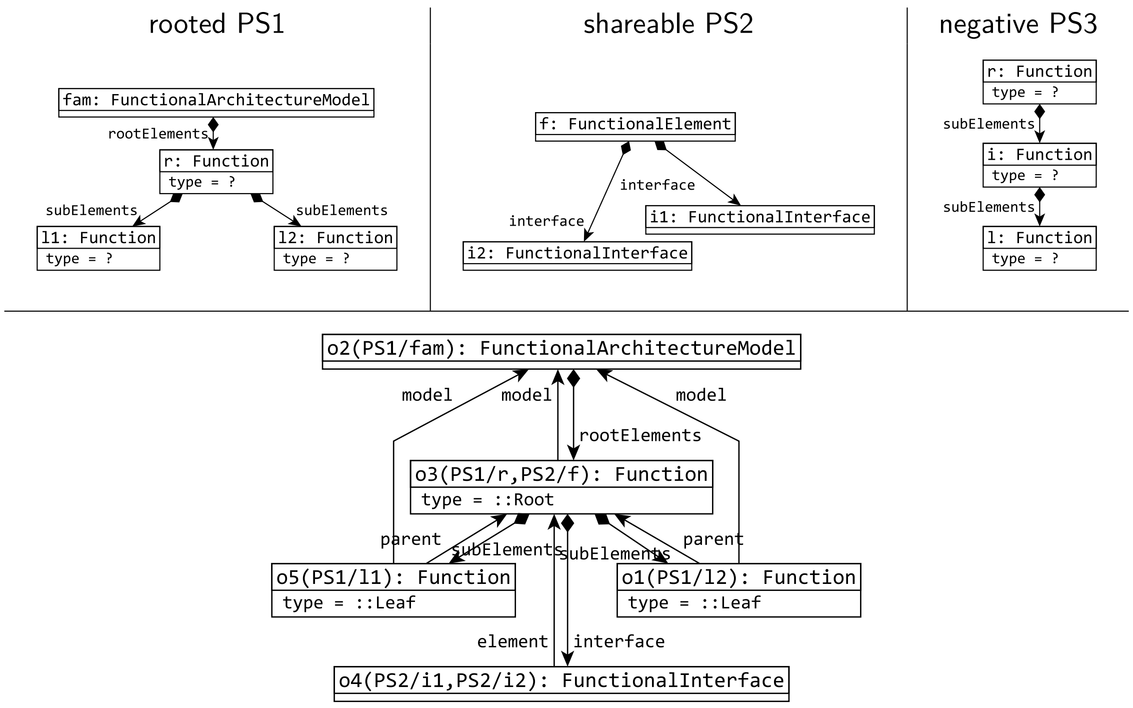


Figure 4.8: *Multiple different partial snapshots*

In Shareable PSes multiple elements can be mapped into the same object if that satisfies the requirements of each source, like the i1 and i2 interfaces of PS2 that are both mapped to the o4 object of the result model in our example.

3. Unrooted / Rooted: In Rooted PSes the root of the selected partial snapshot have to be mapped to the root of the result model, while in Unrooted mode it is not necessary. For example the root of the PS2 snapshot loses its role, but PS1/fam have to be the root in every generated model.

4. Modifiable / Unmodifiable: By default, the PSes are Modifiable which means that the reasoning process can add model element to them in an incremental way, for example add extra relations to the objects or fill empty attributes. An Unmodifiable PS means that the process can not add any new element to the submodel, for example if two objects in the PS was unrelated with a relation they will remain unrelated in the result model too. Embedding the PS into a model with newly created incoming or outgoing relations is still allowed.

   For example, if the PS1 would be noted as an unmodifiable pattern it would be unsatisfiable, because it is not allowed to create the inverse parent relations of subElements.

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

4.4.2 Search Parameters

There are many ways to further configure the reasoning process. The most useful and adjustable ones are discussed in this subsection.

Model size It is possible to explicitly define the size of the checked models. As Figure 4.9 shows, the range of the checked models can be set to:

1. Initial only: Only the objects in the first Partial Snapshot can be used; new objects cannot be created. This option is ideal for simple model-completion tasks.
2. Limited to size: The overall number of objects in the model has a predefined limit.
3. Unlimited: Every model is checked without size limitation.

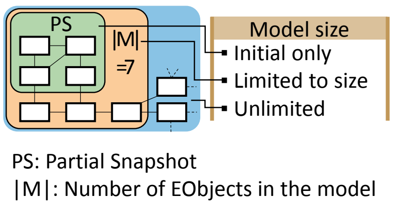


Figure 4.9: *Model Size*

Model Count In some use cases it is required to generate multiple answers to the reasoning problem. The tool is capable of generating different models by iteratively adding the previous answers as invalid model and then forcing the reasoner to find a new answer. The available options for model count are:

1. Simple: By default, the tool generates only one result.
2. Fix Amount: The tool tries to generate a fix number of different instance models. If the tool realises that there are not any more different results, it proves that and then stops.
3. Unlimited: Same as the Fix Amount, but it does not stops until it runs out of possible results.

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 a limit called approximation level.

4.5 Summary

In this chapter we have defined the functional input and output and the possible configuration options of the DSL validation approach. The discussed validation services are demonstrated on a case study in Chapter 5, and the reasoning process is detailed in Chapter 6.

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.

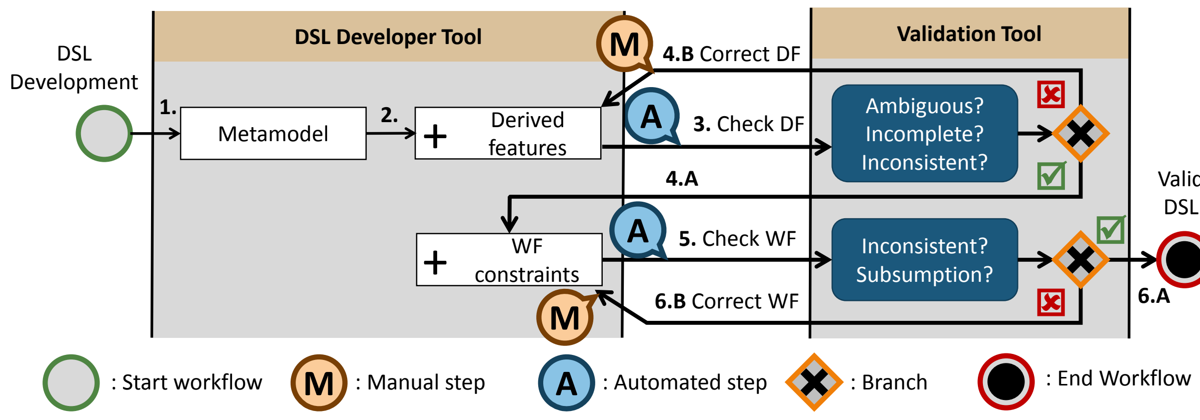


Figure 5.1: *DSL validation workflow*

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 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.

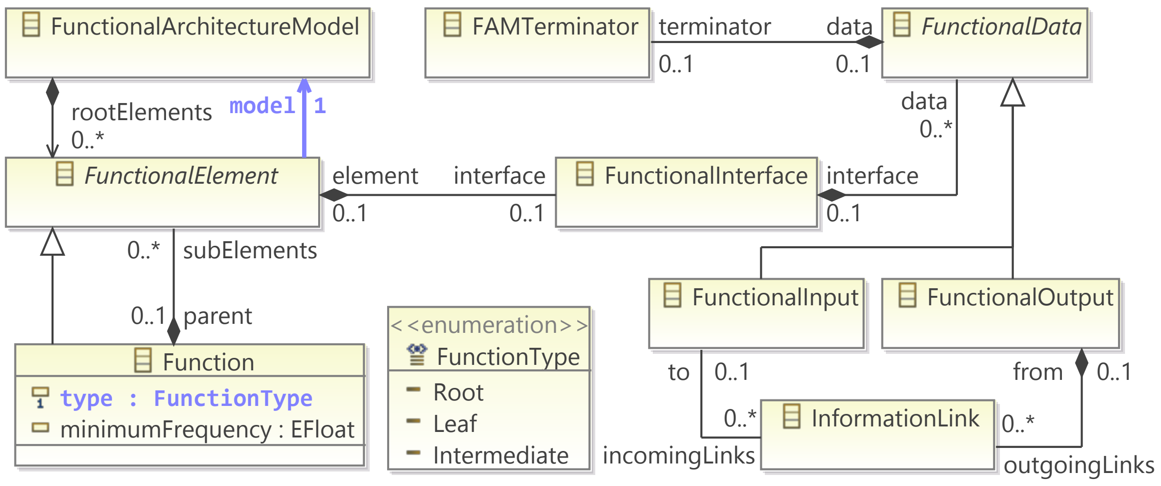


Figure 5.2: *Metamodel for functional architecture of avionics systems*

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 [43] or Matlab Simulink [34].

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.

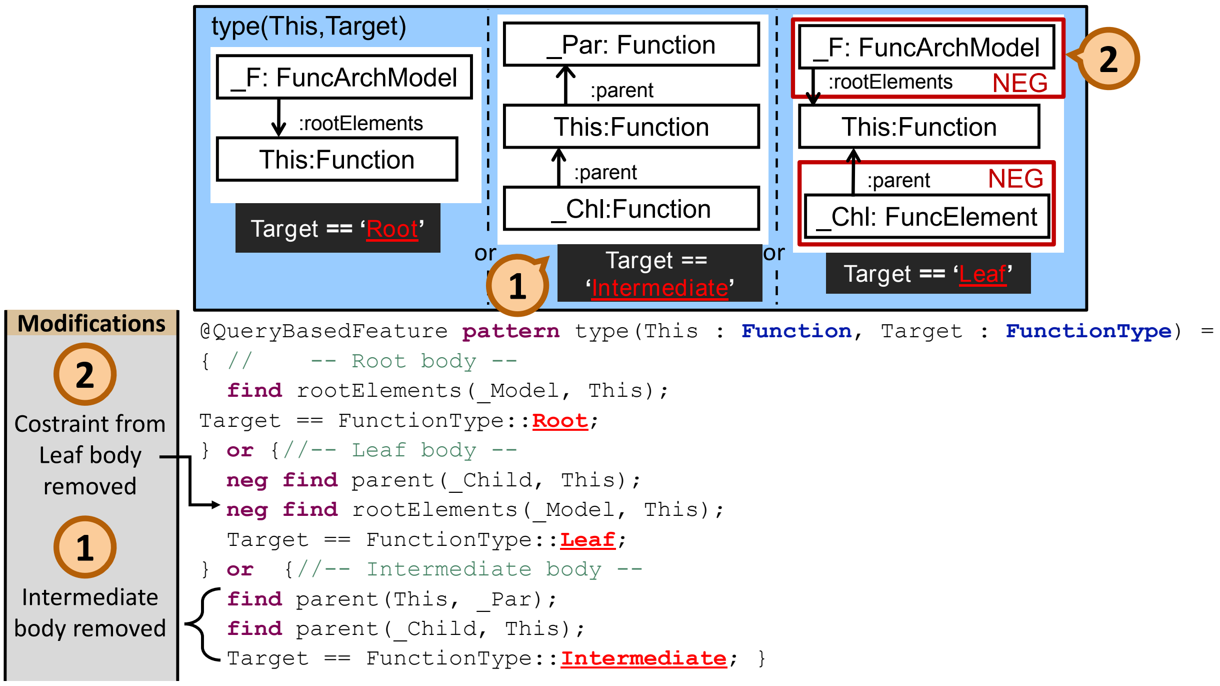
Additionally two derived feature is added to the DSL (highlighted in blue in Figure 5.2):

- 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.



```
@QueryBasedFeature pattern type(This : Function, Target : FunctionType) =
{ //    -- Root body --
  find rootElements(_Model, This);
Target == FunctionType::Root;
} or {//-- Leaf body --
  neg find parent(_Child, This);
  neg find rootElements(_Model, This);
  Target == FunctionType::Leaf;
} or  {//-- Intermediate body --
  find parent(This, _Par);
  find parent(_Child, This);
  Target == FunctionType::Intermediate; }
```

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

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 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:

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*

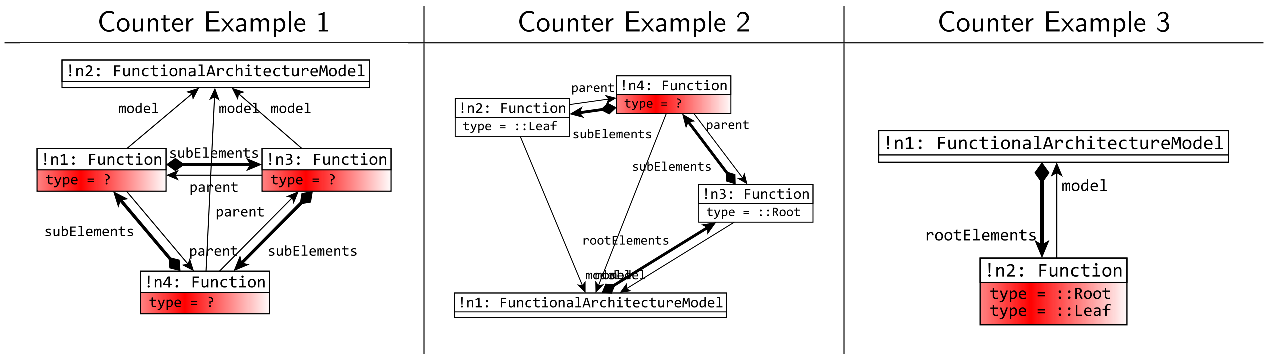


Figure 5.5: *Counter Examples of the* `type` *validation*

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

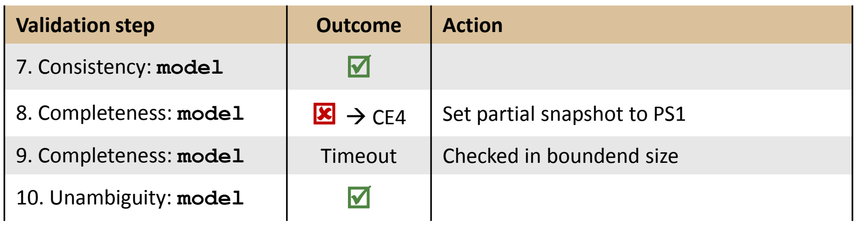
2 The constraint defines that the leaf elements cannot be referred with the rootElements reference is also removed. This will lead to an ambiguity by making the body representing the leaf case more permissive.

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.

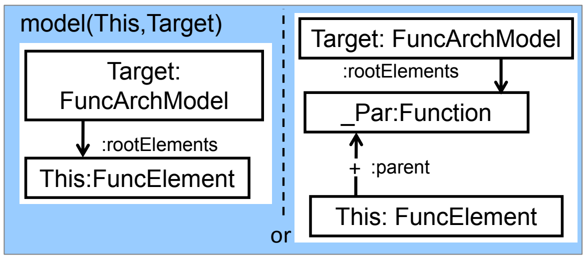
It is visible (and our tool detects it too) that almost every properties of the instance model is correct but the containment hierarchy is unfortunately violated (n1-n3-n4 circle), so the example is invalid. It may happen because the acyclicity of the containment hierarchy can only be approximated in 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.



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*

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.

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

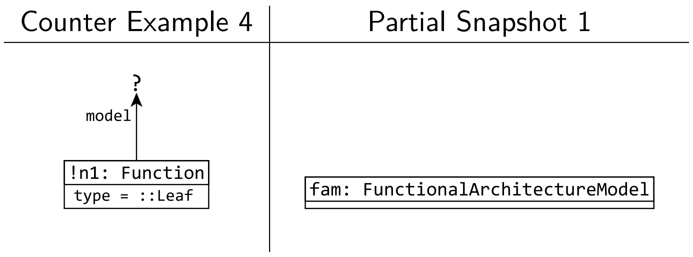
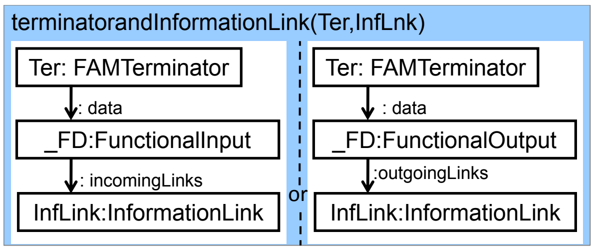
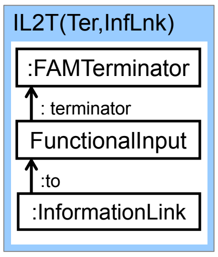


Figure 5.8: *Counter Examlpe and Partial Snapshot of the* model *validation*



```
@Constraint
pattern terminatorAndInformation(
  Ter : FAMTerminator,
  InfLink : InformationLink) =
{
  FAMTerminator.data.incomingLinks(Ter,InfLink);
} or {
  FAMTerminator.data.outgoingLinks(Ter,InfLink);
}
```



```
@Constraint
pattern InformationandTermintator(
  T : FAMTerminator, I : InformationLink) =
{
  InformationLink.to.terminator(I, T);
}
```

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

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.

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.

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*

5.6 Model Generation

The developed tool is able to generate multiple valid example instance models with the selected properties for an arbitrary domain-specific language. Those instance models can be used in several different phases of DSL developement from early validation to automatic test-case generation.

There are many customisable elements in this process that allows the user to specify the requirements of the output model, most importantly the different partial snapshots. Note that if the values of the derived features are set in the partial snapshots than the tool generates models where all of the DFs are evaluated to the predefined value. To extend the expressive power of the input PSes it is also to even add extra derived features to the language to represent custom relations between the instance model elements. Those extra DFs are referred as Predefining Features from now.

The standard process of generating instance models with the general steps is illustrated of the model generation workflow (see in Figure 5.11).

1. The model generation takes a valid DSL as an input. Optionally adding predefining elements to the DSL to define custom relations over the objects of a partial snapshot.
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.

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.

Figure 4.7 and 4.8 already shows examples for completing single or multiple partial snapshots. In the following, an other example where the values of the type derived features are previously defined. At first, a partial snapshot visible in upper part of Figure 5.12 is defined that requires two-two from every type of Function. Next the reasoning tool automatically generates models

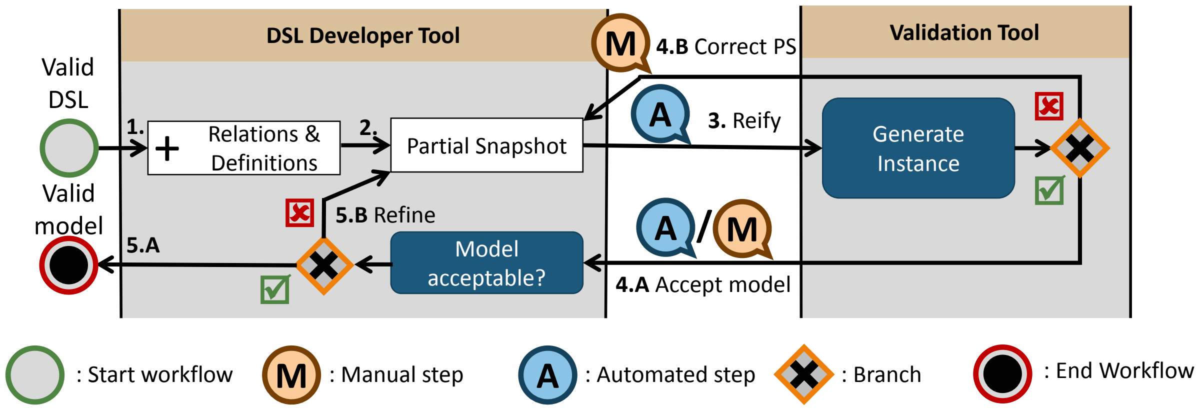


Figure 5.11: *Workflow of the Model Generation*

from the graph pattern definition of the derived features and creates models where the features evaluated to the required values. The bottom of Figure 5.12 shows four different output models from the many possible combinations, where the non-containing references are omitted for the sake of simplicity.

Those instance models then can be used for various purposes, like automatic testing. Model generation can be also used in development time to aid the user with quick-fix like model completions, or show that the edited partial model does not have valid any valid completion.

5.7 Summary

This chapter demonstrates the DSL validation approach described in Chapter 4 on an industrial case-study from the avionics domain. A general workflow for using the validation tool is also proposed. Following the workflow the completeness and the unambiguity check of a derived attribute and a derived reference defined by graph patterns are demonstrated, and subsumability and inconsistency between constraints are explored. The chapter also demonstrates how to tackle harder tasks where the reasoning process fails due to the low level of approximation or insufficient resources.

In the following, Chapter 6 specifies the theory of transformation of the DSL to first order formulae, while Chapter 6 discusses the engineering challenges of implementation.

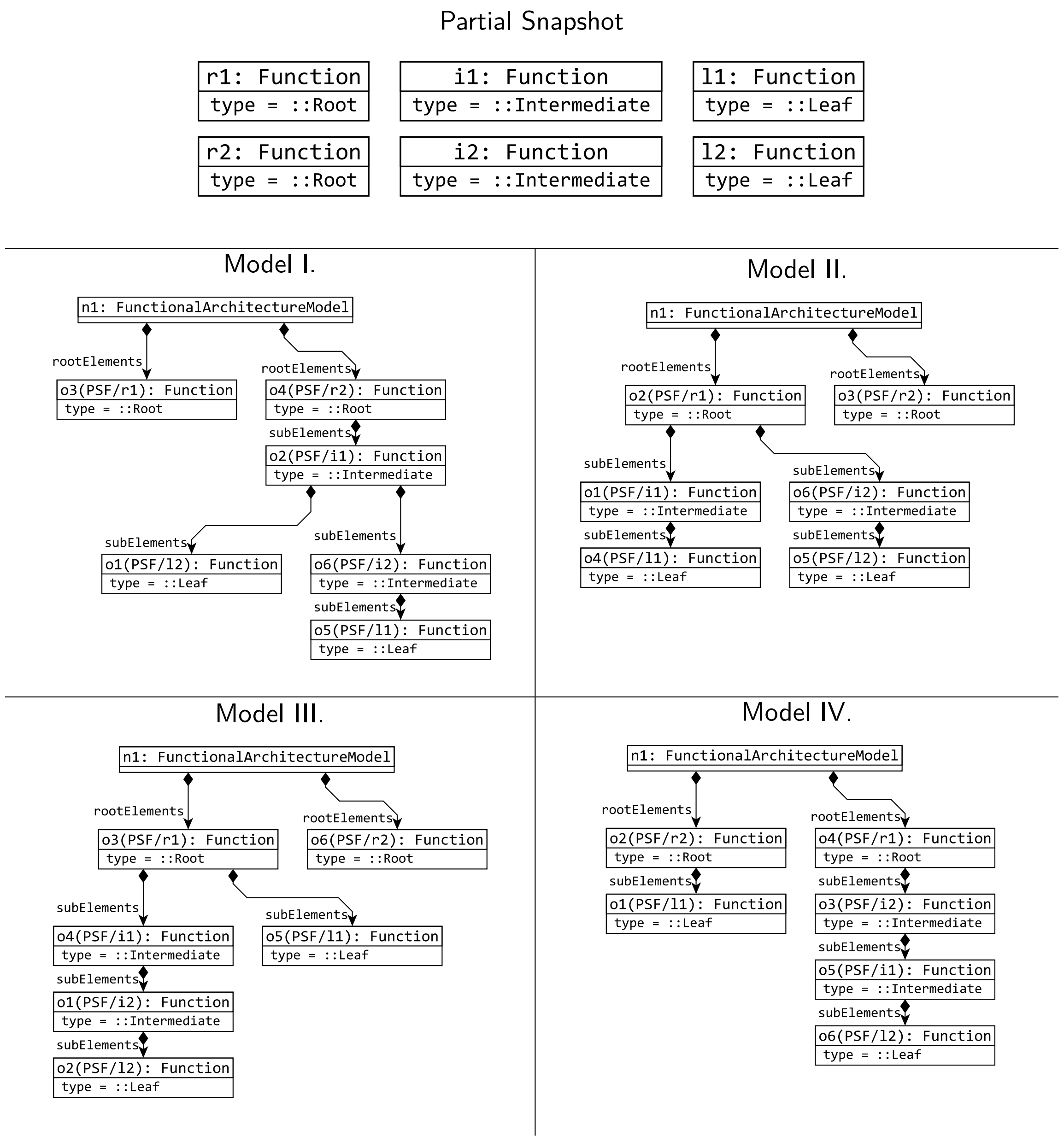


Figure 5.12: *Multiple Model Generation from a partial snapshot using predefining features*

Chapter 6

Mapping DSLs to FOL Formulae

This chapter discusses the transformation process of the DSL artifact to SMT language in details. The metamodel, instance model and the graph patterns are uniformly transformed to first order logic formulae to be able to perform various reasoning tasks to solve the different requirements risen in the motivating scenarios.

6.1 Strategy of the Transformation

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

6.1.1 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:

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

$$\begin{array}{lrcl} 1. & DSL_F \models M_F & \Leftrightarrow & \text{M instance of DSL} \\ 2. & DSL_F \text{ does not have model} & \Leftrightarrow & \text{DSL does not have instance} \end{array}$$

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 6.2).
- The instance model transformation maps the formulae PS_F from the optional partial snapshots (discussed in Section 6.3).

- 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 6.4).

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 6.5. 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.

6.1.2 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 [22]. Our choice of background theorem was the effectively propositional logic (EPR)[40] 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 constants, relations and its formulae in prenex form build from some existentially quantified variables, then some universally quantified variables, then the relations 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. So the strategy is to express most of formulae in the target language, and approximate the inexpressible features.

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. In the following an example for the application of approximation will be introduced.

In the following the approximation of the containment hierarchy will be presented. The tree hierarchy defines that the containment graph satisfies the following properties (as described in details in Section 6.2):

- Every object is contained by an other with the only 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. By doing this the reasoning tasks 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. The negative side effect is that the tool may provide false positives too.

6.2 EMF metamodel transformation

Table 6.2 summarises the transformed features of the metamodel. It also presents which property is expressible in FOL or EPR.

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 6.1: *Expressing Ecore features in Z3*

6.2.1 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 EObjects are uniformly mapped to a newly declared Z3 type: `(declare-sort Object)`. If the number of objects is bounded then the type is defined with the fix range of values. For example instance models with exactly four elements can be defined in the following way:

```
(declare-datatypes () ((Object element1 element2 element3 element4)))
```

where `element1`, `element2`, `element3` and `element4` literals are modelling the objects.

6.2.2 Types

The objects in the EMF instance models are labelled with a type from the classes of the metamodels. To sign that if an object is an instance of a class indicator predicates are introduced, so when an object is an instance of the type then the predicate evaluates to true, and if not it then it evaluates to false.

Every class in the metamodels are transformed to a Z3 predicate, for example, the class Function looks like this:

```
(declare-fun isType!Function (Object) Bool) ; Declared type
```

If `o` is a Function then it is expressed like this: `isType!Function o`

6.2.3 Type hierarchy

In most cases an object is instance of multiple classes because of the generalisation relation between the classes, and several classes signed as abstract or interface which do not have direct instances. Those restrictions are formulated in this subsection. A 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 every possible types and the rows the concrete (not abstract, not interface) ones. The cell represents a literal, which is positive if the type of the row is compatible with the type of column, and negative if it is not.

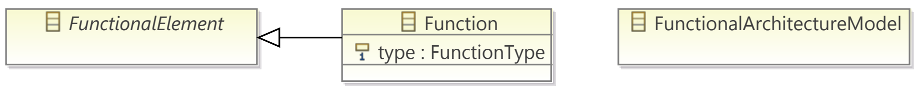


Figure 6.1: *Example type hierarchy*

The transformation of metamodel visible in Figure 6.1 is shown in the next table:

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

	FunctionalElement	Function	FAM	
Function	`(and (isType!FE o)`	`(isType!Function o)`	`(not (isType!FAM o))`	`)`
FAM	`(and (not(isType!FE o))`	`(not (isType!Function o))`	`(isType!FAM o)`	`)`

```
)))
```

6.2.4 References

The references of the metamodels define the edges between the instance objects. Those edges are directed, loops are allowed but parallels not, so relations are suitable to model them. 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.

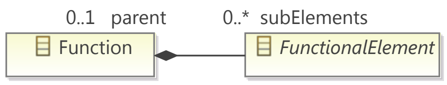


Figure 6.2: *Example references*

For example the definition of the subelements reference on Figure 6.2 is translated to the following reference:

```
(declare-fun Function!subElements (Object Object) Bool)
```

An edge can not got randomly between any objects, the types of the objects on the end of the relations has to be limited to the appropriate type:

```
(assert (forall ((o Object) (t Object)) (=> (Function!subElements o t) (and
  (isType!Function o) (isType!FunctionalElement t)))))
```

6.2.5 Multiplicity

By default, references with 0..* multiplicity are modelled with relations. In other cases creating explicit multiplicity limiting assertions are necessary. In the n..m form the n lower bound means that every object is in relation with n different one, and 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.

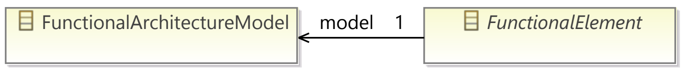


Figure 6.3: *Example 1..1 multiplicity*

For example, the transformation of model reference of the FunctionalElement class visible in Figure 6.3, creates the following lower and upper bound constraints:

```
; Lower bound: 1
(assert (forall ((src Object)) (=> (FunctionalElement!model src)
  (exists ((trg0 Object)) (FunctionalElement!model src trg0)))))
; Upper bound: 1
(assert (forall ((src Object) (trg0 Object) (trg1 Object)) (=>
  (and (FunctionalElement!model src trg0)
       (FunctionalElement!model src trg1))
  (= trg0 trg1))))
```

6.2.6 Inverse edges

It is possible to define inverse relations in EMF, like in case of the parent and subElements references in Figure 6.2 This constraint can be also expressed in first order logic by expressing that if there is a relation form the object o to the target t then there have to be an inverse one from t to o. For example:

```
(assert (forall ((o Object) (t Object)) (=>
  (Function!subElements o t) (FunctionalElement!parent t o)))
(assert (forall ((o Object) (t Object)) (=>
  (FunctionalElement!parent o t) (Function!subElements t o)))
```

6.2.7 Containment

The objects of an EMF model are arranged in a directed tree hierarchy by the containment edges. First the containment relation is defined as the union of the containment-edge relations:

```
(declare-fun contains (Object Object) Bool) ;declaration
(assert (forall ((parent Object)(child Object)) (iff
  (contains parent child)
  (or (Function!subElement parent child)
      ; ... Every containment edge
      ))))
```

The top of the containment hierarchy is called root of the model. Every objects of the root has have exactly one parent with the only exception of the root which does not have any. This can be formulated in the following way:

```
; Declaration of the root object
(declare-fun root () Object)
; The root object does not have parent
(assert (forall ((parent Object)) (not (contains parent root))))
; Every other object has one parent
(assert (forall ((o Object)) (or
  (= o root)
  (exists ((parent Object)) (and
    (not (= parent o))
    (contains  parent o))))))
; Every object has at most one parent
(assert (forall ((child Object) (parent1 Object) (parent2 Object)) (=>
  (and (contains parent1 child) (contains parent2 child))
  (= parent1 parent2))))
```

The tree hierarcy also requires acyclicity which means that any object is unreachable from itself by the path of the containment edges. 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. The following SMT assertion forbids the three length circles.

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

6.2.8 Attributes

The attributes of the metamodel are the properties of the classes. Boolean, integer and real attributes have the appropriate types in Z3, and an enum type can be constructed too. For example, the FunctionType enum visible in Figure 6.4 is representable in the following way:

```
(declare-datatypes () ((enumType!FunctionType
  enumLiteral!FunctionType!Root
  enumLiteral!FunctionType!Intermediate
  enumLiteral!FunctionType!Leaf)))
```

The attributes are transformed in the same way as the relations, the only difference is the range of the parameter defining the value of the attribute (like every odd integer). An example attribute declaration:

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

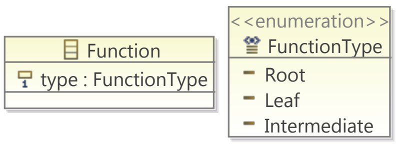


Figure 6.4: *Example Enum attribute*

This representation of attributes fails if the upper bound of the attribute multiplicity is unlimited: it could result objects that have infinite values for an attribute. Currently those cases are not supported.

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

Configuration of the PS	
Positive / Negative	E +
Injective / Shareable	E +
Unrooted / Rooted	E +
Modifiable / Unmodifiable	E +
Multiple Models	E +

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

Table 6.2: *Expressing Ecore features in Z3*

6.3 EMF Instance Model Transformation

The current section defines how instance models or partial snapshots are transformed. The analysis can be parametrized by initial instance models of the metamodel with different configurations. Those initial models can be inserted to the axiom system of the input of the Z3. Generally, the metamodel and the constraints of the language level define universally quantified statements over all the objects. With partial snapshots it is also possible to efficiently configure the validation process with large existentially quantified properties. Table 6.2 summarises the transformed features.

6.3.1 Instance Object

The basic approach of the instance model transformation is to create a statement that expresses the structure of the partial snapshot. The instance objects are the instances of the classes of the metamodel. They are transformed into existentially quantified `Object` variables of a statement which gives structural constraints over the variables. For example, a four element PS is transformed to the following code sequence:

```
(exists ((obj1 Object) (obj2 Object) (obj3 Object) (obj4 Object)) (
  and (
    ; Structural constraints over obj1, ..., obj4
  )))
```

If the PS is configured to be injective a differentiating constraint have to be added to the statement. In the shareable case this is not necessary as multiple variables can be binded to the same object. If the previous four element PS is injective then the constraint is the following:

```
(distinct obj1 obj2 obj3 obj4)
```

After every features of the PS is transformed, the generated statement is added to the axiom system of the DSL to express the occurance or the absence of the structure. Depending on that the PS is configured as positive or negative, the assertion of the generated statement or its negation is inserted to the axioms. In positive cases producing information for back annotation is also necessary, and the binding of the variables should be obtainable. This can be done by replacing the existentially quantified variables to constants in the process of Skolemization. For example in case of the four element PS the result would look be the following:

```
(declare-fun PS!FourElement!obj1 () Object)
(declare-fun PS!FourElement!obj2 () Object)
(declare-fun PS!FourElement!obj3 () Object)
(declare-fun PS!FourElement!obj4 () Object)
(assert (
  ; Structural constraints over PS!FourElement!obj1, ..., PS!FourElement!obj4
  ))
```

The partial snapshots are transformed independently, each of them has to be satisfied separately, and the traceability information is produced for every one of them.

6.3.2 Type

The type of the instance object must also be specified in the statement of the partial snapshot. A speciality of the technique that abstract classes can be instantiated in the PSes, and transformed to a structural constraint. For example, if the type of `InstanceObject!o1` is FunctionalElement it would be transformed to the following constraint:

```
(isType!FunctionalElement obj1)
```

If the partial snapshot is unmodifiable it also states that an instance object can not have less general type than the one in the PS.

6.3.3 References and Attributes

The references between the instance objects can be defined by simply stating that the pair of the source and the target is in the relation that models the reference. Instance attributes are defined in the same way. For example in case of the subElements reference:

```
(Function!subElements obj1 obj2)
```

If the PS is unmodifiable then the absence of a reference is also stated by a negated form. The containment relation of the objects is defined too by the references. If the Ps is Rooted then the root of the partial snapshot should be equal to the root of the logic model, for example if the root of the PS is the `o1` individual then this constraint looks like this: `(= root o1)`.

6.3.4 Multiple Model Generation

The proposed transformation method is also capable of generating multiple results for a validation task by tracing back previous logic models and axiomatising the prohibition of model isomorphism. At first let us define the concept of model isomorphism:

Definition 7 (Model Isomorphism) *A* ***model morphism*** *is a $m : Obj_1 \mapsto Obj_2$ function where the Obj_1 and Obj_2 are sets of two instance models, and it satisfies the following conditions:*

- *if o instance of T then $m(o)$ is also an instance of T*
- *if o_1 has an R reference to o_2 then $m(o_1)$ has an R reference to $m(o_2)$*
- *if o has an A attribute value v then $m(o)$ has an A attribute value v*

Two models with Obj_1 and Obj_2 objects are isomorphic if there the following bijective model morphisms exist:

$$m_1 : Obj_1 \mapsto Obj_2, \qquad m_2 : Obj_2 \mapsto Obj_1$$

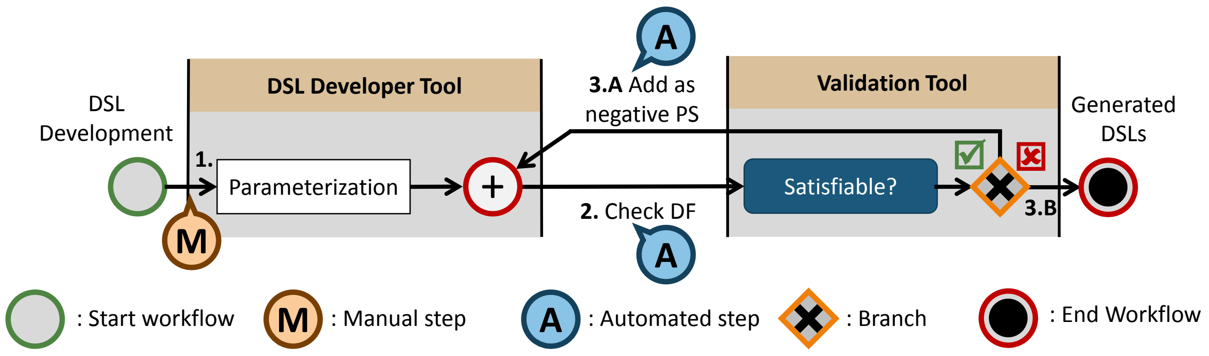


Figure 6.5: *The automated process of generating multiple results for a validation task*

The $m : Obj_1 \mapsto Obj_2$ function is bijective if

- Injective (if $o_1 \neq o_2$ then $m(o_1) \neq m(o_2)$)
- Surjective (there is an o_1 for every $o_2 \in Obj_2$ where $m(o_1) = o_2$)

The concept of morphism can be translated to the language of partial snapshots: if there is a model morphism then the source is contained as a shareable partial snapshot. A bijective model morphism is an injective partial snapshot where the size of the result model is equal to the source model. If there would be a bijective morphism $m_1 : Obj_1 \mapsto Obj_2$ but not the inverse $m_2 : Obj_2 \mapsto Obj_1$ then it could be detected with an unmodifiable PS. So stating that the DSL contains an arbitrary instance model:

- The logic model has the same number of object than the instance model.
- The logic model does not contains the instance model as a positive, injective and unmodifiable partial snapshot.

Therefore by the negation of this statement the reasoning process can be configured to generate a different (none isomorphic) model than the selected one.

During the generation of multiple results the reasoning process works as shown in Figure 6.5:

1. The reasoning tool is parametrized the same way as in case of generating a single results.
2. The reasoning tool process the input and the validation task.

3.A If the reasoning process results in a logic model it is traced back to the parametrization to generate different model from the actual one.

3.B If the satisfiability check fails, or enough models are generated then the process stops with all generated models as its results.

6.4 EMF-IncQuery Graph Pattern Transformation

This section describes how EMF-IncQuery patterns can be transformed to first order formulae. Table 6.3 shows which feature can be translated to SMT and EPR whether they used as wellformedness constraints or derived features.

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

6.4.1 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 the disjunction of 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.

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

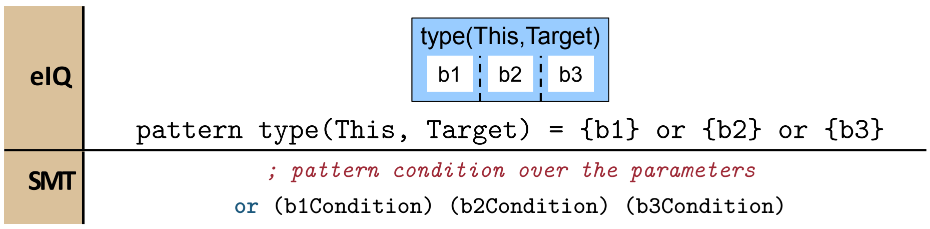
For example, let us take a two parameter pattern called type with the parameter list This: Function and Target: FunctionType. The matches of this pattern are defined by the following predicate:

```
(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.



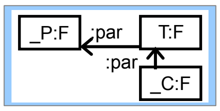
```
pattern type(This, Target) = {b1} or {b2} or {b3}
```

```
; pattern condition over the parameters
or (b1Condition) (b2Condition) (b3Condition)
```

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:

eIQ



```
C1:  FunctionalElement.parent(T, _P);
C2:  FunctionalElement.parent(_C, T);
C3:  Function(T);
C4:  Function(_C);
C5:  Function(_P);
```

SMT

```
; pattern body condition over the parameters and inner variables
exists ((_P Object) (_C Object)) (
and (C1) (C2) (C3) (C4) (C5))
```

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.

6.4.2 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:

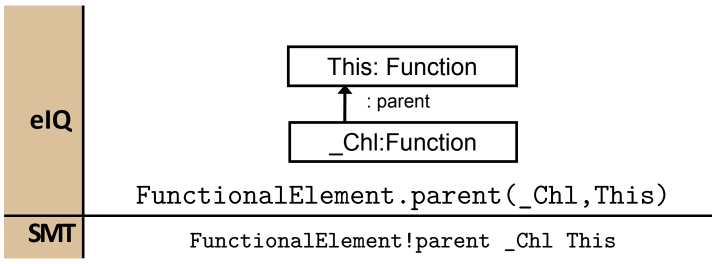
eIQ

```
Function(This);
```

SMT

```
isType!Function This
```

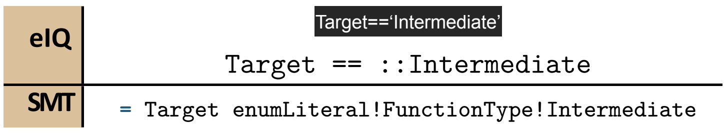
Path constrains in IncQuery defines that there is a path consists of sequence of references from the defined type that leads from a variable to another. By introducing the implicit object variables as the inner nodes of the path, the expression can be compiled into simple reference requirements. For 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.



```
FunctionalElement.parent(_Chl,This)
```

```
FunctionalElement!parent _Chl This
```

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



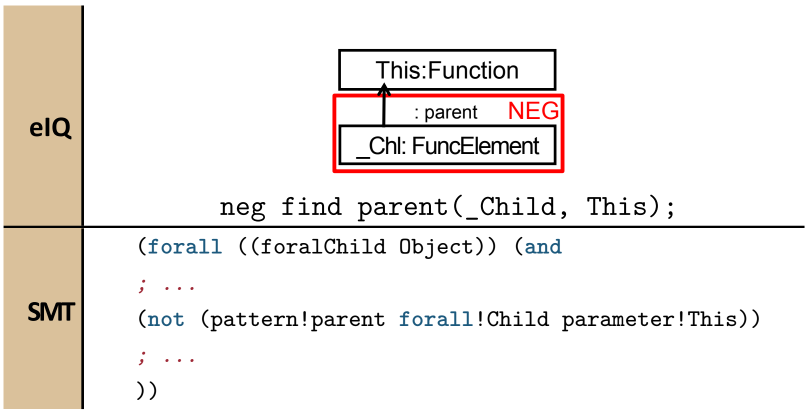
```
Target == ::Intermediate
```

```
= Target enumLiteral!FunctionType!Intermediate
```

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:



```
neg find parent(_Child, This);
```

```
(forall ((foralChild Object)) (and
; ...
(not (pattern!parent forall!Child parameter!This))
; ...
))
```

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 transitive closure approximation and other recursive pattern call transformation is available in [39].

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: $parent0Match(This, P, d1, d2) \Rightarrow parent(This, P) \vee \exists m3(m3 \neq d1, m3 \neq d2) : parent(This, m3) \wedge \textbf{true}$

6.4.3 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))))
  ```

6.5 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:

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

- 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)))))
```

6.6 Summary

This chapter details the transformation method of the proposed reasoning process. First the main concept of the logic formalism is described, then the mapping of every aspect of the domain specific language is given. The independedly mapped metamodel, constraints, derived features and instance models are uniformly added axiom system which satisfiability is checked in order to execute the selected validation task.

The different validation problems introduced in Chapter 4 can be solved with the described method to validate complex domain-specific languages like the one presented in Chapter 5.

Chapter 7

Implementation

In this chapter the most relevant engineering challenges and implementation decisions are discussed. It also and presents performance measurements of the research prototype.

7.1 Development Details

The validation tool is fully integrated to the Eclipse DSL developer tool, and can be used immediately on the developed DSL in design time. A very important advantage of the proposed validation tool is that it can be operated by a DSL developer, and does not require any theorem proving knowledge from the user. It can be directly configured by DSL artifacts.

7.1.1 Architecture

A high-level architecture overview of the tool is depicted in Figure 7.1 where the reasoning process is divided into four separate sequential component:

- Input Parametrization: The tool collects the domain-specific language artifacts and the reasoner customisations according to the a configuration file. The fully defined validation task is passed to the next component. (See in Subsection 7.1.2)
- Modular Transformation: The different DSL artifacts are forwarded to the appropriate transformation module, which transforms the validation task to an abstract consistency checking problem. The generated formula set is passed to the next component. (See in Subsection 7.1.3)

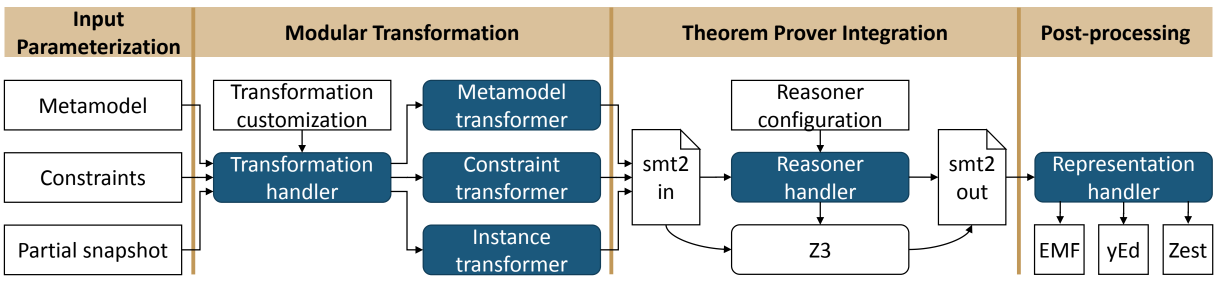


Figure 7.1: *Architecture of the tool*

```
 1 ModelGenerationWithMultiplePS () {
 2   language {
 3     metamodel = "http://hu.bme.mit.transima/fam/1.0";
 4     constraint type = eIQ pattern
 5       "hu.bme.mit.transima.models.fam.derived.type";
 6     constraint rootElements = eIQ pattern
 7       "hu.bme.mit.transima.models.fam.derived.rootElements";
 8     constraint parent = eIQ pattern
 9       "hu.bme.mit.transima.models.fam.derived.parent";
10     constraint terminatorAndInformation = eIQ pattern
11       "hu.bme.mit.transima.models.fam.derived.terminatorAndInformation"; }
12   customisation {
13     model size = 7;
14     rooted partial snapshot = "FamModelConsistencyCheck/PS1.partialSnapshot";
15     shareable partial snapshot = "FamModelConsistencyCheck/PS2.partialSnapshot";
16     negative partial snapshot = "FamModelConsistencyCheck/neg.partialSnapshot";}
17   configuration {
18     path = "D:\\Programs\\z3-4.3.0-x64\\bin\\z3.exe";
19     time = 120;
20     strategy = (check-sat-using (and-then
21       qe (using-params smt :qi-eager-threshold 0 :qi-lazy-threshold 0))); }
22   validation {
23     task = model generation;
24     model count = 20; }
25 }
```

Figure 7.2: *Example validation configuration*

- Theorem Prover Integration: The input axiom system is transformed to the concrete syntax of the target reasoning platform, which in our case is the SMT2 input language of the Z3 SMT solver. The framework calls the theorem prover with the specified configuration and processes the result of the reasoning. The output of the reasoner is compiled to a logic model and passed to the final component. (See in Subsection 7.1.4)
- Post-processing: In this phase, a completed partial snapshot is built by the results of different queries over the the logic model and interpreted in the context of the validation task. At the end of the validation process the result partial snapshot can be presented to the user in multiple way, for example as an object diagram-like graph, or as a standard instance model with the original concrete syntax of the language. (See in Subsection 7.1.5)

The next subsections details those components and the achieved engineering results.

7.1.2 Input Parametrisation

The selected validation task can be parametrized by a simple configuration file as shown on Figure 7.2. Using a configuration file has the benefit that the validation setup is easily editable, modifiable, and persistable. A validation project consists of four separate section:

- Language Reference: The user selects the target metamodels (like Line 3 in the configuration file) and refers to the graph patterns which are included to the validation process (from Line 4 to 11).

- Customisation: The user may give some restrictive search parameter to the reasoning process such as model size (like in Line 13) limitation or different partial snapshots (from Line 14 to 16).
- Reasoner Configuration: The user selects the target SMT solver which is used for the reasoning (in Line 18). The user may set the time and memory limit to the actual validation task (in Line 19), and optionally define a reasoning strategy which is believed to be more efficient for the target task (in Line 20).
- Task Configuration: The user defines the validation task (Line 23) and the number of the required models or counter examples for the task (Line 24).

A configuration file may consist of a sequence of validation projects which allow the user to a define a batch of multiple DSL validation.

The validation process can be parametrised by multiple partial snapshots which ones can be specified in a more general formalism than the standard instance model (see Section 4.4.1 for more details). Advanced tooling was developed to support the definition of the various PSes. At first, in the validation configuration standard instance models can be referred as a PS, so if the extra options of the PS are not used the formalism of the PS can be completely ignored.

Secondly, a partial snapshot can be automatically derived from an instance model or transformed back to it. And finally, a reflective partial snapshot editor is created, which highly resembles to the default tree editor of the EMF framework, but implements the extra services. Figure 7.3 shows a screenshot of the editor, where an abstract class is instantiated. PSes can be created or modified in this editor.

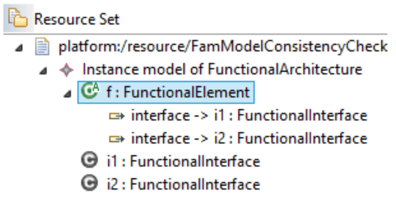

Resource Set
platform:/resource/FamModelConsistencyCheck
Instance model of FunctionalArchitecture
f : FunctionalElement
interface -> i1 : FunctionalInterface
interface -> i2 : FunctionalInterface
i1 : FunctionalInterface
i2 : FunctionalInterface

Figure 7.3: *Editor of the partial snapshot*

7.1.3 Modular Transformation

The collected DSL artifacts are forwarded to the adequate transformation module to create the input axiom system for the theorem proving. The transformation modules, which are implemented as Xtend [49] templates, execute Model-to-Model transformations to create and iteratively extend the software model[1] of the logic axiom system. This software model represents general logic expressions (like logic operators, function declarations, assertions), and easily modifiable by the transformation modules, for example, it is easy to add an extra operand to a conjunction. The axiom s model of the axiom system is passed to the next component.

If the validation process results with a logic model then the result is also represented in this formalism by interpreting the declared functions. The resolution of the created interpretation is not a trivial task because the logic structure is defined through several indirections. Every function definition can contain other function definitions, branches, logical or numeric values and or constant reference. These structure has to be resolved recursively in order to evaluate a value.

[1] For the sake of clarity the two usage of the term 'model' is distinguished: software model means a instance model of the EMF, and logic model means the result logic structure of a satisfiable axiom system.

In Figure 7.4 an example is shown. The value of the `isType!Event(e1)` is required. The steps of the resolution are the following:

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 `k!552` the function `isTypeEvent!556` is parametrised, and resolved. The result of the function composition is also the result of the `isType!Event(e1)`.

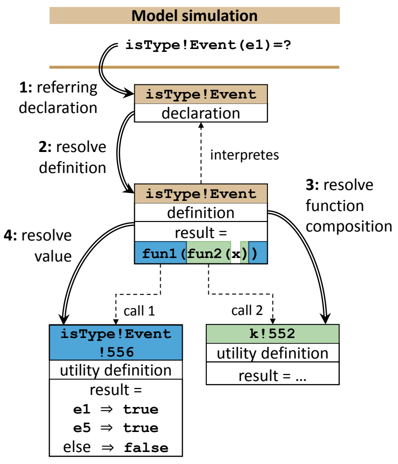


Figure 7.4: *Example of the structure of the SMT output*

An API is created to support the creation of the axiom system and create queries to the logic structures.

7.1.4 Integrated Use of a Theorem Prover

The generated model of the axiom system is passed to the reasoner handler. First, a reasoner specific textual representation is created from the generic logic model of the axiom system. To handle this problem a new XText[50] generative grammar is created to support the serialisation of the generic axiom system model to the SMT language of the Z3 reasoning platform. Using the defined grammar the input and output SMT files can be easily parsed or serialised.

The reasoner handler serialise the axiom system and forward it to the configured reasoning tool whit the proper parametrization. After the successful execution its output is processed and parsed using the created SMT grammar. The reasoner handler interprets this output as a logic structure and builds its corresponding partial snapshot. Then the prepared partial snapshot is passed to the next phase.

Both the input and output of the theorem prover is saved as a document in order to use the resolution in the future.

7.1.5 Post-processing Phase

The output partial snapshot represent an example or a counter example. Rich and extensible set of different presentation methods are implemented to ease the understanding of the result of the validation.

- Different visualisation techniques are applied to show an object diagram-like graphical representation of a partial snapshots. At first, a graph file is created for every instance models, which file is editable with the yFiles yEd[57] tool. Additionally, a graph is presented in the editor using the Zest[51] technology.

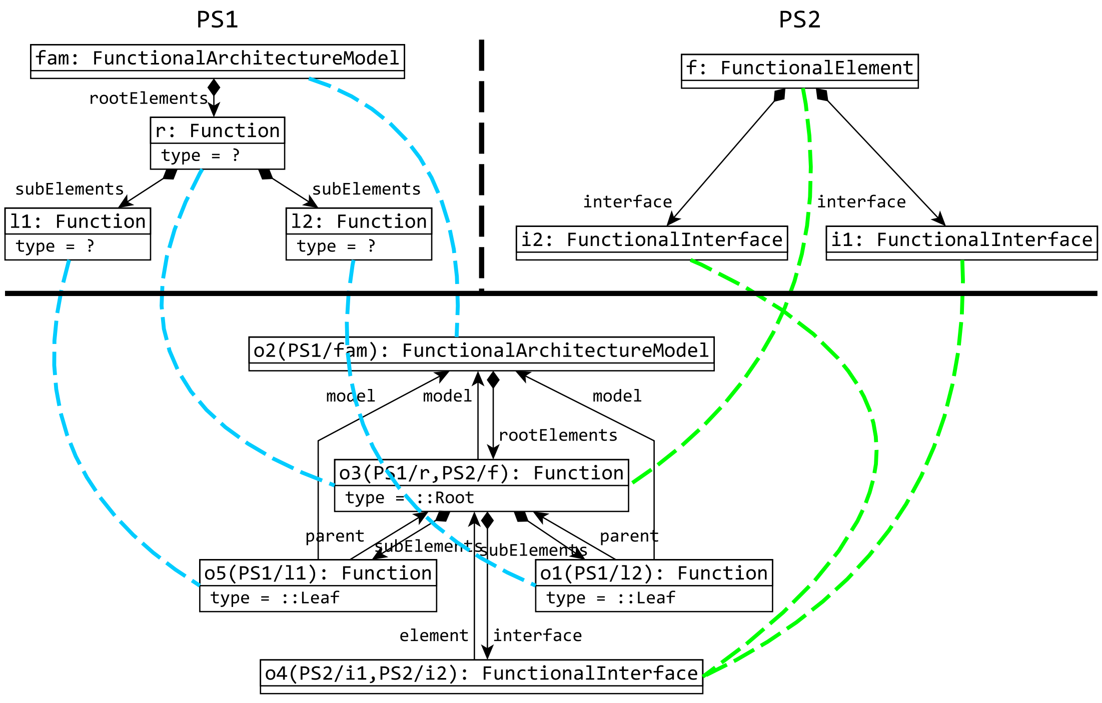


Figure 7.5: *Traceability example*

- If the result model is a valid instance model then the partial snapshot is transformed to a standard instance model of the domain specific language. This can useful when the validation tool is used in a tool chain. This can be also very convenient to the user, because the instance model can be examined in its native editor.
- Finally, a csv file is generated to summarise the results of multiple validation task executed in batch mode. The file also highlights the statistic values of the theorem proving.

The traceability is the ability to follow the lifecycle of elements, models and requirements in both a backward and forward direction. The traceability of the partial snapshots during the model generation is completely supported. The tool identifies to the instance objects with a unique ID, automatically. The instance objects of the completed partial snapshots are associated with their ancestors by referring their identifiers in ID(ancestors) form. Figure 7.5 shows two initial partial snapshot and their completed counterparts. The compliance of the objects are symbolized dashed lines, and the relation with the different partial snapshots are presented with different colored lines.

The following list shows some examples which elements that can be highlighted:

- Filled Attributes: It is possible to define instance objects with undefined attributes which are filled in the model generation process. For example, in Figure 7.5 the PS1/l1 named Function has an unfilled type attribute which is defined to be a Leaf in the object o5. Those new attributes might be marked for the user.
- Newly Created References: The partial snapshots might be completed with new references, like in the example, where all non-containment edges are added by the validation tool.

- New Objects: New objects might be added to the partial snapshots during the creation of the instance models. In some cases it would be convenient to mark those elements.
- Refined Objects: The type of an abstract instance object is refined to a concrete one, for example, PS2/f is refined from FunctionalElement to Function.
- Counter Example Occurrence: In most of the various validation tasks the goal is to prove that the language is free from a certain antipattern, like the occurrence of a failed derived feature. In the counter examples the match of this pattern would be highlighted.

7.2 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 visualization.

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 the problem is undecidable in general.

Nonetheless, some performance measurements made to approximate resource requirements of the validation process. It is concluded that the main limitation is the runtime, which depend on two factor:

1. The size of the required models
2. The size (complexity) of the axiom system

The performance measurements aim to check the effect of those two factor on the runtime. The measurements executed on iteratively harder model generation tasks for the Functional Architecture Model. The DSL is discussed in Chapter 5, the metamodel visible on 5.2, which is enriched with two derived features: the model reference which definition is located in Figure 5.7 and the type attribute which is defined by the pattern visible in Figure 5.3. Additionally, a well-formedness constraint is added to the DSL which is defined in the top of Figure 5.10. Finally, the model generation task is configured to satisfy three partial snapshots visible on Figure 7.6 and previously mentioned in Subsection 4.4.1. The goal of this choice is to cover most of the proposed functions of the validation tool on a DSL used in the industry.

The measurements are executed on an average personal computer.

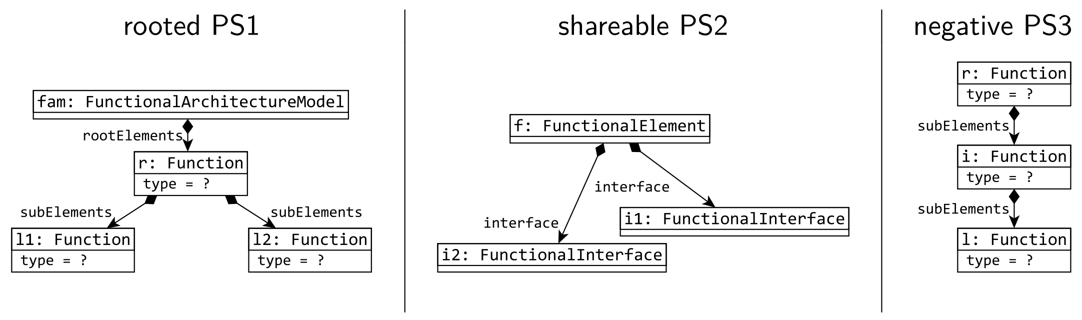


Figure 7.6: *Partial Snapshots used in the measurements*

7.2.1 Model Size

Our experiences clearly shows that executing validation tasks on higher model size is generally a much harder reasoning problem. To measure the effect of the size on the runtime, models with object count up to 30 has been generated. From 1 to 30 each measurement has been repeated five times.

The results are summarized in Figure 7.7. Each measurement has been marked with a dot on the diagram, and the average of the runtimes is illustrated with blue.

In general, the runtime grows exponentially with the model size, but it remains acceptable up to 30 elements. The trend is illustrated with red dashed line. Furthermore it can be concluded that the deviance of the runtimes also grows. Sometimes bigger models are generated with low runtime, but in those cases the results contain many symmetries.

7.2.2 Axiom System Size

The complexity of the reasoning task can be estimated with the size of the axiom system. As Section 6.3.4 describes, during multiple model generation the previous answers are added back to the reasoning process while incrementing the axiom system with antipatterns. So generating more and more different models will evenly increase the size of the axiom system with partial snapshot specifications. Therefore checking the runtime of multiple model generation is both a valid and practical way to measure effect of the size of axiom systems. 30 different model were generated with the size of seven, which is repeated twelve times.

The measured runtimes are in Figure 7.8. The measurements have been marked with a dots on the diagram, and the average of the runtimes is illustrated with blue line.

In general, the runtime is trending polynomially with lower deviation from the average in the function of model count. This allows the user to generate new models in an acceptable time. Interestingly, the runtimes on new models are uniformly fluctuating. The reason of this phenomenon might be that the previous models are often the the same in each test run, so generating a new model is consequently a more or less complex task.

7.3 Conclusion

The proposed validation tool is proved to be applicable in two industrial case studies. These validation tasks can be characterised by the following features:

- Checking language properties with relatively small counterexamples (1-15 elements).
- Completing more complex (30+ elements) models that are almost completed

The common property of these two case studies is that despite their reasoning task can be challenging, degrees of freedom (e.g., unfilled attributes, new objects, extra relations) is relatively low. Based on our experiences, the proposed validation tool is applicable in those reasoning tasks.

However, in cases where generating a lot of objects is needed, the validation tool usually perform poorly. For example, creating large instance models for a simple DSL might be a trivial task, but the tool likely to be insufficient.

Despite the theoretical challenges and the high complexity of the problems the practicality of the proposed method is proved on industrial the case studies.

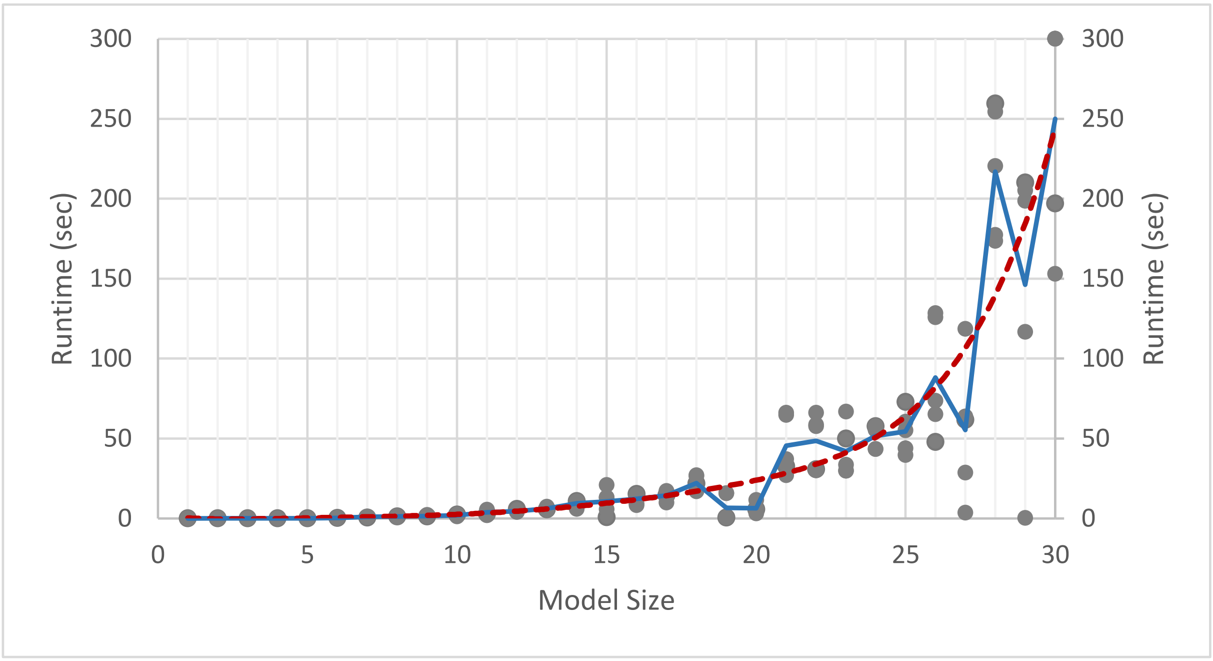


Figure 7.7: *Runtime measurements of model generation with increasing model size*

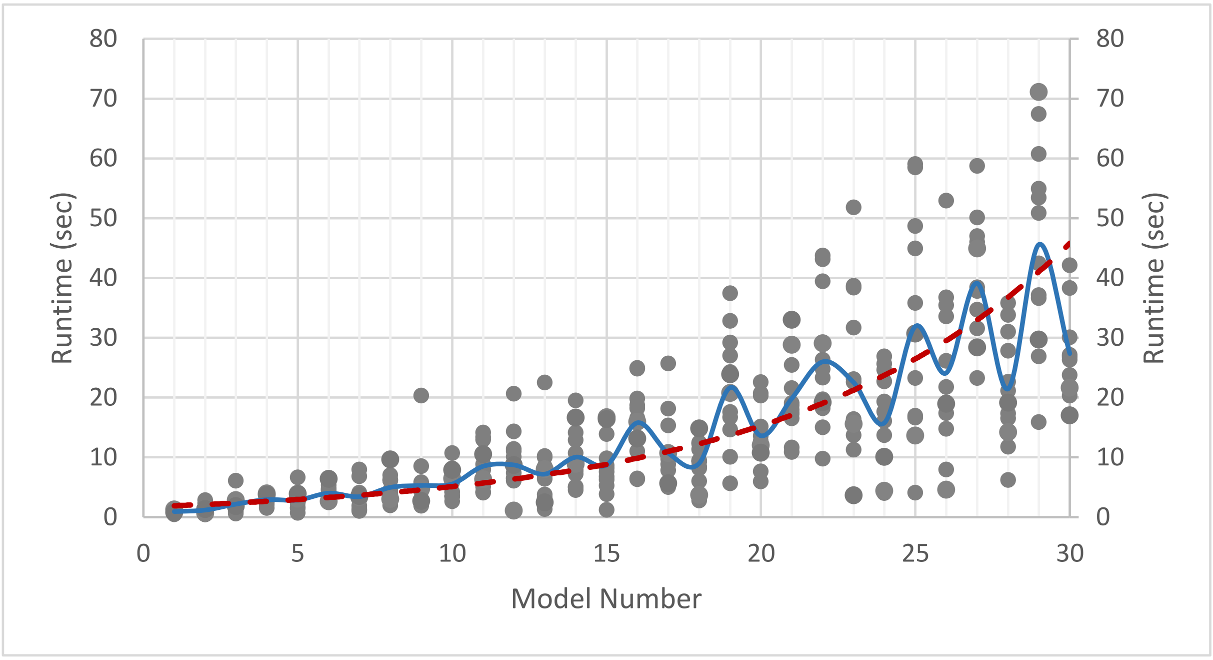


Figure 7.8: *Runtime measurements of multiple model generation*

Chapter 8

Conclusions and Future Work

8.1 Results

The main scientific results presented in the thesis is summarized in the following points:

- This report summarise a new **DSL validation approach** where a logic based validation method is proposed.
- This approach was **applied on EMF** formalism and can be generalised to other meta-modeling techniques (like Meta-Object Facility (MOF)[38]).
- The method is able to **validate different properties** of the language,
- or **generate multiple instance models** for an arbitrary DSL.
- In the case of failed validation a **counterexample** is generated to show the reason of the inconsistency.
- In addition to EMF metamodel the **mapping the EMF-IncQuery graph patterns** to first order logic formulae is presented.
- **Instance models** can be added to the validation process to require or forbid certain structures in the analysed language.
- 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.
- I 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 I developed the **back annotation** technology to support valid instance model generation.

The achieved engineering results in the implementation of the application:

- The 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.
- I 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 addition, the 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.)

The approach previously presented with the following successes:

- The mapping method has been successfully demonstrated during the Trans-IMA industrial project in the avionics domain. Part of this contribution was published[44] 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.
- A report about the approach is also presented[23] in Scientific Students' Associations Conference 2013 (Tudományos Diákkör 2013) where it won first prize in software section.

8.2 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[54].

Finally, to investigate the applicability of my approach in context with the new DO-178C certification standard [37] 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] Florian Lapschies. *SONOLAR*. `http://www.informatik.uni-bremen.de/~florian/sonolar/`.

[22] 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.

[23] Ágnes Barta and Oszkár Semeráth. Consistency analysis of domain-specific languages. Technical report, Budapest University of Technology and Economics, 2013.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

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

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

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

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

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

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

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

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.

[46] 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.

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

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

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

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

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

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

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

[54] 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.

[55] 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.

[56] 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).

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