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

Formal Validation and Model Generation for Domain-Specific Languages by Logic Solvers

Ph.D. Dissertation

Oszkár Semeráth

Thesis supervisor:
Prof. Dániel Varró

Budapest, 2019

Oszkár Semeráth
https://inf.mit.bme.hu/en/members/semeratho

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

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

H-1117 Budapest, Magyar tudósok körútja 2.
March 2019

http://hdl.handle.net/10890/13134

Declaration of own work and references

I, Oszkár Semeráth, hereby declare that this dissertation, and all results claimed therein are my own work, and rely solely on the references given. All segments taken word-by-word, or in the same meaning from others have been clearly marked as citations and included in the references.

Nyilatkozat önálló munkáról, hivatkozások átvételéről

Alulírott Semeráth Oszkár kijelentem, hogy ezt a doktori értekezést magam készítettem és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2019. 03. 26.

Semeráth Oszkár

Acknowledgments

First and foremost, I would like to express my gratitude to my supervisor, Prof. Dániel Varró. He provided me with continuous guidance, insight and his many visionary advice over the past ten years, through my undergraduate and master studies to my Ph.D.

I am thankful to all my former and present colleagues in the Fault Tolerant Systems Research Group, especially Prof. András Pataricza, Dr. István Majzik and Dr. Zoltán Micskei for their hard work for guiding the research the group. I would like to thank all of my co-authors and collages I worked with: Ágnes Barta, Márton Búr, Gábor Bergmann, Csaba Debreceni, Rebeka Farkas, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, Kristóf Marussy, András Szabolcs Nagy, István Ráth, Gábor Szárnyas, Zoltán Szatmári, Zoltán Ujhelyi, András Vörös, and everybody worked in my research group. I was very fortunate that I could work on my research in cooperation with many talented students: Mária Bekő, Csaba Hajdu, Benedek Horváth, Raimund-Andreas Konnerth, Ádám Lengyel, Krisztián Mayer, Anna Monory, Dénes Terebesi, Alexandra Anna Sólyom and many others from my research group, and Aren Babikian, Anqi Li, Sebastian Pilarski from McGill University.

I would like to thank all the financial support I got for my research. My research was partially conducted in MTA-BME Lendület Cyber-Physical Systems Research Group. Additionally, my research was supported by research projects:

- CERTIMOT (Design and Analysis Techniques for Certifiable Model Transformations, ERC_HU-09-01-2010-0003),
- CONCERTO (Guaranteed Component Assembly with Round Trip Analysis for Energy Efficient High-integrity Multi-Core Systems, ART-2012-333053),
- MONDO (Scalable Modelling and Model Management on the Cloud, EU ICT-611125),
- R3-COP (Resilient Reasoning Robotic Co-operating Systems, ART-100233),
- and collaborative projects with Embraer and Ericsson Hungary.

Moreover, my research was partially supported by scholarships UNKP-17-3-III New National Excellence Program of the Ministry of Human Capacities, and Schnell László Foundation. My publishing was partially supported by the SRC program of Microsoft Research. I am thankful for the Student Research Trainee program of McGill University.

Finally but most importantly, I would like to thank the support of my parents, my sister and my grandmother for their help, support and encouragement.

Köszönetnyilvánítás

Mindenekelőtt szeretnék köszönetet mondani konzulensemnek, Prof. Varró Dánielnek, aki iránymutatásával, szakértelmével és értékes, előremutató tanácsaival támogatott az elmúlt tíz évben, az egyetemi tanulmányaim kezdetétől a doktori végéig.

Hálás vagyok továbbá minden korábbi és jelenlegi munkatársamnak a Hibatűrő Rendszerek Kutatócsoportból és a Méréstechnika és Információs Rendszerek Tanszékről. Külön szeretnék köszönetet mondani Prof. Pataricza Andrásnak, Dr. Majzik Istvánnak és Dr. Micskei Zoltánnak a csoport vezetéséért végzett áldozatos munkájukért. Szeretném továbbá megköszönni az összes társszerzőmnek és munkatársamnak a közös munkát: Barta Ágnesnek, Búr Mártonnak, Bergmann Gábornak, Debreceni Csabának, Farkas Rebekának, Hegedüs Ábelnek, Horváth Ákosnak, Izsó Benedeknek, Marussy Kristófnak, Nagy András Szabolcsnak, Ráth Istvánnak, Szárnyas Gábornak, Szatmári Zoltánnak, Ujhelyi Zoltánnak, Vörös Andrásnak és mindenki másnak a csoportomból. Volt szerencsém továbbá számos rendkívül tehetséges hallgatókkal is együtt dolgoznom: Bekő Máriával, Hajdu Csabával, Horváth Benedekkel Konnerth Raimund-Andreasszal, Lengyel Ádámmal, Mayer Krisztiánnal, Monory Annával, Terebesi Dénessel, Sólyom Alexandrával, és még sok mással a tanszékünkről, valamint Aren Babikiannal, Li Anqival és Sebastian Pilarskival a kanadai McGill Egyetemről.

Szeretném köszönetet mondani a munkámat támogató kutatási projekteknek. Munkámat részben a MTA-BME Lendület Kiberfizikai Rendszerek Kutatócsoportban végeztem. Ezen kívül több kutatási projekt is támogatta a munkámat:

- CERTIMOT (Design and Analysis Techniques for Certifiable Model Transformations, ERC_HU-09-01-2010-0003),
- CONCERTO (Guaranteed Component Assembly with Round Trip Analysis for Energy Efficient High-integrity Multi-Core Systems, ART-2012-333053),
- MONDO (Scalable Modelling and Model Management on the Cloud, EU ICT-611125),
- R3-COP (Resilient Reasoning Robotic Co-operating Systems, ART-100233),
- valamint Embraer repülőgépgyártó vállalattal and Ericsson Hungaryval közös projektek.

Ezen kívül kutatásomat támogatta az Emberi Erőforrások Minisztériuma UNKP-17-3-III kódszámú Új Nemzeti Kiválóság Programja és a Schnell László Alapítvány. Publikációimat támogatta a Microsoft Research SRC programja. Hálás vagyok továbbá a kanadai McGill Egyetem kutatói gyakornoki programjának.

Végül, de nem utolsó sorban, szeretném megköszönni családom, szüleim, testvérem és nagymamám kitartó támogatását és bátorítását.

Summary

Graph-based models are widely used in the development of complex, safety and business critical systems like automotive, avionics and financial software. Advanced modeling environments based on Domain Specific Languages (DSLs) (1) supports the development of models by continuously validating them, (2) derives different views to highlight task-specific aspects of the model, (3) automates several steps in the development (e.g. by code generators) and (4) provides mathematical analysis to check the correctness of the underlying design (e.g. by model checkers). Thus they can significantly improve the overall productivity of the development and quality of the product.

Despite the wide range of existing tool support, constructing a complex DSL and a modeling environment is a tedious task, and – like any software artifacts – they are error-prone. First, to define the structure of graph models of a language, a DSL is defined with a large number of complex, interacting constraints (i.e. design rules), that can easily be formalized incorrectly. This could result in inconsistent, incomplete or ambiguous languages. Moreover, errors in the implementation of the modeling environment inject errors to the generated code and invalidate the results of any verification process. Therefore, it is important to ensure the correctness of modeling tools themselves. As model-driven tools are frequently used in critical systems design, those tools should be validated with the same level of scrutiny as the underlying system as part of a software tool qualification process in order to provide trust in their output. Therefore software tool qualification raises several challenges for building trusted DSL tools for a specific domain.

The objective of this thesis is to improve validation and testing support for DSLs and modeling tools using automated model generation techniques. For this purpose, this thesis proposes a logic-based approach that able precisely capture the definitions of DSLs as logic theorems. Those theorems are analyzed by advanced logic solver methods in order to derive proofs or counter-examples for expected language properties or to create a set of different models that can be used as test data for modeling environments.

In my thesis, I present three main groups of contributions. (1) I propose an efficient logic solver algorithm for the generation of graph-based models that reason directly over graph structures using partial modeling and 3-valued logic with classic logic solving techniques. (2) I propose a translation and validation techniques for DSL specifications to detect language level inconsistencies, incompleteness, and ambiguity. (3) I propose an iterative, multi-step model generation approach that with three applications. First, (3/A) iterative generation that improves the scalability of existing solver-based model generation techniques. Next, (3/B) an iterative generation is able to measure and control the diversity of the generated models thus improve the quality of test suites. And finally, (3/C) it is able to incrementally resolve view model inconsistencies. I presented my results in the context of three case studies: the validation of an avionics architecture modeling language, the testing of an industrial statechart modeling environment, and view synchronization in a remote health monitoring system.

As a proof-of-concept demonstration of my conceptual results, I developed an open-source modeling tool VIATRA Solver framework, that uses two popular logic solvers (Alloy and Z3), and features the proposed graph solver algorithm, which scales 1-2 orders of magnitude better than existing solutions. The framework natively supports EMF-based (Eclipse Modeling Framework) modeling languages with VIATRA graph patterns for validation and testing and does not require additional theorem-proving skills.

Összefoglaló

Megbízható komplex rendszerek – autók, repülők vagy pénzügyi szoftverek – tervezése során széles körben alkalmaznak gráfalapú modelleket. Szakterület-specifikus nyelvekre (Domain-Specific Language, DSL) épülő fejlett modellezőeszközök jelentősen támogatják a fejlesztési folyamatot azáltal, hogy (1) a fejlesztés alatt álló modelleket folyamatosan ellenőrzik, (2) nézeti modellekkel kiemelik a modell különböző fontos aspektusait, (3) fejlesztési lépéseket automatizálnak például automatikus kódgenerátorok alkalmazásával, valamint (4) a tervek helyességét ellenőrizhetővé tehetik különböző analízis eszközök (például modellellenőrzők) alkalmazásával. Modellezőeszközök alkalmazásával tehát jobb minőségű szoftverek készíthetők, várhatóan kevesebb idő alatt.

Azonban, mint bármely szoftver, maguk a modellezőeszközök és modellezési nyelvek is tartalmaznak hibákat. Mindenekelőtt maguknak a nyelveknek a meghatározásához használt összetett tervezési szabályok is lehetnek ellentmondásosak, többértelműek vagy hiányosak. Továbbá a modellezőeszközben található hibák továbbterjedhetnek a generált kódba, ezáltal érvénytelenítve egy matematikai precíz analízis eredményét. Ezért a modellezőeszközök helyességének ellenőrzése kiemelten fontos. Ez különösen érvényes a biztonságkritikus rendszerek esetén, ahol a felhasznált eszközök alkalmazásához az eszközöknek ugyanolyan szigorú ellenőrzésen kell átesnie, mint magának a végterméknek.

Disszertációm célja modellezőeszközök és modellezőnyelvek ellenőrzésének és tesztelésének támogatása automatikus modellgenerálási technikák segítségével. Munkám során olyan logika-alapú megközelítést alkalmaztam, amely matematikai elméletek formájában képes precízen reprezentálni a vizsgált szakterület-specifikus nyelvek gráf-struktúráját. Ezek az elméletek matematikai következtetési módszerekkel elemezhetővé válnak, ezáltal bizonyítékot vagy ellenpéldát tudunk adni elvárt nyelvi tulajdonságok teljesülésére vagy megszegésére. Ezen felül, az automatikusan előállított helyes modellek tesztbemenetként is hasznosíthatóak.

Disszertációmban három területen mutatok be új tudományos eredményeket. Elsőként készítettem egy újszerű, hatékony modellgenerálásra alkalmas logikai következtető algoritmust, amely háromértékű parciális modelleket és klasszikus logikai algoritmusokat ötvözve közvetlenül gráfalapú logikai struktúrákon következtet. Másodjára, olyan transzformációs és ellenőrzési technikát javasoltam, amely logikai következtetők felhasználásával képes nyelvi szintű ellentmondások, hiányosságok és többértelműségek felfedésére. Végül harmadsorban olyan iteratív többlépéses generálási folyamatot javasoltam, amely nagyban javítja meglévő következtetőkre épülő tesztgenerátorok skálázhatóságát, képes mérni és szabályozni a generált modellek diverzitását ezzel javítani a generált tesztkészlet minőségét, és képes feloldani a nézeti modellekben megfogalmazott inkonzisztenciákat. Eredményeim gyakorlati alkalmazhatóságát három esettanulmány segítségével szemléltetem: egy repülőgép architektúra modellezőnyelv ellenőrzésével, egy ipari állapotgép modellezőeszköz tesztelésével, és távoli egészségügyi felügyeleti rendszer nézeti modelljeinek elemzésével.

Az elméleti eredményeimre építve egy nyílt forráskódú szoftver prototípust is elkészítettem, és publikusan elérhetővé tettem a VIATRA Solver modellgenerátor keretrendszerben, ami integráltan használ két népszerű logikai következtetőt (Alloyt és Z3-at), valamint tartalmazza az új gráfalapú következtető algoritmust, amelynek segítségével 1-2 nagyságrenddel nagyobb modellek is előállíthatóak mint más, logikai következtetésen alapuló eszközökben. A keretrendszer közvetlenül támogatja az EMF (Eclipse Modeling Framework) és VIATRA gráfmintákon alapuló modellezési nyelvek ellenőrzését.

Contents

Chapter 1

Introduction

1.1 Domain-specific modeling languages

My thesis is motivated by the challenges in the development of complex, safety-critical systems such as automotive, avionics or cyber-physical systems, which is characterized by a long development time and strict safety standards (like DO-178C [Do1] or DO-330 [Do3]). *Model-Based System Engineering (MBSE)* is a widely used technique in those application domains [WHR14], which facilitates the use of models in different phases of design and on various levels of abstraction. Furthermore, MBSE promotes the use of *Domain-Specific (Modeling) Languages (DSLs)* to precisely capture the main features of a target domain, thus enabling the engineers to model their solutions with the concepts of the problem domain. Additionally, advanced *modeling environments* can automate several development steps, with a particular emphasis on verification and validation (V&V). Thus they can significantly improve the overall productivity of the development and quality of the product.

A complex industrial modeling environment supports the development of models by continuously evaluating consistency constraints to ensure that the models satisfy the design rules of the domain. There is already efficient support for automatically validating constraints and design rules over large model instances of the DSL using tools like Eclipse OCL [Ocl; Wil12; KPP09] or Viatra [Ber+11; Ber+10]. Modeling environments often incorporate the automated generation of various artifacts such as source code, task specific views, or documentation by using code generation or model transformation techniques. Additionally, the environment may incorporate mathematical analysis techniques to verify the correctness of the underlying design (e.g. by model checking).

Industrial modeling environments like Capella (by Thales), Artop, Yakindu (by Itemis), MagicDraw (by NoMagic) or Papyrus UML (by CEA) are frequently built on top of open source DSL frameworks such as Xtext [Xte], or Sirius [Sir] built on top of model management frameworks such as Eclipse Modeling Framework [Emf] to significantly simplify productivity of tool development by automating the production of rich editor features (e.g. syntax highlighting, auto-completion) to enhance modeling for domain experts.

1.2 Towards the validation of modeling environments

However, the design of modeling environments for complex domain-specific languages is still a challenging task. As such 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 as

part of a software tool qualification process in order to provide trust in their output. Therefore, the need for software tool qualification (e.g. defined in safety-related standards DO-330) raises several challenges for building trusted DSL tools for a specific domain.

1.2.1 Architecture of a modeling environment

Architecturally, a modeling environment is composed of multiple components that need to be validated. First, the language specification of a DSL is based on a *Metamodel (MM)*, which defines the main concepts and relations of the language. The metamodel is extended by additional *Well-Formedness (WF) constraints* to restrict the range of valid models. Moreover, industrial models are frequently extended by *Derived Features (DF)*, which are calculated model elements offering shortcuts to access or navigate models. All together MM, WF and DF specify the graph-based data structure of instance models of the DSL (so-called *abstract syntax*). A modeling environment defines one or more concrete textual or graphical representations for a model (*concrete syntax*), and supports the editing, saving (serializing) and loading (parsing) of the models by providing advanced editors.

View models are a key concept in advanced modeling environments to provide viewpoint-specific focus (e.g., power flow or communication architecture of a system) to engineers by deriving another model (or diagram, table, etc.) which highlights relevant aspects of the system to help detect conceptual flaws. Typically, multiple views are defined for a given model (referred as source model), which are refreshed automatically upon changes in the source model. The derivation and maintenance of views have been extensively studied for a long time in database theory over relational knowledge bases, while it has recently become a popular research topic in MBSE [Deb+14; Gho+15; MC13].

Additionally, a modeling environment incorporates several procedures that use valid models as input, e.g. *model transformations* and *code generators*. Like any piece of software, those components are not free from flaws.

1.2.2 Language level validation

In case of complex, standardized industrial domains (like ARINC 653 [ARI] for avionics or AUTOSAR [AUT13] in automotive), the sheer complexity of the language specification and the models is a major challenge in 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 (one constraint may contradict another) or subsumability (one constraint may accidentally cover another). (2) The specification of derived features can also be inconsistent (e.g. DF specification may contradict another design rule), ambiguous or incomplete (DF specification implies more or fewer values than expected). In summary, the consistency and unambiguity of a DSL specification have to be ensured.

The definition of such large DSLs is a very challenging task not only due to their size and complexity, but also as we need to precisely understand the interactions between the additional design rules. Declaring a large number of DFs is also challenging with respect to the safety-specific WF constraints.

In general, mathematical precise *validation of DSL specifications* has been attempted by only a few approaches so far [JLB11; JS06], and even these approaches lack a systematic validation process. Therefore, I have identified language-level validation as the first research question of my thesis.

Research Question 1: Language Validation. How to validate the consistency, completeness and unambiguity of DSL specifications?

1.2.3 Testing modeling environments

Even if the DSL specification is sound, the implementation of the modeling tool may contain errors. The need for software tool qualification raises several challenges to build trusted DSL tools for a specific domain which typically uses systematic testing methods for the modeling environment itself. Such testing of modeling environments is significantly enhanced by the *automated generation of valid (or intentionally faulty) models* as test inputs [Mou+09] for DSL modeling tools, model transformations or code generators [Bro+06].

Testing of critical systems frequently uses coverage metrics like MC/DC [Hay+01] to assess the thoroughness of a test suite and safety standards prescribe designated minimal test coverage with respect them (e.g. 100% MC/DC coverage for components with SIL5-level criticality). However, existing coverage metrics are dominantly developed for imperative source code, and they fail when applied on testing of modeling environments, which are dominantly data-driven applications working with complex graph structures (e.g. 100% MC/DC coverage can be achieved with a low number of trivial examples while it still provides very little assurance in reality). Therefore, coverage metrics are needed to objectively measure the quality of a test suite in such applications.

Research Question 2: Tool Testing. How to provide a test suite of instance models for a complex DSL with designated coverage?

1.2.4 Backward view synchronization

When several views are derived from a source model in a complex modeling environment, such view models are dominantly read-only representations derived by a unidirectional transformation from a source model, and those views cannot be changed directly. When a view model needs to be changed, the engineer is forced to edit and manually check the source model until the modified model corresponds to the expected view model. Additionally, the effects of a source change need to be observed in all other view models to avoid unintentional changes and to prevent the violation of structural well-formedness (WF) constraints. My thesis focuses on the back propagation of view changes.

My goal was to add backward change propagation support to view models in the modeling environment, so upon a change in the view model, a set of candidates for corresponding source model changes could be created.

Research Question 3: View Synchronization. How to propagate changes of a view model back to the source model to restore consistency?

1.2.5 Graph model generation

All the three questions **1-3** lead to the challenge of model generation as an underlying technique: test suite generation requires the generation of different instance models as test inputs, view synchronization uses model generation to create source model change candidates, and language validation requires the non-existence of counterexample models with undesired properties (like incompleteness and ambiguity) [JS06]. Similarly, a sufficiently powerful solution for any of the previous **Research Questions**, with proper parametrization, could be used as a general purpose graph model generator. This leads to an unifying main research question of the thesis:

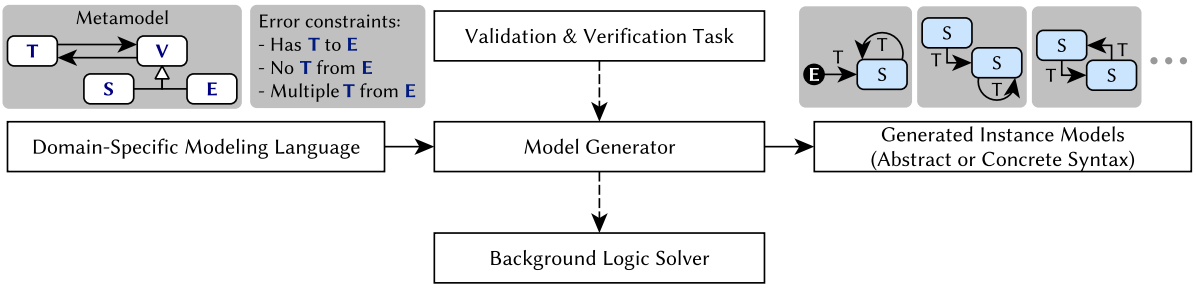


Figure 1.1: Setup of automated model generation

Research Question 4: Graph-Model Generation. How to automatically generate valid instance models for DSLs?

Figure 1.1 illustrates the use of model generation for various challenges of DSL environments as proposed in the current thesis. The setup is based on a *model generator* that reads the definition of the target *Domain-Specific Modeling Language* (typically defined by a metamodel and some well-formedness constraints) to carry out designated *validation and verification tasks* (e.g. expected language property, view model change or required test coverage) by producing certain *instance models* (or proves that no such models can be constructed).

Automated graph model generation is also a prerequisite of several other research lines. Object-oriented data structures can be represented as graphs of objects and pointers, and such, test generation [Mil+07; MK01] requires graph generation with different structures to discover bugs. Similarly, static analysis and verification techniques like [Ren04; RSW04] are using graph consistency checking techniques to ensure that certain invalid structures cannot emerge. Auto-generated graphs may also help the testing and benchmarking in other domains like graph databases [Bag+17; Szá+17; Szá19], since obtaining real graphs from business use cases is often difficult to due to the protection of intellectual property rights.

1.3 Challenges in model generation

As both language level validation, testing and view synchronization (**RQ1-3**) are based on generation of well-formed models (**RQ4**) and they imply similar inherent challenges. Figure 1.2 illustrates the connections between those research questions and challenges.

The main difficulty lies in the complexity of a DSL specification: each generated model needs to satisfy a set of *complex*, *global*, *interacting structural constraints* (a typical characteristic for DSLs). This necessitates the use of advanced *logic reasoning* technique during generation with a *background logic solver*, and a mapping of the DSL specification to (first order) logic and back. Although existing logic solving tools and algorithms (like SMT [DMB08] or SAT-solvers [Jac02]) are getting more and more powerful, their application imposes several challenges. In the following, I highlight four main challenge groups for using logic solvers in (graph) model generation.

First, the complete specification of the target DSL needs to be automatically translated into a formal language in order to represent model generation as a mathematical reasoning problem. The language elements in a DSL specification are representable by sets and relations, and first order logic with transitive closure may formalize most constraints that are needed in practice. Different constraint languages (like OCL, graph patterns or graph predicates) attached to the modeling languages are

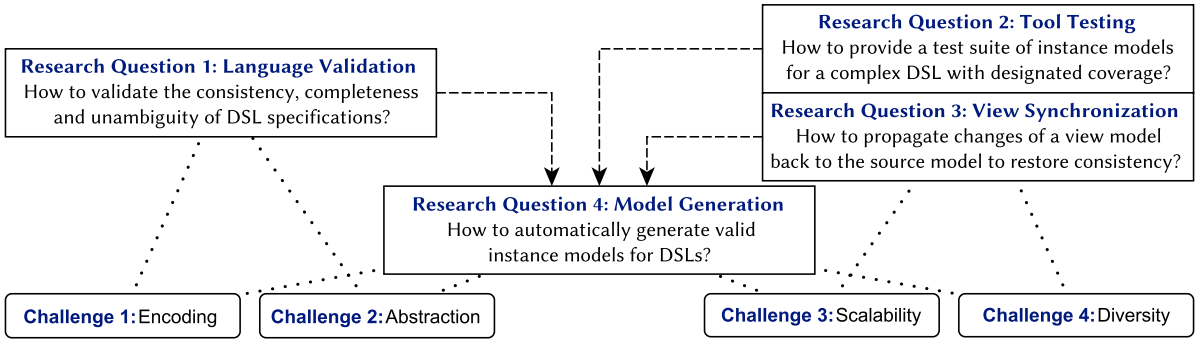


Figure 1.2: Relation between challenges and research questions

semantically close to first order logic, with possible extensions with higher-order language elements, like counting, sum or other aggregates.

Model Generation Challenge 1: Encoding. How to encode DSL specifications as a logic problem where solutions of the logic problem represent valid models?

Language level validation (**RQ1**) gives particularly emphasis on the encoding to ensure that all language elements are correctly and completely mapped to logic. Similarly, a model generator (**RQ4**) need to support the logic problem created from a DSL.

Language level reasoning required for language validation (**RQ1**) is a challenging task as complete analysis has to cover an infinite range of possible design models. Moreover, a DSL specification uses a more expressive specification than first order logic, which is already undecidable. This necessitates the use of sophisticated abstractions and bounded analysis techniques, which sacrifice soundness or completeness, but are still able to carry out the target verification task by proving a stronger assumption, or giving counter-example to weaker one. Similarly, abstraction and approximation are key concepts in logic solvers used for model generation (**RQ4**).

Model Generation Challenge 2: Abstraction. How to provide efficient abstractions for language-level analysis of a DSL?

Even if an abstraction or encoding is sound, it may imply scalability challenges for model generation in practical application scenarios. As the metamodel of an industrial DSL may contain hundreds of types and relations, any meaningful instance model can easily be of similar size. Unfortunately, model generation cannot currently be achieved by a single direct call to the underlying (SAT/SMT) solver [JLB11][C10] due to scalability issues. Our finding was that existing model generation approaches based upon mappings to underlying SMT/SAT solvers could only scale to very small problems due to the fact that the mapping introduces far too many Boolean variables to encode potential graph nodes and edges that immediately explode the search space of the solver. Moreover, when rewriting (language-level) well-formedness constraints to their equivalent Boolean formula over a particular model, the size of this formula is grows rapidly with the size of the model. As such, evaluating a very large formula already creates major challenges for solvers. Indeed, we found no published results in related literature using logic solvers for graph model generation purposes that were able to derive graphs with over 150 nodes. Therefore, existing solver-based generators using currently available

logic solvers fail to derive non-trivial and useful models for complex DSLs (**RQ4**). This hinders the use of model generators in practical tool testing and view synchronization scenarios (**RQ2-3**).

Model Generation Challenge 3: Scalability. How to derive large instance models for complex DSLs?

Finally, existing logic solvers tend to retrieve simple, unrealistic models consisting of unconnected islands, many isolated nodes and highly symmetric (copy-paste) fragments, which is problematic in a real testing scenario (**RQ4**). In fact, test suites created by existing solvers often contain highly similar (or even isomorphic) model sequences, which violates the common best practice of testing, i.e. to use a diverse set of test inputs that cover various equivalence classes. Furthermore, there are no widely used coverage metrics for measuring the diversity of graph-based models. Even preventing isomorphic solution in a model generator is challenging, as it may necessitate computationally expensive graph isomorphism checks.

Model Generation Challenge 4: Diversity. How to measure model diversity and generate a diverse set of models?

Conversely, in bidirectional synchronization (**RQ3**), one of the main goal is to avoid unnecessary changes into prevent information loss.

1.4 Research method

My research method has been aligned with the best practices of software engineering research. First, all **Research Questions 1-4** and **Challenges 1-4** have been motivated by generalizing practical issues gained in the context of industrial case studies. Moreover, the feasibility and usefulness of conceptual contributions have been demonstrated by developing prototype implementations (grouped into the open source Viatra Solver framework [C8]). I integrated my contributions into general-purpose industrial modeling technologies (like EMF [Emf] and Viatra [Ber+11; Ber+10]). Finally, I evaluated the feasibility and scalability of my approach using the prototype implementation on several (scalability) benchmarks derived from existing modeling environments and models of case studies. In my thesis I use three case studies to illustrate the challenges and my solutions on language validation, test generation, and view synchronization.

1.4.1 Avionics Architecture

An avionic DSL validation case study is taken from Trans-IMA [Hor+14] project (and used to answer **Research Question 1**). Trans-IMA aims at defining a model-driven approach for the synthesis of integrated Matlab Simulink models amenable to simulating the software and hardware architecture of an airplane. The project aimed to (i) define a model-driven development process for allocating software functions captured as Simulink models [Mat] over different hardware architectures and (ii) develop domain-specific languages and tools for supporting the definition of the allocation process.

This development environment is built upon eight large metamodels, where complex Viatra graph patterns were extensively used for capturing constraints and derived features. The DSL contains 118 classes, 90 attributes, and 170 references, where 56 features were marked as derived (about 20% of total) and each was specified by a corresponding model query. The design rules are defined by 31 well-formedness constraints. In my work, I validated several expected language properties

for the Functional Architecture Model (FAM) fragment of the Trans-IMA DSL. A FAM represents an abstraction of the avionics functions including their functional decomposition and their corresponding information links (the data flow structure) from a Simulink model. An example functional architecture model is illustrated in Figure 1.3. The development of other popular standardized industrial DSLs like AADL [SAE] and AUTOSAR [AUT13] and have similar challenges.

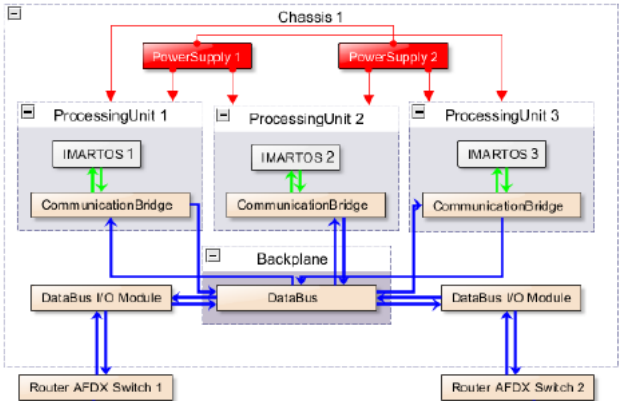


Figure 1.3: Example chassis platform description from Trans-IMA [Hor+14]

1.4.2 Yakindu Statecharts

In my thesis, I used Yakindu Statecharts Tools [Yak] as an industrial case study for test generation (used in **Research Questions 2** and **4**). Yakindu Statecharts Tools is an integrated modeling environment developed by Itemis AG for the specification and development of reactive, event-driven systems based on the concept of statecharts captured in combined graphical and textual syntax. Yakindu simultaneously supports static validation of well-formedness constraints as well as simulation of (and code generation from) statechart models. A sample statechart is illustrated in Figure 1.4.

Validation is crucial for domain-specific modeling tools to detect conceptual design flaws early and ensure that malformed models do not get processed by tooling. Therefore missing validation rules are considered as bugs of the editor. While Yakindu is a stable modeling tool, it is still possible to develop model instances as corner cases which satisfy all (implemented) well-formedness constraints of the language but the simulator or code generator crashes due to synchronization issues.

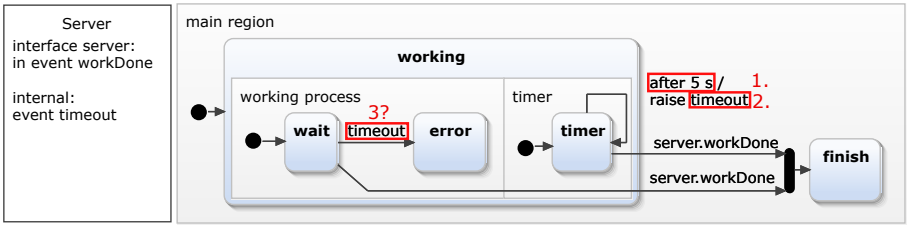


Figure 1.4: Example Yakindu statechart with synchronizations

1.4.3 Remote Healthcare System

My change propagation technique (**Research Questions 3**) is illustrated in a case study of a remote health care system developed in the Concert ARTEMIS project [Con]. The Concerto ARTEMIS research project proposed a reference multi-domain architectural framework for complex, highly concurrent, and multi-core critical systems. As a subproject, it developed a multi-view, hierarchical cross-domain design environment for heterogeneous platform architectures. A view of a medical application is illustrated in Figure 1.5, which presents a dataflow abstraction of an architecture for pulse and blood pressure measurement environment controlled by a smartphone. However, this view is only a read-only representation.

My task was to add backward change propagation support to the view modeling environment, so upon a change on the dataflow abstraction, the framework would propose valid architectural changes.

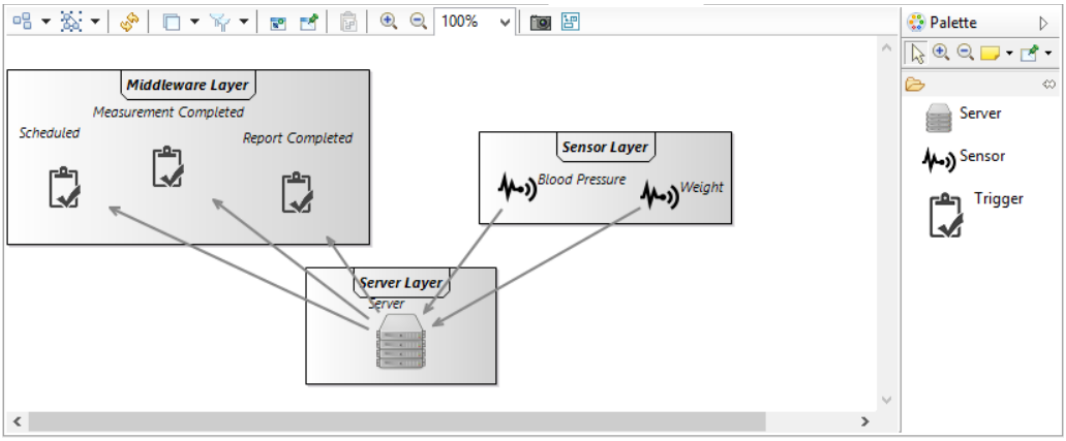


Figure 1.5: Example view model of a remote measurement setup

1.5 Contribution overview and thesis structure

The results of my thesis are organized into three contribution groups.

Contribution group 1. I proposed a novel logic solver that operates directly on graph models. I integrated existing SAT [Jac02; TJ07; LBP10; ES03] and SMT [DMB08] solvers to the framework.
Contribution group 2. I proposed formal analysis techniques for the language-level validation of domain-specific languages by mapping them to formal logic specifications.
Contribution group 3. I proposed iterative techniques for generating a diverse set of input models with increasing model size and source model candidates using the output of logic solvers.

The results of my thesis are organized around the model generation setup, and illustrated in Figure 1.6). **Contributions** are highlighted in blue areas, solid lines denote transformations, dashed lines are dependencies, and dotted lines connect the **Contributions** with **Research Questions** and **Challenges**.

My first contribution group covers a *model generation framework*, addressing **Research Question 4**. The framework takes the formal description of a domain-specific language and a task task encoded

as a logic problem, solves the problem with a background solver, and creates solutions representing instance models. The framework (and the contribution group) features an highly efficient novel *Graph Solver* algorithm, and integrates other popular background logic solvers (Z3 SMT solver [DMB08] and Alloy [Jac02; TJ07] with two underlying SAT solvers [LBP10; ES03]).

Next, my second contribution group proposes an encoding (**Challenge 1**) of domain-specific modeling languages (which covers EMF meta- and instance models, and VIATRA queries) as logic problems. It supports abstraction techniques (**Challenge 2**) to over- and under-approximate constraints to a decidable fragment of logic [PMB08]. It formulates several DSL *validation tasks* (**Research Question 1**) based on the encoding of the DSL elements, which can be checked with the underlying model generator framework.

My third contribution group features iterative model generation techniques. It proposes the generation of models in multiple steps, reusing the previous solution(s) in the construction of the next. For test input generation (**Research Question 2**), it proposes an *incremental* and *diverse* model generation techniques. In incremental model generation, each model is generated as a sequence of models increasing in size where each generation step adds new elements to the existing models, thus improving scalability (**Challenge 3**). For diverse model generation (**Research Question 3**), the thesis proposes a shape-based [Ren04; RD06] distance metric to measure the diversity of models, and controls the model generation by ensuring a minimal distance between each model (facing **Challenge 4**). For view model synchronization (**Research Question 3**), the thesis proposes a backward change propagation technique that reuses the previous state of the source model (as a submodel) when it generates source model candidates to restore consistency.

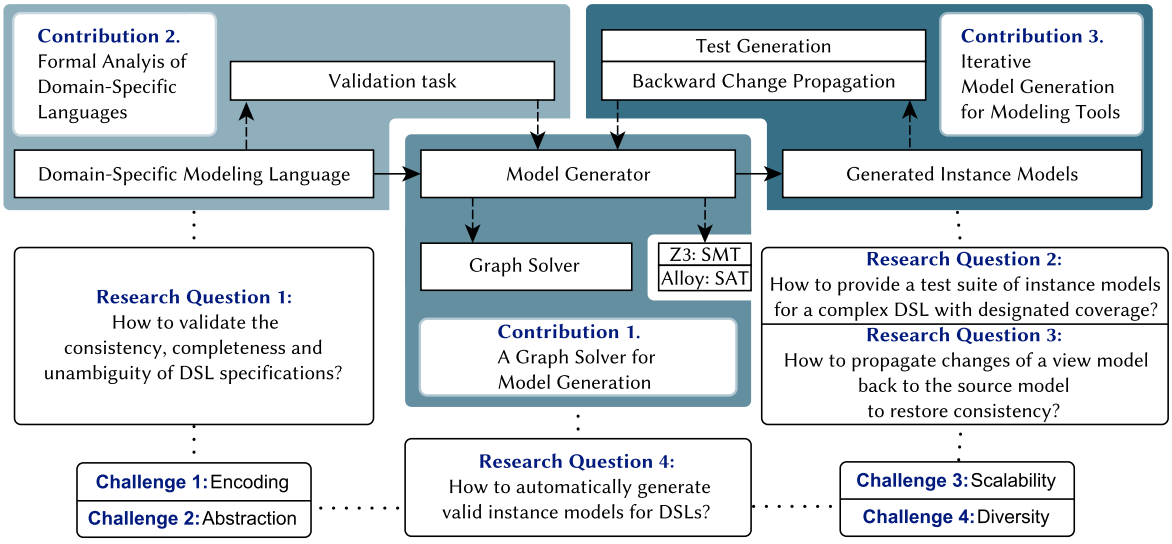


Figure 1.6: Contribution overview

Figure 1.7 illustrates the structure of the thesis. The remaining chapters are structured as follows:

- Chapter 2 gives a concise overview of formal logic, and introduces the mathematical formalism and notation I used in the rest of the thesis.
- Chapter 3 presents the modeling preliminaries and a transformation technique to map modeling concepts to formal logic (**Contribution group 2**).

- Next, Chapter 4 proposes partial models as an extension of logic structures (**Contribution group 1**).
- This serves as the theoretical background for the graph solver (**Contribution group 1**) presented in Chapter 5.
- In Chapter 6 proposes an incremental model generation technique (**Contribution group 3**) to increase the scalability of model generators.
- In Chapter 7 proposes diversity metrics to measure the quality of models in testing scenarios (**Contribution group 3**).
- In Chapter 8 introduces a change propagation technique using model generators (**Contribution group 3**).
- Chapter 9 introduces various language validation techniques for DSLs (**Contribution group 1**) using model generation.
- Finally, Chapter 10 summarizes the results of the thesis and formally states the **Contributions** of my work.

In my thesis, I used three **Case Studies** as running examples I used Yakindu Statecharts as the main case study for most of the thesis (from Chapter 3 to Chapter 7), Remote Healthcare Systems for Chapter 8 and Avionics Architecture for Chapter 9.

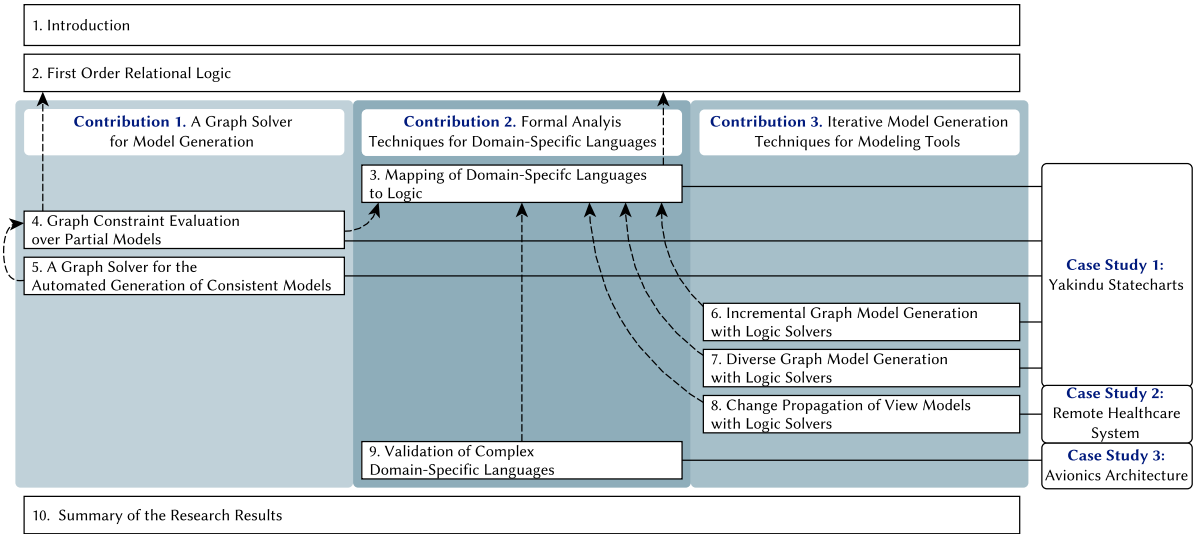


Figure 1.7: Thesis structure with **Contributions**, **Case Studies** and dependencies between the chapters

Chapter 2

First order relational logic

The precise analysis of domain-specific languages necessitates the application of formal mathematical methods. This chapter gives an overview of the mathematical notation used in the thesis. The thesis uses a relational variation of first order (relational) logic (FOL) as the main form of formal representation, which is able to cover the key features of DSLs. First, Section 2.1 introduces the syntax of *first order relational logic with transitive closure*. Then, Section 2.2 defines the precise semantics of a logic expression, theorems and logic models, including approximations of logic expressions and theorems. Next, Section 2.3 defines two possible restrictions of FOL (bounded analysis and effectively propositional logic), and Section 2.4 extends the description power of relational logic with functions and constants (without changing the expression power). Finally, Section 2.5 concludes the chapter.

The notation of the chapter is based on papers [C5] and [C6], but follows classic logic introductions like [MM16]. Similar syntactical and semantical variations of relational logic are used in query languages [VB07] formal methods [RSW04] and model generation approaches [TJ07; KJS11].

2.1 Syntax of first order relational logic

A logic language is defined over a signature with a vocabulary of relational symbols and an arity function.

Definition 1 (Vocabulary of Relational Logic) *A signature is a $\langle \Sigma, a \rangle$ pair, where $\Sigma = \{R_1, \ldots, R_n\}$ is a nonempty, finite set of relation symbols called vocabulary, and $a : \Sigma \to \mathbb{N}$ is an arity function.*

Relational logic expressions of $\langle \Sigma, a \rangle$ are inductively constructed using logic true and false values (1 and 0 respectively), an infinite sequence of variable symbols ($v_1, v_2, \ldots \in \mathcal{V}$), relation application ($R(\cdot, \cdots, \cdot)$), equivalence ($\cdot \sim \cdot$) and inequivalence ($distinct(\cdot, \cdots, \cdot)$), standard logic connectives (negation: $\neg\cdot$, and: $\cdot \wedge \cdot$, or: $\cdot \vee \cdot$, implication: $\cdot \Rightarrow \cdot$ and logic equivalence $\cdot \Leftrightarrow \cdot$), logic quantifiers (exists: $\exists$ and forall: $\forall$), transitive closure over binary relations ($R_i^+(\cdot, \cdot)$).

Definition 2 (Syntax of Relational Logic) *An expression of $\langle \Sigma, a \rangle$ is defined recursively using relational symbols $R \in \Sigma$, variable symbols $v \in \mathcal{V}$ and logic connectives:*

$\langle expr \rangle$	::=	$1 \mid 0$	*logic true and false values*
	$\mid$	$R(\underbrace{\langle term \rangle, \ldots, \langle term \rangle}_{a(R)})$	*relation application,* $R \in \Sigma$
	$\mid$	$\langle term \rangle \sim \langle term \rangle$	*term equivalence*
	$\mid$	$distinct(\langle term \rangle, \ldots, \langle term \rangle)$	*term distinctness*
	$\mid$	$\neg \langle expr \rangle$	*logic negation*
	$\mid$	$\langle expr \rangle \wedge \langle expr \rangle$	*logic and*
	$\mid$	$\langle expr \rangle \vee \langle expr \rangle$	*logic or*
	$\mid$	$\langle expr \rangle \Rightarrow \langle expr \rangle$	*logic implication*
	$\mid$	$\langle expr \rangle \Leftrightarrow \langle expr \rangle$	*logic equivalence*
	$\mid$	$\exists v : \langle expr \rangle$	*existential quantifier,* $v \in \mathcal{V}$
	$\mid$	$\forall v : \langle expr \rangle$	*universal quantifier,* $v \in \mathcal{V}$
	$\mid$	$R^+(\langle term \rangle, \langle term \rangle)$	*transitive closure,* $R \in \Sigma$, $a(R) = 2$
$\langle term \rangle$	::=	v	*variable symbol* $v \in \mathcal{V}$

Additionally, we use parentheses $(\cdot)$ to group expressions. We use the standard precedence of logic:

$$(\cdot), R(\cdot, \cdots, \cdot), R^+(\cdot, \cdot), distinct(\cdot, \cdots, \cdot), \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \sim, \exists, \forall.$$

In a logic expression the same variable can occur in multiple times, which needs special attention:

Definition 3 (Bound and Free variables) *An occurrence o of a variable symbol v is bound by a quantified expression $\exists v : \varphi'$ or $\forall v : \varphi'$, if (1) φ contains that occurrence o, and (2) φ does not contain any quantified subexpression with v that contains the occurrence o ($\exists v : \ldots o \ldots$ or $\forall v : \ldots o \ldots$). If an occurrence of a variable symbol v is not bounded then it is free. If an expression has free variable symbol then it is open, otherwise it is closed.*

Logic expressions create logic predicates and formulae.

Definition 4 (Predicates and Formulae) *An expression φ with free variables $v_1, \ldots, v_n$ is called predicate, and denoted with $\varphi(v_1, \ldots, v_n)$. A closed expression φ is called formula.*

2.2 Semantics of relational logic

The semantic of a logic expression is evaluated over of logic structure, which is defined with a base set and an interpretation function.

Definition 5 (Logic Structure) *A logic structure M of a signature $\langle \Sigma, a \rangle$ is a structure $M = \langle O_M, \mathcal{I}_M \rangle$, where:*

- *O_M is the nonempty base set of individuals in the model (i.e. the objects)*
- *$\mathcal{I}_M(R) : O_M^{a(R)} \to \{0, 1\}$ provides a logic interpretation function for all $R \in \Sigma$.*

When discussing a logic structure, $o_1 \equiv o_2$ denotes the equality of two individuals in set O, and $v_1 \sim v_2$ is used as a logic expression representing the equality of variable values. Two logic structures are considered equivalent, if their base set can be mapped to each other in a way that they give the same interpretation.

Definition 6 (Structure morphism, isomorphism) *Logic structures $M = \langle O_M, \mathcal{I}_M \rangle$ and $N = \langle O_N, \mathcal{I}_N \rangle$ defined over signature $\langle \Sigma, a \rangle$. Function $m : O_M \to O_N$ is called a structure morphism, if*

$$\forall o_1, \ldots o_n \in O_M : \mathcal{I}_M(R)(o_1, \ldots, o_n) = \mathcal{I}_N(R)(m(o_1), \ldots, m(o_n)) \qquad n = a(R)$$

for every $R \in \Sigma$. Structures M and N are equivalent, if there are two morphisms $m_1 : O_M \to O_N$ and $m_2 : O_N \to O_M$.

A logic predicate can be evaluated on a logic structure with a variable binding. A variable binding maps each free variables of an expression to an object of a logic structure.

Definition 7 (Variable Binding) *A variable binding of predicate $\varphi(v_1, \ldots, v_n)$ is a mapping $Z : \{v_1, \ldots, v_n\} \to O_M$ from variables to objects in M.*

In the following definition, *min* and *max* takes the numeric minimum and maximum values of 0 and 1. Furthermore, an extension of a function $f : A \to B$ with a pair $a \mapsto b$ is denoted by $f, a \mapsto b$, which denotes a function $(f, a \mapsto b) : A \cup \{a\} \to B \cup \{b\}$, where:

$$(f, a \mapsto b)(x) = \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise} \end{cases}$$

The semantics of logic expression is defined by a function $[\![\cdot]\!]$ with range $\{0, 1\}$. A predicate $\varphi(v_1, \ldots, v_n)$ is evaluated on logic structure M along a variable binding Z in accordance with the semantic rules of the following definition.

Definition 8 (Semantics of Logic Expressions) *The semantic of predicate $\varphi(v_1, \ldots, v_n)$ over a logic structure M and variable binding Z is denoted by $[\![\varphi(v_1, \ldots, v_n)]\!]^M_Z$, and defined as follows:*

$$\begin{aligned}
[\![1]\!]^M_Z &:= 1 \\
[\![0]\!]^M_Z &:= 0 \\
[\![R_i(v_i, \ldots, v_j)]\!]^M_Z &:= \mathcal{I}_M(R_i)(Z(v_i), \ldots, Z(v_j)) \\
[\![v_1 \sim v_2]\!]^M_Z &:= \begin{cases} 1 & \text{if } Z(v_1) \equiv Z(v_2) \\ 0 & \text{otherwise} \end{cases} \\
[\![\mathit{distinct}(v_1, \ldots, v_n)]\!]^M_Z &:= \mathit{min}\{[\![\neg(v_i \sim v_j)]\!]^M_Z : 1 \leq i < j \leq n\} \\
[\![\neg\varphi]\!]^M_Z &:= 1 - [\![\varphi]\!]^M_Z \\
[\![\varphi_1 \wedge \varphi_2]\!]^M_Z &:= \mathit{min}([\![\varphi_1]\!]^M_Z, [\![\varphi_2]\!]^M_Z) \\
[\![\varphi_1 \vee \varphi_2]\!]^M_Z &:= \mathit{max}([\![\varphi_1]\!]^M_Z, [\![\varphi_2]\!]^M_Z) \\
[\![\varphi_1 \Rightarrow \varphi_2]\!]^M_Z &:= [\![\neg\varphi_1 \vee \varphi_2]\!]^M_Z \\
[\![\varphi_1 \Leftrightarrow \varphi_2]\!]^M_Z &:= [\![(\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)]\!]^M_Z \\
[\![\exists v : \varphi]\!]^M_Z &:= \mathit{max}\{[\![\varphi]\!]^M_{Z, v \mapsto x} : x \in O_M\} \\
[\![\forall v : \varphi]\!]^M_Z &:= \mathit{min}\{[\![\varphi]\!]^M_{Z, v \mapsto x} : x \in O_M\} \\
[\![R^+(v_1, v_2)]\!]^M_Z &:= \begin{array}{l} \mathit{max}([\![R(v_1, v_2)]\!]^M_Z, \\ \qquad \mathit{max}\{[\![\exists m_1, \ldots, m_n : R(v_1, m_1) \wedge \ldots \wedge R(m_n, v_2)]\!]^M_Z : n \in \mathbb{N}^+\}) \end{array}
\end{aligned}$$

The semantics of $R^+(v_1, v_2)$ is defined for all positive length $n \in \mathbb{N}^+$, but for finite models, checking only interval $n \in [1; |O|]$ is sufficient.

2.2.1 Theories and models

Next, we define theories as a collection of formulae.

Definition 9 (Theory, Axioms) *A theory $\mathcal{T} = \{\varphi_1, \varphi_2, \ldots\}$ is a (possibly infinite) set of formulae. Formulae $\varphi_1, \varphi_2, \ldots$ are referred as axioms of theory $\mathcal{T}$*

Then, we define when a logic structure satisfies a theory.

Definition 10 (Model) *A logic structure M satisfies a formula φ (denoted by $M \models \varphi$) if $[\![\varphi]\!]^M = 1$. Similarly, a logic structure M satisfies a theory $\mathcal{T}$ (denoted by $M \models \mathcal{T}$) if $M \models \varphi$ for all $\varphi \in \mathcal{T}$. Such an M is called the model of the formula or theory. If a structure M does not satisfies (or, in other words, violates) a formula φ or theory $\mathcal{T}$ it is denoted by $M \not\models \varphi$ and $M \not\models \mathcal{T}$. The (possibly infinite) set of models that satisfies a theory $\mathcal{T}$ is denoted by $\mathcal{M}_{\mathcal{T}}$.*

An important characterization of a theory is that it satisfiable by or not.

Definition 11 (Consistency, Inconsistency) *A theory $\mathcal{T}$ is consistent, if there is a model M which satisfies $\mathcal{T}$: $M \models \mathcal{T}$. Otherwise, $\mathcal{T}$ is inconsistent.*

Semantic consequence is defined with consistency.

Definition 12 (Semantic Consequence) *A formula φ is a semantic consequence of theory $\mathcal{T}$ (denoted as $\mathcal{T} \models \varphi$) if there is no model M for which $M \models \mathcal{T}$ and $M \not\models \varphi$.*

In other words, if a model satisfies a theory, it satisfies all semantic consequences. It is important to note that for an inconsistent theory $\mathcal{T}$ every φ is a semantic consequence.

2.2.2 Approximations

Definition 13 (Approximations of Predicates) *Predicate $\varphi^U(v_1, \ldots, v_n)$ underapproximates (similarly $\varphi^O(v_1, \ldots, v_n)$ overapproximates) a predicate $\varphi(v_1, \ldots, v_n)$ if it satisfies the following implications for every evaluation:*

$$\forall v_1, \ldots, v_n : \varphi^U(v_1, \ldots, v_n) \Rightarrow \varphi(v_1, \ldots, v_n)$$

$$\forall v_1, \ldots, v_n : \varphi(v_1, \ldots, v_n) \Rightarrow \varphi^O(v_1, \ldots, v_n)$$

As a trivial example, constant $\varphi^O(v_1, \ldots, v_n) = 1$ predicate is always a good overapproximation, and $\varphi^U(v_1, \ldots, v_n) = 0$ underapproximates every predicate. A formula also approximates itself. So the strategy of our mapping is to express most of formulae in the target designated logic fragment language, and approximate the inexpressible features. An axiom system $\mathcal{T}$ can be also approximated to $\mathcal{T}^U$ or $\mathcal{T}^O$ if every axiom is approximated in it. Approximating axiom systems allows us to reason over problems that otherwise would be undecidable.

Theorem 1 (Approximated Axiom systems) *If $\mathcal{T}^U$ and $\mathcal{T}^O$ are under- and overapprocimated axiom systems, and $\varphi(v_1, \ldots, v_n)$ is a predicate, then*

$$\mathcal{T}^U \models \varphi \Rightarrow \mathcal{T} \models \varphi$$

$$\mathcal{T}^O \not\models \varphi \Rightarrow \mathcal{T} \not\models \varphi$$

Additionally:

$$\begin{array}{rcl} \mathcal{T}^U \text{ satisfiable} & \Rightarrow & \mathcal{T} \text{ satisfiable} \\ \mathcal{T}^O \text{ unsatisfiable} & \Rightarrow & \mathcal{T} \text{ unsatisfiable} \end{array}$$

2.3 Restrictions of relational logic

Because FOL problems are undecidable in general, SMT and SAT solvers can use one or a combination of multiple background theories, therefore they can reason over a certain set of logic problems. This paper uses two kinds of restrictions of relational logic with existing decision procedure: *bounded analysis* and *effectively proposal logic* [PMB08].

First, bounded analysis (BA) deals with models only with a specific number of objects. Solvers like Alloy [TJ07] with SAT solvers [LBP10; ES03] and Z3 [DMB08] allow bounded analysis.

Definition 14 (Bounded analysis) *Analysis with bounded size s of a theory $\mathcal{T}$ deals with models M, where (1) $M \models \mathcal{T}$, and (2) $|O_M| = s$. In other words, $\mathcal{T}$ contains two formulae:*

$$\varphi_{min} := \exists v_1, \ldots, v_s : distinct(v_1, \ldots, v_s)$$

$$\varphi_{max} := \forall v_1, \ldots, v_s, v_{s+1} : \neg distinct(v_1, \ldots, v_s, v_{s+1})$$

Another background theory is effectively propositional logic (EPR) [PMB08] as it provides logical formulae that can cover the large set of DSL language features yet provide efficient reasoning capabilities over a potentially infinite range of models. For example, Z3 contains background theorem [GM09] to reason over effectively propositional logic.

Definition 15 (Effectively propositional logic) *Effectively propositional logic is a fragment of first order relational logic, where each formulae φ of a theory $\mathcal{T}$ is defined in the following form:*

$$\exists e_1, \ldots, e_n \forall a_1, \ldots, a_m : \varphi'(e_1, \ldots, e_n, a_1, \ldots, a_m), \quad 0 \leq n, m$$

and φ' does not contain quantifier or transitive closure.

2.4 Extensions of relational logic

Classic first order logic may also incorporate functions, constants and types beside relations. In the following, I show that those extensions can be reduced to the relational definition of first order logic.

An n-ary *function* f defines a *total mapping* of n elements to a *single target*. A nullary function is called a *constant*. Function symbols are also used as terms in logic expression with the following syntax:

$$\langle term \rangle ::= f(\underbrace{\langle term \rangle, \ldots, \langle term \rangle}_{n}) \quad \text{Function application, } f \text{ is } n\text{-ary function symbol}$$
$$\mid \; c \quad \text{Constant value, } c \text{ is a constant symbol}$$

Function symbols and function applications can be reduced to relation symbols and relational applications. First, an n-ary function f can be modelled as relation R_f with arity $a(R_f) = n + 1$. In order to ensure the functional property of the relations, the following additional assertions need to be added to the axioms:

$$\begin{aligned} &\textbf{Total:} && \forall v_1, \ldots v_n : \exists v_{n+1} : R_f(v_1, \ldots, v_n, v_{n+1}) \\ &\textbf{Single Target:} && \forall v_1, \ldots v_n, v_{n+1}, v'_{n+1} : \\ & && (R_f(v_1, \ldots, v_n, v_{n+1}) \wedge R_f(v_1, \ldots, v_n, v'_{n+1})) \Rightarrow (v_{n+1} \sim v'_{n+1}) \end{aligned}$$

In case of constant symbol c, the assertions simplified to the following expressions:

$$\begin{aligned} &\textbf{Total:} && \exists v : R_c(v) \\ &\textbf{Single Target:} && \forall v, v' : (R_c(v) \wedge R_c(v')) \Rightarrow (v \sim v') \end{aligned}$$

In accordance, logic expressions need to be rewritten to relational logic expression, which is carried out by replacing function applications and constant values with the corresponding relations.

1. In an expression, only subexpressions $R(\cdot, \cdots, \cdot)$ and $\cdot \sim \cdot$ contain function applications or constant values, thus we search for those subexpressions and handle them independently as follows.

2. Let φ be a logic expression $R(\cdot, \cdots, \cdot)$ or $\cdot \sim \cdot$ which contains function applications $f_1(t_1^1, \ldots, t_i^1)$, …, $f_n(t_1^n, \ldots, t_j^n)$ and constant symbols $c_1, \ldots, c_m$. All function application and constant value terms are replaced with (newly introduced) variables $v_{f_1}, \ldots, v_{f_n}$ and $v_{c_1}, \ldots, v_{c_m}$ respectively, which creates a relational expression $\varphi'(v_{f_1}, \ldots v_{f_n}, v_{c_1}, \ldots, v_{c_m})$.

3. Then, φ is is replaced with the following expression:

$$\varphi'' = \overbrace{\exists v_{f_1}, \ldots v_{f_n}, v_{c_1}, \ldots, v_{c_m}}^{\text{introducing variables to replace terms with}} : \overbrace{\varphi'(v_{f_1}, \ldots v_{f_n}, v_{c_1}, \ldots, v_{c_m})}^{\text{replacing terms with variables}} \wedge \underbrace{R_{f_1}(t_1^1, \ldots, t_i^1, v_{f_1}) \wedge \ldots \wedge R_{f_n}(t_1^n, \ldots, t_j^n, v_{f_1})}_{\text{binding variables } v_{f_1}, \ldots v_{f_n} \text{ to functions}} \wedge \underbrace{R_{c_1}(v_{c_1}) \wedge \ldots \wedge R_{c_m}(v_{c_m})}_{\text{binding variables } v_{c_1}, \ldots v_{c_m} \text{ to constants}}$$

Example 1. Let *parent* be an unary function, and *root* a constant. The following logic expression means that the parent of the root is itself: φ:=$parent(root) \sim root$.

In order to represent this problem in relational logic, *parent* and *root* is replaced with R_{parent} binary and R_{root} unary relation, and the following assertions needs to be added to the problem:

$$\forall v_1 : \exists v_2 : R_{parent}(v_1, v_2) \qquad \forall v_1, v_2, v'_2 : (R_{parent}(v_1, v_2) \wedge R_{parent}(v_1, v'_2)) \Rightarrow v_2 \sim v'_2$$

$$\exists v : R_{root}(v) \qquad \forall v, v' : (R_{root}(v) \wedge R_{root}(v')) \Rightarrow v_2 \sim v'_2$$

The relational logic equivalent of φ is the following

$$\varphi' := \exists v_1, v_2, v_3 : R_{parent}(v_1, v_2) \wedge R_{root}(v_1) \wedge R_{root}(v_3) \wedge v_2 \sim v_3$$

2.5 Summary

This chapter introduced the formalism first order relational logic (FOL), which will be used as the basic notation through the thesis. We showed that this notation is equivalent with classic notation of first order logic using constants and functions. Moreover, it introduced two restricted, efficiently analyzable fragments of FOL: bounded analysis (BA) and effectively propositional logic (EPR). However, the theory graph patterns (GP) in VIATRA [VB07] or the OCL language [Ocl] (OCL) are more expressive than first order relational logic (FOL). Therefore, certain constraints such as recursive patterns, transitive closures, set cardinalities and check expressions cannot be directly compiled into FOL, which needs the applications of approximations. The expressiveness of the different constraint languages and logic fragments can be summarized as follows:

$$\text{BA, EPR} < \text{FOL} < \text{GP, OCL.}$$

Next, Chapter 3 introduce a mapping technique to represent modeling artifacts to FOL symbols and formulae, which will be used in the rest of the paper to automatically solve different modeling problems with automated reasoning.

Chapter 3

Mapping of Domain-Specific Languages to Logic

Complex domain-specific languages necessitate a combination of different specification techniques. The abstract syntax of the DSL is usually captured by a *metamodel*. To create an advanced modeling environment, the metamodel can be augmented with *well-formedness constraints* (or *design rules*), which capture additional restrictions any well-formed instance model needs to respect. Such constraints can be defined by model queries or as OCL invariants.

This chapter defines the translation of a DSL to first order (relational) logic. The chapter is structured as follows: first, Section 3.1 introduces the modeling concepts of EMF [Emf] and Viatra query language [Var+16; VB07]. Moreover, it introduces Yakindu Statecharts as the main case study of the thesis. Next, Section 3.2 presents the overview of the translation technique. Then Section 3.3 presents the mapping of EMF metamodels, Section 3.4 the mapping of Viatra queries. Our OCL [Ocl] translation translation technique is also included in Section A.1, although not the contribution of this thesis. Finally, summarizes the chapter. The chapter is based on publications [C9] and [J2].

3.1 Modeling preliminaries

This section introduces the language of EMF metamodeling language [Emf] and Viatra query language [Var+16; VB07], and illustrates the Yakindu Statechart [Yak] case study.

3.1.1 Metamodeling

A metamodel $MM = \langle D_{MM}, S_{MM} \rangle$ defines the main concepts, relations of the target domain (D_{MM}), and specifies the basic structure of the models (S_{MM}). The thesis uses the Eclipse Modeling Framework (EMF) [Emf] for domain modeling, which is used by many industrial modeling tools including Capella (Thales), Artop, Yakindu (Itemis), MagicDraw (NoMagic) and Papyrus. Thus systematically generating test models for tool certification provides an immediate application with high practical relevance. In EMF, a metamodel is defined by the following domain:

$$D_{MM} = \langle Cls_{MM}, Data_{MM}, Enum_{MM}, Lit_{MM}, lit_{MM}, Ref_{MM}, Attr_{MM} \rangle$$

In a metamodel MM, Cls_{MM} represents the set of *EClasses* (which are simply referred to as classes) that can be instantiated to *EObjects* (or objects) in an instance model. Additionally, a metamodel refers

to a set of predefined data types $Data_{MM}$, which includes primitive types like *EInt* (integers), *EString* (strings), or introduce custom *EEnum* types (enumerated types) $Enum_{MM}$ with predefined set of literal elements Lit_{MM}. The metamodel Cls_{MM} specifies the set of literals of an enumerated type with function $lit_{MM} : Enum_{MM} \to 2^{Lit_{MM}}$; for each literal $\mathsf{l} \in Lit_{MM}$ there is exactly one enumerated type $\mathsf{E} \in Enum_{MM}$ where $\mathsf{l} \in lit_{MM}(\mathsf{E})$. Relations between the classes are captured with a set of *EReference*s (references) Ref_{MM}. Finally, the relations between classes and data types are captured by *EAttribute*s (or attributes) $Attr_{MM}$. Sets Cls_{MM}, $Data_{MM}$, Ref_{MM}, $Attr_{MM}$, $Enum_{MM}$ and Lit_{MM} are finite and disjoint.

An EMF metamodel $MM = \langle D_{MM}, S_{MM} \rangle$ also specifies several structural constraints in S_{MM}:

$$S_{MM} = \langle sup_{MM}, abs_{MM}, src^{ref}_{MM}, trg^{ref}_{MM}, src^{attr}_{MM}, trg^{attr}_{MM}, mul^{min}_{MM}, mul^{max}_{MM}, inv_{MM}, cont_{MM} \rangle, \text{ where:}$$

- Relation $sup_{MM} \subseteq Cls_{MM} \times Cls_{MM}$ specifies a *generalization* between two classes to express that a more specific (child) class has every structural feature of the more general (parent) class. In EMF, multiple inheritance is supported, but loops are forbidden:

$$\neg\exists c_1, c_2 \ldots, c_{n-1}, c_n \in Cls_{MM} : sup_{MM}(c_1, c_2) \wedge \ldots \wedge sup_{MM}(c_{n-1}, c_n) \wedge sup_{MM}(c_n, c_1)$$

- Relation $abs_{MM} \subseteq Cls_{MM}$ select some of the classes as *abstract*, which means it is disallowed to have direct instances.

- Functions $src^{ref}_{MM}, trg^{ref}_{MM} : Ref_{MM} \to Cls_{MM}$ selects the *source* and *target types* of the references.

- Function $src^{attr}_{MM} : Attr_{MM} \to Cls_{MM}$ selects the *container class* of the attribute, and $Attr_{MM} : Ref_{MM} \to Data_{MM}$ selects the *value type*.

- Functions $mul^{min}_{MM} : Ref_{MM} \cup Attr_{MM} \to \mathbb{N}$ and $mul^{max}_{MM} : Ref_{MM} \cup Attr_{MM} \to \mathbb{N}^+ \cup \{*\}$ specifies *lower* and *upper multiplicity* bounds to the references and attributes, typically written in lower..upper form. Multiplicity $*$ denotes unlimited upper multiplicity.

- Two parallel but opposite directional *inverse* references can be defined as inverses of each other to specify that they always occur in pairs with relation $inv_{MM} \subseteq Ref_{MM} \times Ref_{MM}$. Relation inv_{MM} is symmetric, and the source and target types of inverse relations have to be the opposite:

$$\forall r_1, r_2 \in Ref_{MM} : inv_{MM}(r_1, r_2) \Leftrightarrow inv_{MM}(r_2, r_1)$$

$$\forall r_1, r_2 \in Ref_{MM} : inv_{MM}(r_1, r_2) \Rightarrow [src^{ref}_{MM}(r_1) = trg^{ref}_{MM}(r_2) \wedge trg^{ref}_{MM}(r_1) = src^{ref}_{MM}(r_2)]$$

- EMF instance models are arranged into a strict *containment hierarchy*, which is a directed tree along relations marked by the set $cont_{MM} \subseteq Ref_{MM}$.

Example 2. A metamodel extracted from Yakindu is illustrated in Figure 3.1. A Statechart consists of Regions, which in turn contain Vertexes and Transitions. An abstract state Vertex is further refined into RegularStates (like State) and PseudoStates like Entry and Synchronization states. The source and target states of a transition are identified by the source and target references.

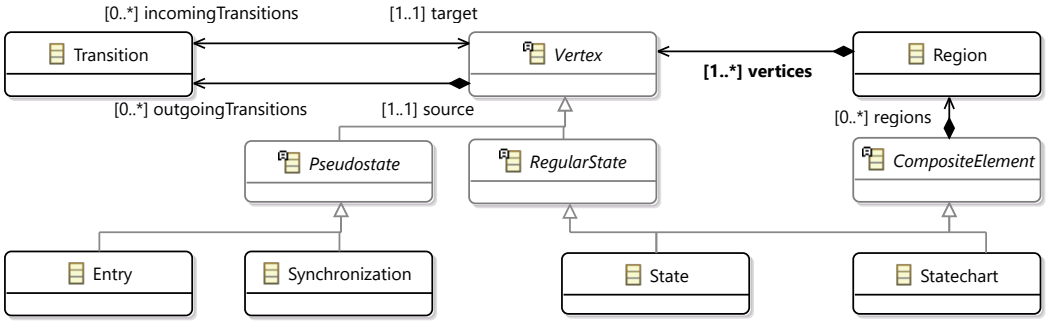


Figure 3.1: Metamodel extract from Yakindu state machines

3.1.2 Instance models

An instance model of a metamodel *MM* are graph-based data structures that use the types and relations introduced in *MM*. This thesis uses logic structures (see Section 2.2) to capture this graph-structure, which are defined over a signature $\langle \Sigma_{MM}, a_{MM} \rangle$ derived from the metamodel:

- $\Sigma_{MM} := Cls_{MM} \cup Data_{MM} \cup Enum_{MM} \cup Lit_{MM} \cup Ref_{MM} \cup Attr_{MM}$
- $a_{MM}(s) = \begin{cases} 1 & \text{if } s \in Cls_{MM} \cup Data_{MM} \cup Enum_{MM} \cup Lit_{MM} \\ 2 & \text{if } s \in Ref_{MM} \cup Attr_{MM} \end{cases}$

In a logic structure $M = \langle O_M, \mathcal{I}_M \rangle$ representation a model, O_M denotes the objects and values of primitive data types (i.e. the nodes of the graph structure), and $\mathcal{I}_M$ represents both the types of the objects and the relations between them (i.e. both node labels and edges with edge labels). The precise representation technique of the instance model as a logic structure follows the mapping of a metamodel to logic theorem, and will be introduced accordingly in Section 3.3.

> **Example 3.** Figure 3.2 illustrates a simple statechart model with a single state and an entry transition using concrete syntax of Yakindu, and as a logic structure. The model corresponds to the metamodel in illustrated in Figure 3.1. Model $M = \langle O_M, \mathcal{I}_M \rangle$ contains four objects (denoted by boxes): $O_M = \{r, e, s, t\}$. The interpretation of unary relations in $\mathcal{I}_M$ are represented as node labels, e.g. Region=1 in node r means that $\mathcal{I}_M(\text{Region})(r) = 1$, missing labels are considered 0. Binary relations are denoted with edges, e.g. $\mathcal{I}_M(\text{vertices})(r, e) = 1$, and missing edges are also considered false values (e.g. $\mathcal{I}_M(\text{vertices})(e, r) = 0$).

3.1.3 Model queries

Model queries are frequently captured by graph patterns (GP) [VB07; Ber+11], which is an expressive formalism used for various purposes in model-driven development alternatively for standard OCL constraints [Ocl]. A graph pattern is a graph-like structure representing a condition (or constraint) matched against a typically large instance model.

A model query $\mathtt{q}(\mathtt{p}_1, \ldots, \mathtt{p}_n) = def$ is defined by a name $\mathtt{q}$, symbolic parameters $\mathtt{p}_1, \ldots, \mathtt{p}_n$, and conditions (or constraints) over the parameters (captured by *def*). A match m of $\mathtt{q}(\mathtt{p}_1, \ldots, \mathtt{p}_n)$ over model M maps each symbolic parameter $\mathtt{p}_i$ to a model element (object, enum literal or primitive)

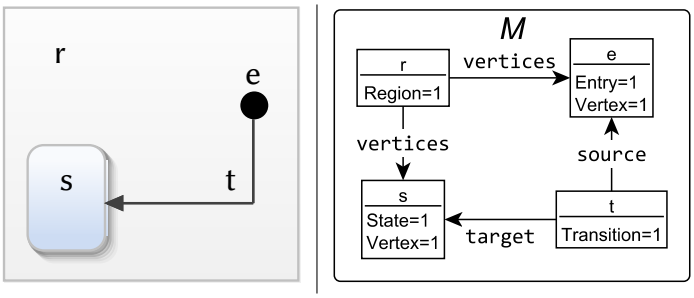


Figure 3.2: Simple statechart model with concrete syntax (left) and as a logic structure (right)

from the target model M, which satisfies the conditions of *def*. The task of the query evaluation on a model is to produce each match that satisfies this condition. VIATRA offers a textual language describing graph patterns as a set of constraints. The following definition summarizes the grammar of the language, the complete query language is described in [Ber+11], while relevant language features will be introduced on demand in several examples below adapted from [C9][J2].

Definition 16 (Syntax of Graph Patterns) *A graph pattern is defined as follows:*

⟨*pattern*⟩ ::= ⟨*annotation*⟩ `pattern` ⟨*name*⟩ `(` ⟨*params*⟩ `)` ⟨*bodies*⟩
⟨*params*⟩ ::= ⟨*param*⟩ | ⟨*param*⟩`,` ⟨*params*⟩
⟨*param*⟩ ::= ⟨*var*⟩ | ⟨*var*⟩`:` ⟨EClassifier⟩
⟨*bodies*⟩ ::= `{` ⟨*conlist*⟩ `}` | `{` ⟨*conlist*⟩ `}` `or` ⟨*bodies*⟩
⟨*conlist*⟩ ::= ⟨*constraint*⟩`;` | ⟨*constraint*⟩`;` ⟨*constlist*⟩
⟨*constraint*⟩ ::= ⟨*classifier*⟩ | ⟨*path*⟩ | ⟨*equality*⟩ | ⟨*call*⟩ | ⟨*check*⟩
⟨*classifier*⟩ ::= ⟨C⟩`(`⟨*var*⟩`)` — C ∈ Cls_{MM}
⟨*path*⟩ ::= ⟨C⟩`.`⟨*featlist*⟩`(`⟨*var*⟩`,`⟨*var*⟩`)` — C ∈ Cls_{MM}
⟨*featlist*⟩ ::= ⟨A⟩|⟨R⟩|⟨R⟩`.`⟨*featlist*⟩ — A ∈ $Attr_{MM}$,R ∈ Ref_{MM}
⟨*equality*⟩ ::= ⟨*var*⟩`==`⟨*var*⟩|⟨*var*⟩`!=`⟨*var*⟩
⟨*call*⟩ ::= `find` ⟨*name*⟩ `(` ⟨*binding*⟩ `)` |
`neg find` ⟨*name*⟩ `(` ⟨*binding*⟩ `)` |
`find` ⟨*name*⟩ `+(` ⟨*binding*⟩ `)`
⟨*binding*⟩ ::= ⟨*var*⟩ | ⟨*var*⟩`,`⟨*binding*⟩
⟨*check*⟩ ::= `check(`⟨*boolexp*⟩`)`
⟨*annotation*⟩ ::= `@Constraint`|`@QueryBasedFeature`|ε

A graph pattern is identified with a *unique name* and specified with a *parameter list* and some *bodies*. The parameters refer to objects, enum literals or primitive types where the type of a parameter can be explicitly defined. The bodies specify constraints over the parameters. A pattern may have multiple bodies with *constraints*, and may introduce additional local variables beside the parameters. If a variable is used only once it is specified as an anonymous variable with '_' as the first character in its name. A pattern with multiple bodies means a disjunction (`or`), thus a valid *match* necessitates that all the constraints are satisfied by a mapping of those variables for at least one body.

The following types of constraints are supported:

- **Classifier constraint:** checks if a variable is an instance of an EClass.

- **Path constraint:** requires a specific reference, an attribute, or a path of EReference and EAttribute sequence between two variables.
- **Equality constraint:** specifies that two variables have to be mapped to the same model element.
- **Pattern call constraint:** enables the composition of multiple patterns. The positive pattern call refers to another pattern and specifies that the called pattern must be satisfied in the context of the actual parameters. Additionally, a pattern may be composed negatively (`neg` keyword), which means that the target negative pattern is disallowed to have a valid match along the actual parameters. Finally, it is possible to compute the transitive closure of a two-parameter pattern by the + symbol.
- **Check constraint:** evaluates a specific attribute expression on the variables of the pattern and accept matches only if the result of attribute condition is true. In this paper, the basic arithmetic and logic operators are covered.

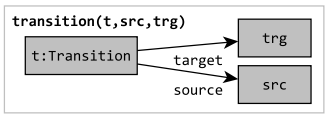
In VIATRA, it is possible to mark the patterns with `@Constraint` annotation to use them as ill-formedness patterns (i.e. as negation of well-formedness constraints), or make them define the values of derived features with the `@QueryBasedFeature` annotation.

By default, the result of a model query expressed as a graph pattern is the set of *all* matches with different values for the pattern parameter variables. However, by *binding* parameter variables to specific model elements or attribute values it is possible to filter the returned values. This allows the use of the same pattern for getting all possible matches and for checking whether a selected match is present in the result set.

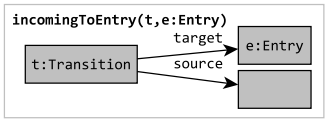
> **Example 4.** Figure 3.3 illustrates five graph patterns defined using both graphical and textual syntax of VIATRA. The first pattern `transition(t,src,trg)` defines a relation between two Vertices which are connected via a Transition using source and target references. Referring to this pattern, three well-formedness constraint is defined concerning Entry states: if any of them has a match, then the model is malformed. First, `incomingToEntry(t, e)` selects invalid Transitions that are leading to an Entry (by referring to the `transition` pattern using the `find` language construct). Next, `noOutgoingTransitionFromEntry(e)` matches to Entry states that does not have any outgoing Transition (by negatively referring to the `transition` using the `neg find`). Pattern `multipleTransitionFromEntry(e,t1,t2)` selects Entries with multiple outgoing Transition (with two `find` expressions), that are not equal (`!=`). Finally, `SynchronizedVerticesInSameRegion` defines a complex error pattern, where a Synchronization uses two Vertices from the same Region (and not synchronizing parallel vertices). The pattern covers both cases where the Vertices are the source or the target of the Synchronization (denoted by the `or` of two bodies).

3.2 Transformation overview

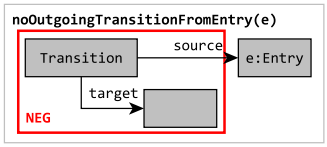
We now discuss the details of transforming DSL artifacts to first order logic formulae to be processed by solvers. Due to its excessive length, the details of the transformation is split into three sections, and we first present some theoretical foundations and the detailed mapping of metamodels and partial snapshots in Section 3.3 followed by transformation of the constraint languages into FOL in Section 3.4.



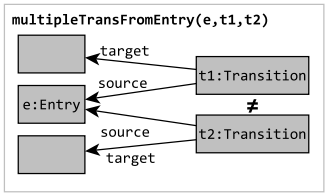

```
pattern transition(t,src,trg) {
  Transition.source(t,src);
  Transition.target(t,trg);
}
```



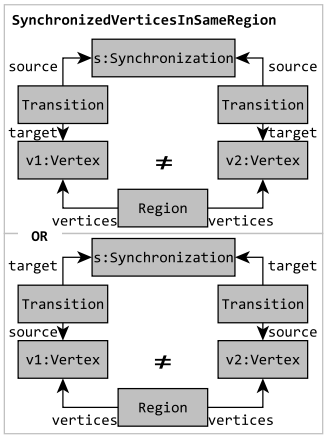

```
@Constraint
pattern incomingToEntry(t, e:Entry) {
  find transition(t,_,e);
}
```




```
@Constraint
pattern noOutgoingTransitionFromEntry(e:Entry) {
  neg find transition(_,e,_);
}
```




```
@Constraint
pattern multipleTransitionFromEntry(e:Entry,t1,t2) {
  find transition(t1,e,_);
  find transition(t2,e,_);
  t1 != t2;
}
```




```
@Constraint
pattern SynchronizedVerticesInSameRegion(
  s:Synchronization,v1,v2) {
  find transition(t,v1,s);
  find transition(t,v2,s);
  Region.vertices(r,v1);
  Region.vertices(r,v2);
  v1 != v2;
} or {
  find transition(t,s,v1);
  find transition(t,s,v2);
  Region.vertices(r,v1);
  Region.vertices(r,v2);
  v1 != v2;
}
```

Figure 3.3: Example graph patterns defined with graphical and Viatra syntax

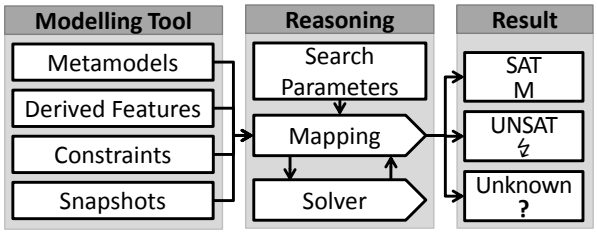


Figure 3.4: Functional overview of the approach

3.2.1 Functional overview of the transformation

This section provides a high-level, functional overview of the DSL analysis approach using an underlying logic solver. Our approach aims to analyze the DSL specification of *modeling tools* by mapping them into a set of logic symbols Σ, and a first order logic theorem $\mathcal{T}$, which can be processed by advanced *reasoning applications* such as *SMT solvers* or *SAT solvers*. The outcome of a reasoning problem is either *Satisfiable* or *Unsatisfiable*. If the problem is satisfiable, the solver constructs an output (or completed) model $M \models \mathcal{T}$ (which is interpreted as *Witness* or *Counterexample* depending on the task), while an unsatisfiable result means a *Contradiction*. Because certain validation tasks are undecidable in FOL it is also possible that validation terminates with an *Unknown* answer or a timeout. Finally, the results of the reasoning needs to be traced back and interpreted in modeling terms as attributes of the DSLs. Linking the independent reasoning tool to the modeling tool allows the DSL developer to make mathematically precise deductions over the developed languages and models including different validation techniques and test generation scenarios.

The validations are initiated and executed in well-defined *context*, which defines $\mathcal{T}$ for the model generation run. This context can be customized during model generation by selecting (or de-selecting) certain DSL artifacts from the following list. As a result, the output model M retrieved during DSL validation needs to respect the context.

- **Metamodels:** The set of domain classes allowed to be instantiated for constructing models can be restricted by explicitly selecting classes and structural features of a metamodel MM. By default, each class from each metamodel is used in the analysis. Then M has to satisfy the constraints of this (possibly pruned) metamodel: $M \models MM$.
- **Derived Features:** The values of the derived features have to be correctly evaluated with respect to their definition yielding unique and complete results (denoted as $M \models DF$).
- **Constraints:** The output model has to satisfy the selected well-formedness constraints: $M \models WF$, thus certain constraints can be relaxed or strengthened for a reasoning process.
- **Partial Snapshots:** The output model M has to combine partial snapshots according to their semantic parameters, denoted as $M \models PS$. Partial snapshots act as explicit assumptions (or proof obligations) in a validation scenario, so by default, no PS is passed to the solver.
- **Search Parameters:** Additionally, the user may define some reasoning-specific input parameters:
  - **Scope:** The number of objects used in the construction of an output model can be restricted by an integer number (defined by $|O_M| \leq size$). By default, $size = *$, which means that the analysis covers all each possible model regardless of its size.

– **Approximation level:** Some DSL property (such as the acyclicity of the containment hierarchy) cannot be represented in FOL. The method is customizable with the level of approximation (see Section 3.2.3), which allows to set the limit of approximation level. Higher approximation level will reduce the possibility of false positives.

The constraints serving as the context of DSL validation are summarized as $\mathcal{T} = MM \cup DF \cup WF \cup PS$, and it defines a (possibly infinite or empty) set of models $\mathcal{M}_{\mathcal{T}} = \{M : M \models \mathcal{T}, |O_M| = size\}$. The logic problem is translated to the input format of the selected logic solver in order to automatically create instances.

3.2.2 Foundations of the transformation

The transformation takes a DSL context as input to create a set of logic symbols Σ, and a set of formulae $\mathcal{T}$ over Σ, which is satisfiable if and only if the original DSL context was consistent. If the $\mathcal{T}$ is satisfiable then by definition there is an interpretation $M \models \mathcal{T}$. Additionally, we back-annotate the logic structures M derived as a result of the validation process to an actual instance model (or partial snapshot) of the DSL, formally:

If the reasoning tool finds the axiom system satisfiable an example interpretation will be created that explicitly defines a (symbolic) value for every uninterpreted feature of the axiom system (e.g. how many objects are there 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 for back-annotation purposes.

3.2.3 Approximation techniques

The constraints in $\mathcal{T}$ may contain expressions which cannot be processed or effectively handled by the underlying solver thus approximation techniques have to be applied. The use of approximations is an integral part of the proposed approach, which is handled in the validation process as Figure 3.5 illustrates it. Based on the *Approximation Parameters*, under- and over-approximations are applied on the logic problems during the transformation to create stronger or weaker conditions by adding, removing or modifying formulae. The modified problem is expected to be solved more efficiently by the target logic reasoner, and the result can be *Satisfiable* with a model (which is only a *Candidate Model* of the original problem), *Unsatisfiable* or *Unknown*. Because of the approximations, an output of this modified problem needs some additional analysis:

- If an *underapproximated* problem is *satisfiable*, then the original problem is *satisfiable* too, and the candidate model (output of the modified problem) is acceptable for the original problem.
- If an *overapproximated* problem is *unsatisfiable*, then the original problem is *unsatisfiable* too.
- But if an *underapproximated* problem is *unsatisfiable*, then it is uncertain if the original problem has contradictions so the result is *unknown*.
- If an *overapproximated* problem is *satisfiable*, then the candidate model might be a *false positive* which does not satisfy the original problem, so additional validation with the original constraints is necessary. Structural correctness, OCL and Viatra constraints can be easily checked on a candidate instance model by dedicated language level validation tools (like Viatra or OCL interpreters). If the validation is successful, then the candidate model is valid in the context of the original problem, otherwise the process fails with *unknown*.

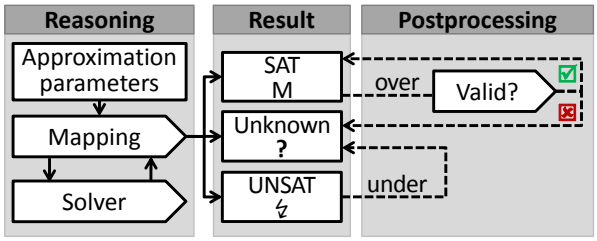


Figure 3.5: Reasoning with approximated constraints

Classifier	Constant symbols	Type Predicate
*EObject*s	–	Object(·)
EEnum type E with literals $lit(E) = l_1, \ldots, l_n$	$l_1, \ldots, l_n$	E(·)
EBoolean	`false`, `true`	Boolean(·)
EInt	$\ldots, -1, 0, +1, \ldots$	Integer(·)
EDouble, *EFloat*	`0.0`, `0.1`, ...	Real(·)
EString	`""`, `"a"`, `"aa"` ...	String(·)

Table 3.1: Mapping of EMF data types

- If the theorem prover provides a model which is formally correct but does not expected in real scenarios (due to implicit WF-constraints), then it is a *spurious counterexample*. To handle those irrelevant cases, the counterexample is generalized into a partial snapshot supplied to the solver in consecutive validation runs.

By using abstraction in the validation process complex language elements can handled in the validation even if they cannot directly handled by the solver.

3.3 Transforming metamodels and partial snapshots

This section describes the mapping of a metamodel $MM = \langle D_{MM}, S_{MM} \rangle$, and represent models as logic structures.

3.3.1 Objects and Primitive Values

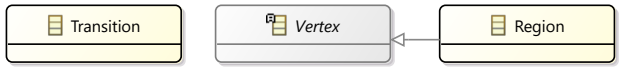
In EMF, instance models are graph structures represented by of *EObject*s and primitive values (strings, integers) as nodes, and *EReference*s links and *EAttribute* values as edges. First, *EObject*s are represented with a predicate Object. Additionally, primitive data types $d \in Data_{MM}$ are represented with a predefined domain of constants and a type predicate. VIATRA Solver supports the basic EMF data types, and they are mapped to a set of elements in accordance with Table 3.1.

3.3.1.1 Classes

Every class in a Cls_{MM} is transformed to unary characteristic predicate symbol.

EMF	$C \in Cls_{MM}$
FOL	$C(\cdot)\ , \forall v : C(v) \Rightarrow \text{Object}(v)$

EMF



FOL

$$\forall o :$$

	Transition		Vertex		Region	
Transition	(Transition(o)	$\wedge$	$\neg$Vertex(o)	$\wedge$	$\neg$Region(o)	)$\vee$
Region	($\neg$Transition(o)	$\wedge$	Vertex(o)	$\wedge$	Region(o)	)

Figure 3.6: Mapping type hierarchy

For example, class State is transformed to predicate State$(\cdot)$, and State(v) is true if v is an instance of State. In a model M, $[\![\mathsf{C}(v)]\!]^M_{v\mapsto o}$ is 1 then the object o is an instance of class C.

3.3.1.2 Type hierarchy

In many cases, an object is an instance of multiple classes due to the generalization relation between the classes, and the existence of abstract classes which do not have direct instances. A simple way to represent the type hierarchy is using a table where the columns represent the possible classes and the rows the concrete (non-abstract) classes. A cell represents a literal whether the type in the row is compatible with the type in the column.

$$\forall o : \mathsf{Object}(o) \Rightarrow \left(\bigvee_{\substack{\mathsf{C}\in Cls_{MM} \\ \mathsf{C}\notin abs_{MM}}} \bigwedge_{\mathsf{S}\in Cls_{MM}} [\mathsf{C}(o) \Leftrightarrow sup_{MM}(\mathsf{C},\mathsf{S})] \right)$$

An extract of the transformation of class hierarchy is shown in Figure 3.6.

3.3.2 References and Attributes

References of the metamodels define the directed edges between the instance objects. Attributes of a metamodel are the properties of the classes with built-in type. For EMF models, we allow directed loops for references, and disallow parallel edges of the same type between the same objects. In such a case, edges can be treated as relations (in the mathematical sense).

EMF	$\mathsf{R} \in Ref_{MM}$	$\mathsf{A} \in Attr_{MM}$
FOL	$\mathsf{R}(\cdot,\cdot)$	$\mathsf{A}(\cdot,\cdot)$

In a model M, if $[\![\mathsf{R}(v_1,v_2)]\!]^M_{v_1\mapsto o_1, v_2\mapsto o_2}$ is 1, then there is a link of C from object o_1 to object o_2. Similarly, if $[\![\mathsf{A}(v_1,v_2)]\!]^M_{v_1\mapsto o_1, v_2\mapsto o_2}$ is 1, then object o_1 has an attribute A value of o_2.

3.3.2.1 Type compliance

In order to ensure the type compliance, type information needs to be attached to the two relation ends. Attributes are transformed in the same way as relations, but the second parameter (i.e. the target) of the parameter defines the data type of the attribute.

EMF	$\mathsf{R} \in Ref_{MM}, \mathsf{C}_1 = src^{ref}_{MM}(\mathsf{R}), \mathsf{C}_2 = trg^{ref}_{MM}(\mathsf{R})$	$\mathsf{A} \in Attr_{MM}, \mathsf{C} = src^{attr}_{MM}(\mathsf{A}), \mathsf{D} = trg^{attr}_{MM}(\mathsf{A})$
FOL	$\forall s,t : \mathsf{R}(s,t) \Rightarrow (\mathsf{C}_1(s) \wedge \mathsf{C}_2(t))$	$\forall o,v : \mathsf{A}(o,v) \Rightarrow (\mathsf{C}(o) \wedge \mathsf{D}(v))$

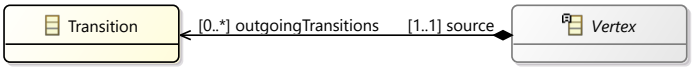
EMF		
FOL	**Decl:**	$\mathsf{outgoingTransitions}(\cdot,\cdot)$, $\mathsf{source}(\cdot,\cdot)$
	TC:	$\forall s,t : (\mathsf{outgoingTransitions}(s,t) \Rightarrow (\mathsf{Vertex}(s) \wedge \mathsf{Transition}(t))$
		$\forall s,t : (\mathsf{source}(s,t) \Rightarrow (\mathsf{Transition}(s) \wedge \mathsf{Vertex}(t))$
	Mul-Upper:	$\forall s,t_1,t2 : (\mathsf{source}(s,t1) \wedge \mathsf{source}(s,t2)) \Rightarrow t_1 = t_2$
	Mul-Lower:	$\forall s : \mathsf{Transition}(s) \Rightarrow \exists t : \mathsf{source}(s,t)$
	INV:	$\forall s,t : \mathsf{source}(s,t) \Leftrightarrow \mathsf{outgoingTransitions}(t,s)$

Figure 3.7: Mapping references

3.3.2.2 Multiplicity

By default, references and attributes with $0..*$ multiplicity are modeled with relations. However, with explicit multiplicity restrictions, further assertions are required. A reference or attribute R with $mul^{min}_{MM}(\mathsf{R})..mul^{max}_{MM}(\mathsf{R})$ multiplicity, the lower bound means that every object is in relation with at least $mul^{min}_{MM}(\mathsf{R})$ different one, which is checked using an existential quantifier (if $mul^{min}_{MM}(\mathsf{R}) > 0$).

EMF	$\mathsf{R} \in Ref_{MM}, m = mul^{min}_{MM}(\mathsf{R}), m > 0, \mathsf{S} = src^{ref}_{MM}(\mathsf{R}), \mathsf{T} = trg^{ref}_{MM}(\mathsf{R})$
FOL	$\forall s : \mathsf{S}(s) \Rightarrow (\exists t_1,\ldots,t_m : distinct(t_1,\ldots,t_n) \wedge \mathsf{R}(s,t_1) \wedge \ldots \wedge \mathsf{R}(s,t_n))$

The upper bound m means that there are no more than m different target elements being in relation with the object, which is prescribed using a universal quantifier.

EMF	$\mathsf{R} \in Ref_{MM}, m = mul^{max}_{MM}(\mathsf{R}), m \neq *, \mathsf{S} = src^{ref}_{MM}(\mathsf{R}), \mathsf{T} = trg^{ref}_{MM}(\mathsf{R})$
FOL	$\forall s,t_1,\ldots,t_m,t_{m+1} : (\mathsf{R}(s,t_1) \wedge \ldots \wedge \mathsf{R}(s,t_m) \wedge \mathsf{R}(s,t_{m+1})) \Rightarrow \neg distinct(t_1,\ldots,t_m,t_{m+1})$

For example, the transformation of source reference of the Transition class visible in Figure 3.7, creates constraint to restrict the upper bound multiplicity to 1.

3.3.2.3 Inverse edges

Inverse edges in metamodels express in that if there is a relation R_1 from the object s to the target t then there has to be an inverse relation R_2 from t to o. The inverse relationship between source and outgoingReferences references is illustrated in Figure 3.7 at line **INV**.

EMF	$(\mathsf{R}_1,\mathsf{R}_2) \in inv_{MM}$
FOL	$\forall s,t : \mathsf{R}_1(s,t) \Leftrightarrow \mathsf{R}_2(t,s)$

3.3.2.4 Containment

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

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

$$contains(\cdot,\cdot), \qquad \forall s,t : contains(s,t) \Leftrightarrow \bigvee_{\mathsf{R} \in cont_{MM}} \mathsf{R}(s,t)$$

2. The top of the containment hierarchy is called root of the model, which is declared as an unary predicate, which is true for exactly one object.

$$root(\cdot), \qquad \exists r : root(r) \wedge \text{Object}(r), \qquad \forall r_1, r_2 : root(r_1) \wedge root(r_2) \Rightarrow r_1 \sim r_2$$

3. The root object does not have a parent.

$$\forall p, r : root(r) \Rightarrow \neg contains(p, r)$$

4. Every other object has at least one parent:

$$\forall o : \text{Object}(o) \Rightarrow (root(o) \vee (\exists p : contains(p, o))$$

5. Every object has at most one parent:

$$\forall c, p_1, p_2 : (contains(p_1, c) \wedge contains(p_2, c)) \Rightarrow (p_1 \sim p_2)$$

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

$$\forall o : \neg contains^{+}(o, o)$$

3.3.2.5 Expression power of metamodel constructs

Table 3.2 summarises the transformed features of the metamodel. It also presents which property is expressible in FOL or EPR or BA (as defined in Section 2.3).

Features of the metamodel	FOL	EPR	BA
Unlimited # of *EObject*s	+	+	−
*EClass*es	+	+	+
Class hierarchy	+	+	+
*EEnum*s	+	+	+
*EReference*s	+	+	+
*EAttribute*s	+	+	+
Multiplicity upper bound	+	+	+
Multiplicity lower bound	+	−	+
Inverse edges	+	+	+
Containment hierarchy	+	-	+

+: Expressible, −: Inexpressible

Table 3.2: Expressing metamodel features in FOL

3.3.3 Transformation of instance models as partial snapshots

Existing instance models can be incorporated to the theory of model generation to ensure that solutions contain a specific submodel. Such submodels are called *partial snapshots*.

The basic approach of transforming partial snapshots into FOL is to create a statement to express that the output logic structure needs to contain the partial snapshot as a substructure. Let

$P = \langle \mathcal{O}_P, \mathcal{I}_P \rangle$ denote a model used as a partial model. First, each object is is represented by a unique variable $r : \mathcal{O}_P \to \mathcal{V}$, so if $o_1 \not\equiv o_2$ then $r(o_1) \not\equiv r(o_2)$. Next, $\mathcal{I}_P$ is encoded as as set of expressions. Class expressions are encoded to expressions φ_i^{Class} in the following way:

PS	$[\![\mathsf{C}(v)]\!]^P_{v \mapsto o} = 1$
FOL	$\mathsf{C}(r(o))$

References and attributes values are mapped similarly to expressions φ_j^{Ref} and φ_k^{Attr}:

PS	$[\![\mathsf{R}(v_1, v_2)]\!]^P_{v_1 \mapsto o_1, v_2 \mapsto o_2} = 1$	$[\![\mathsf{A}(v, d)]\!]^P_{v \mapsto o} = 1$
FOL	$\mathsf{R}(r(o_1), r(o_2))$	$\mathsf{A}(r(o), d)$

Therefore, in summary, a constraint enforcing the presence of a partial model $P = \langle \mathcal{O}_P, \mathcal{I}_P \rangle$, where $\mathcal{O}_P = \{o_1, \ldots, o_n\}$ is constructed in the following way:

$$\exists r(o_1), \ldots, r(o_n) : distinct(r(o_1), \ldots, r(o_n)) \wedge \bigwedge_i \varphi_i^{Class} \wedge \bigwedge_j \varphi_j^{Ref} \wedge \bigwedge_k \varphi_k^{Attr}$$

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

Mapping a partial snapshot to the theory is mainly driven by how to transform instance models to corresponding formulae which are included in the set of axioms for a DSL context. In general, the metamodel and the constraints of the language define universally quantified formulae over all model elements. Instance models enable to efficiently configure the validation process using large existentially quantified properties

Features of the PS	FOL	EPR	BA
Instance Objects	+	+	+
Abstract or Concrete Types	+	+	+
Filled References	+	+	+
Filled Attributes	+	+	+

+: Expressible, −: Inexpressible

Table 3.3: Mapping partial snaptshots to FOL

3.4 Transforming constraints to first order logic

This subsection describes how graph patternsin VIATRA language can be transformed to first order logic formulae.

3.4.1 Structure of graph queries

A graph query Q consists of a list of symbolic parameters as header and a content that specifies logical conditions over the parameters. The parameter list of the query Q is a fix sized vector of variables

	Declaration	Bodies	Constraints
VQL	`pattern synch (s,v1,v2){...}`	`{` b_1 `}or{` b_2 `}`	c_1:`find transition(t,v1,s);` c_2:`find transition(t,v2,s);` c_3:`Region.vertices(r,v1);` c_4:`Region.vertices(r,v2);` c_5:`v1 != v2;`
FOL	$\text{synch}(s, v_1, v_2)$	$\text{synch}(s, v_1, v_2) \Leftrightarrow$ $b_1(s, v_1, v_2) \vee b_2(s, v_1, v_2)$	$b_1 = \exists t, r : c_1 \wedge c_2 \wedge \ldots \wedge c_5$

Figure 3.8: Example pattern structure transformation

denoted as $p_1, \ldots p_n$. In order to represent a query Q in the axiom system, their match set of each pattern is transformed to a relation symbol Q, where the arity of relation is equal to the number of parameters: $a(\text{Q}) = n$. The value of predicate $\text{Q}(\cdot, \cdots, \cdot)$ is 1 in logic structure M for a specific assignment parameters $Z : \{p_1, \ldots, p_n\} \rightarrow O_M$ exactly when Z constitutes a match:

VQL	`pattern` Q $(p_1, \ldots, p_n)$	Q has a match Z in M
FOL	$\text{Q}(\cdot, \cdots, \cdot), a(\text{Q}) = n$ $\forall p_1, \ldots, p_n : \text{Q}(p_1, \ldots, p_n) \Leftrightarrow$ constraint over $(p_1, \ldots, p_n)$	$[\![\text{Q}(p_1, \ldots p_n)]\!]^M_Z = 1$

In Viatra, the constraint of a query is specified by disjunction of bodies denoted by the `or` keyword. A match satisfies the query condition if it satisfies the condition defined by one of the queries bodies, which are defined the disjunction conditions in the query body.

VQL	`pattern` Q $(p_1, \ldots, p_n)\{b_1\}$`or`$\ldots$`or`$\{b_m\}$
FOL	$\text{Q}(p_1, \ldots, p_n) \Leftrightarrow (b_i(p_1, \ldots, p_n) \vee \ldots \vee b_m(p_1, \ldots, p_n))$

The query body is defined by a list of constraints, where the condition is the conjunction of the constraints. A body may introduce additional existentially quantified local variables. (A variable might be introduced for a single use in a constraint, in this case the variable is specified as anonymous with '_' as the first character in its name.)

VQL	$\{c_1(pars, vars); \quad \ldots \quad c_n(pars, vars);\}$
FOL	$b_i(p_1, \ldots, p_n) \Leftrightarrow \exists vars \bigwedge_{1 \leq i \leq n} c_j(pars, vars)$

Example 5. Figure 3.8 illustrates the mapping of pattern `transition` with the parameter list `t,src,trg`. The matches of this pattern are defined by the predicate in **Declaration** column of Figure 3.8. Pattern `transition` specifies two bodies which is illustrated at the in **Bodies** column of Figure 3.8. Both bodies has five-five constraint, which are mapped to a conjunction.

The following subsection defines the transformation for each supported type of constraints.

3.4.2 Constraint Mapping

Classifier constraints define the type of the objects that are bound to a variable. A graph constraint can be easily compiled to a type predicate as follows:

VQL	$\langle C \rangle(v)$;	$v:\langle C \rangle$
FOL	$C(v)$	$C(v)$

Path constraints define that there is a path consisting of a sequence of references of corresponding types that leads from a source object to a target object (identified by pattern variables). In the most simple case, a path constraint consists of navigating along a single reference or attribute, which is transformed in the following way:

VQL	$\langle C \rangle.\langle R \rangle(src, trg)$;	$\langle C \rangle.\langle A \rangle(src, trg)$;
FOL	$R(src, trg)$	$A(src, trg)$

For complex paths, we introduce implicit object variables as the inner nodes of the path, thus the expression can be compiled into a conjunction of simple reference predicates form the first variable through the inner ones to the last.

VQL	$\langle C \rangle.\langle R_1 \rangle. \ldots .\langle R_{n-1} \rangle.\langle F_n \rangle(src.trg)$;, F_n is either in Ref_{MM} or $Attr_{MM}$
FOL	$\exists v_1, \ldots v_{n-1} : R_1(src, o_1) \wedge [\bigwedge_{2 \le i \le n-2} R_i(o_{i-1}, o_i)] \wedge F_n(o_{n-1}, trg)$

Equality and non-equality of two individuals can be simply defined as FOL equality:

VQL	$a==b$;	$a!=b$;
FOL	$a \sim b$	$distinct(a, b)$

Pattern call constraints enable the creation by calling elementary ones. The following list provides the transformation rules for each kind of transformation, and an example result of a negative pattern call is presented in the **Pattern call** column of Figure 3.9.

- A positive call defines that the substituted parameters have to create a match of the referred pattern.
- Negative calls may introduce new (in this case universally quantified) variables. A negative pattern call defines that the target pattern should not have a match for the substituted old variables with for any possible substitution of the new parameters.
- Transitive closure is an advanced language element of the VIATRA pattern language. The transitive closure of a two-parameter GP matches on the pair (e_1, e_n) if there is a sequence $e_1, e_2, \ldots e_n$ of model elements where the pattern matches every pair (e_i, e_{i+1}).

VQL	`find` $\langle Q \rangle$ $(v_1, \ldots, v_n)$;	`neg find` $\langle Q \rangle$ $(v_1, \ldots, v_n)$;	`find` $\langle Q \rangle$ $+(v_1, v_2)$;
FOL	$Q(v_1, \ldots, v_n)$	$\forall \underbrace{\ldots, v_i \ldots}_{\text{new}} : \neg Q(v_1, \ldots, v_n)$	$Q^+(v_1, v_2)$

Example 6. Figure 3.9 illustrates four example constraint transformations.

	Classifier	Path	Equivalence	Pattern call
VQL	`e:Entry | Entry(e);`	`Trans.source(t,src);`	`t1 != t2;`	`neg find t(_,e,_);`
FOL	$\mathrm{Entry}(e)$	$\mathrm{source}(t, src)$	$\neg(t_1 \sim t_2)$	$\forall v_1, v_2 : \neg R_t(v_1, e, v_2)$

Figure 3.9: Example constraints

Transitive closure approximation The transitive closure of a pattern can only be approximated in FOL. The essence of this approximation is to generate a sequence of predicates p_i by unrolling its definition so that each predicate p_i checks for matches of length i. At depth i, a predicate checks if there is a match exactly at length i or recursively checks for a match at depth $i+1$ by using predicate p_{i+1}. At maximal depth n, the predicate is overapproximated by true (1) or underapproximated by false (0).

Example 7. For example, let us define an overapproximation for length 2 of the transitie call of `connected`:

Transitive pattern:

```
pattern connected(src,trg) {
  Transition.source(t,src);
  Transition.target(t,trg);
}
```

Transitive approximations:

2: $R^{+,O=2}_{connected}(src, trg) = R_{connected}(src, trg) \vee \exists m : (R_{connected}(src, m_1) \wedge R^{+,O=1}_{connected}(m, trg))$

1: $R^{+,O=1}_{connected}(src, trg) = R_{connected}(src, trg) \vee \exists m : (R_{connected}(src, m_1) \wedge R^{+,O=0}_{connected}(m, trg))$

0: $R^{+,O=0}_{connected}(src, trg) = R_{connected}(src, trg) \vee \exists m : (R_{connected}(src, m_1) \wedge \mathbf{1})$

Overapproximated pattern specification:

$R^{+}_{connected}(This, P) \Rightarrow R^{+,O=2}_{connected}(This, P)$

Check expression By the use of check constraint it is available to call imperative (Java-like) xBase expressions to be evaluated on the variables of the pattern. A check constraints specifies that the result of the evaluation have to be true for each valid match. This paper discusses the translation of basic arithmetic and logic operators, which are simply translated to the corresponding logic expression.

VQL	`a+b`	`a-b`	`a*b`	`a/b`	`a==b`	`a&&b`	`a||b`	`!a`
FOL	$a + b$	$a - b$	$a \cdot b$	a/b	$a \sim b$	$a \wedge b$	$a \vee b$	$\neg a$

3.4.3 Patterns for advanced DSL constructs

Graph patterns are used in different ways to specify restrictions on the structure of the DSL by well-formedness (or ill- formedness) constraints or defining derived features.

- In Viatra, *well-formedness* is defined by *error patterns* (using the `@Constraint` keyword) are defined as a statement that the model is free from matches of this pattern:

VQL	`@Constraint pattern`$\langle q \rangle$ `(`v_1`,...,`v_n`)`
FOL	$\forall v_1, \ldots, v_n : \neg R_q(v_1, \ldots, v_n)$

Features of model query	FOL	EPR		BA
		DF	WF	
Classifier constraint	+	+	+	+
Path constraint	+	−	+	+
Acyclic pattern call	+	−	+	+
Negative pattern call	+	−	+	+
Transitive closure	+	−	A	+
Arbitrary call graph	+	−	−	+
Aggregate (eg. Count, Sum)	A	−	−	+
Check (for algebra)	−	−	−	−

+: Expressible, −: Inexpressible, A: Approximable

Table 3.4: Expressing Ecore and VIATRA language features in logic language

- Predicates for *derived features* state that features evaluate to the value exactly when the specifying pattern has a match on the given object and the value.

VQL	`@QueryBasedFeature(feature=⟨F⟩)` `pattern` $\langle q \rangle$ (v_1, v_2), $F \in Ref_{MM}$ or $F \in Attr_{MM}$
FOL	$\forall v_1, v_2 : R_q(v_1, v_2) \Leftrightarrow F(v_1, v_2)$

3.4.4 Expression power of graph patterns

Table 3.4 shows an overview on which feature can be translated to FOL, EPR or BA when using them as well-formedness constraints or as derived features.

3.5 Summary

This chapter presented a transformation technique from a DSL specification to a logic theory. The presented technique supports the translation of EMF metamodels and innovatively incorporates graph predicates (of VIATRA) capturing derived features and well-formedness rules. In summary, a DSL can be translated to a signature $\Sigma = \{C_1, \ldots, C_n, R_1, \ldots, R_m, Q_1, \ldots, Q_o\}$, where:

- a unary relation symbol C_i $(1 \leq i \leq n)$ is defined for each *EClass*,
- a binary relation symbol R_j $(1 \leq j \leq m)$ is derived for each *EReference* (or *EAttribute*).
- an n-ary relation symbol Q_k $(1 \leq k \leq o)$ is derived for each graph predicate φ_k.

The theory of a DSL (denoted as *DSL*) is summarized as axioms over Σ derived from the structural constraints of the metamodel S_{MM}, the definition of the graph predicates $\varphi_1, \ldots, \varphi_o$, the definitions of the derived features *DF* and well-formedness constraints *WF*, and optionally, the definition of initial partial snapshots *PS*. Therefore, a logic structure *M* represent a valid instance model of *DSL* if:

$$M \models MM \wedge DF \wedge WF \wedge PS.$$

In the following, this generic transformation technique will be used to generate instance models by logic solvers.

Chapter 4

Graph Constraint Evaluation over Partial Models by Constraint Rewriting

4.1 Introduction

During the early phase of development as well as in case of software product line engineering, the level of uncertainty represented in the models is still high; it gradually decreases as more and more design decisions are made. To support uncertainty during modeling, a rich formalism of partial models has been proposed in [FSC12a] which marks model elements with four special annotations (namely, may, set, variable and open) with well defined semantics. During the design, these partial models can then be concretized into possible design candidates [SFC12][C10].

However, evaluating well-formedness constraints over partial models is a challenging task. While existing graph pattern matching techniques provide efficient support for checking well-formedness constraints over regular model instances [KPP09; NNZ00; Ujh+15; Búr+15], SMT/SAT solvers have been needed so far to evaluate the same constraints over partial models, which have major scalability problems [C10]. In general, a a graph generation problem for n nodes (objects) for a metamodel with $\#C$ concepts (classes) and $\#R$ relations (references) is represented with $O(\#C \cdot n + \#R \cdot n^2)$ number of Boolean variables. This causes a huge state-space explosion even for relatively small models (25.000-65.000 variables for 50-80 nodes with 10 concepts and 10 relations). Moreover, complex, quantified well-formedness constraints need to be evaluated on those variables.

Our objective is to evaluate well-formedness constraints over partial models by graph pattern matching instead of SAT/SMT solving, which poses several conceptual challenges. First, a single node in a graph constraint may be matched to zero or more nodes in a concretization of a partial model. Moreover, graph constraints need to be evaluated over partial models with open world semantics as new elements may be added to the model during concretization.

In the chapter, we propose (i) a new partial modeling formalism based on 3-valued logic [Kle+52], (ii) a mapping of a popular partial modelling technique called MAVO [FSC12a] into 3-valued partial models, and (iii) a novel technique that rewrites the original graph constraints (to be matched over partial models) into two graph constraints to be matched on 3-valued partial models. One constraint will identify matches that *must* exist in all concretizations of the partial model while the other constraint will identify matches that *may* exist. Although the complexity of the pattern increases by the proposed rewrite, we can still rely upon efficient existing graph pattern matching techniques [Ber+11] that can scale up to millions of graph elements [Szá+17], which is a major practical benefit compared to SAT solvers. As a result, engineers can detect if concretizations of a partial model will (surely)

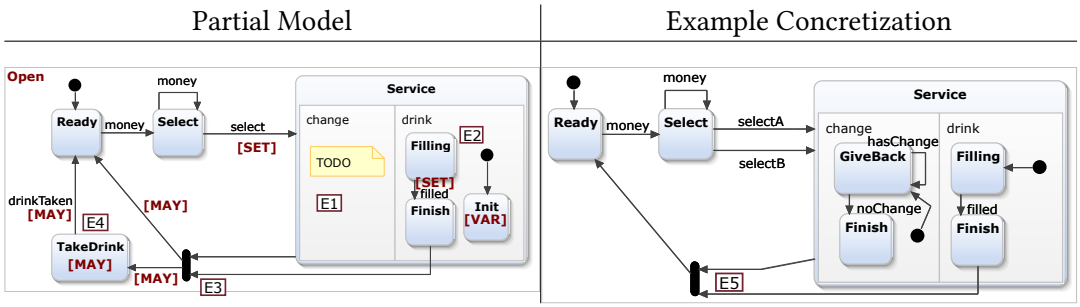


Figure 4.1: A partial statechart model and a sample concretization.

violate or may (potentially) violate a well-formedness constraint which helps them gradually to resolve uncertainty. Our approach is built on top of mainstream modeling technologies: partial models are represented in Eclipse Modeling Framework [Emf] annotated in accordance with [FSC12a], well-formedness constraints are captured as graph queries [Ber+11].

The rest of this chapter is structured as follows: Section 4.2 summarizes core modeling concepts of partial models and queries in the context of a motivating example. Section 4.3 provides an overview on 3-valued partial models with a graph constraint evaluation technique. Section 4.6 provides initial scalability evaluation of the approach, Section 4.7 overviews related approaches available in the literature. Finally, Section 4.8 concludes the paper. Proofs for this chapter are collected in Section A.3.

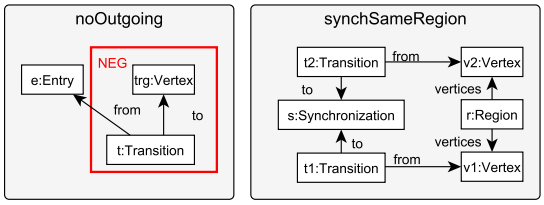
4.2 Motivating example: Validation of partial models

A partial Yakindu Statechart [Yak] model of a coffee machine is illustrated on the left part of Figure 4.1 together with a sample concrete model on the right. Initially, the machine starts in state Ready and after inserting coins by money events, a drink can be selected in state Select. While multiple concrete drink options may be available in the concrete model (like selectA and selectB), but in the partial model each one is represented by a generic select event. After the selection, the machine starts filling coffee, and gives back the change in state Service. The change management region is missing in the partial model, while a drink preparation region already contains some details. As the developer is uncertain about the initial state in this region, a placeholder state Init is created. In the partial model, it is undecided if it is required to wait until the previous drink is taken (in state TakeDrink), or the machine can enter its initial Ready state immediately. These uncertainties are captured by special annotations introduced in [FSC12a] such as may (elements can be omitted), var (elements that can be merged), set (representing sets of elements) or open (new elements can be added).

The Yakindu IDE checks several well-formedness rules on the statecharts. Here, we highlight two:

- C_1: Each region shall have exactly one entry, which has a transition to a state in the same region.
- C_1: The target and source states of a synchronization shall be contained in the same parent state.

Both constraints can be defined (e.g. in OCL [Ocl] or graph constraints [Ber+11]) and checked over complete models, but our paper focuses on detecting (potential and certain) conceptual errors (marked by E1-4 in Figure 4.1) in partial models.



$\mathsf{noOutgoing}(e) :=$
$\mathsf{Entry}(e) \wedge \neg\exists t, trg : \mathsf{from}(t, e) \wedge \mathsf{to}(t, trg)$
$\mathsf{synchSameRegion}(s) := \exists t_1, t_2, v_1, v_2, r :$
$\mathsf{Synchronization}(s) \wedge \mathsf{from}(t_1, v_1) \wedge \mathsf{to}(t_1, s) \wedge$
$\mathsf{from}(t_2, v_2) \wedge \mathsf{to}(t_2, s) \wedge \mathsf{vertices}(r, v_1) \wedge \mathsf{vertices}(r, v_2) \wedge$
$\neg t_1 \sim t_2$

Figure 4.2: Sample graph patterns for statecharts with their equivalent logic formula

E1 marks that an entry state is missing from region change, thus violating C_1. However, as the model is under construction, it can be repaired in a later stage. The other region (marked by E2) already contains an entry state, thus the WF constraint is currently satisfied, but it can potentially be violated in a future refinement by connecting it to a state located in a different region. E3 shows evidence of an invalid synchronization of parallel states Finish and its parent Service violating C_2. This error will be present in *all possible concretizations* (or completions) of the partial model, e.g. as E5 in Figure 4.1. Finally, E4 marks a possible error for synchronizing two target states that are not parallel (TakeDrink and Ready if all may elements are preserved).

To capture the erroneous case as a pattern match, the WF constraints C_1 and C_2 from the Yakindu documentation need to be reformulated as follows:

1. φ_{1a}: There is an entry state without an outgoing transition.
2. φ_{1b}: There is an entry state with a transition to a vertex in a different region.
3. φ_2: The target and source states of a synchronization are contained in different regions of the same parent state.

Graph patterns and the corresponding logic formulae for φ_{1a} and φ_2 are depicted in Figure 4.2. With a negative condition (marked by NEG) in `noOutgoing`, `Entry` states can be detected without any outgoing transitions. Moreover pattern `synchSameRegion` searches for synchronizations between vertices v_1 and v_2 which are in the same region.

4.3 Formalism of 3-valued partial models with interpreted equivalence and existence

Partial modeling [FSC12a; JLB11][J2] is a generic technique to introduce uncertainty into instance models. Semantically, one abstract partial models represents a range of possible instance models, which are called *concretizations*. During the development, the level of uncertainty can be gradually reduced by *refinements*, which results in partial model with less concretizations. Next, the thesis presents an extended logic structure formalism, which includes the interpretation of existence and equivalence. Next, we introduce a novel 3-valued partial modeling formalism which allows the explicit modeling of uncertainty. Finally, we give a method to evaluate graph predicates on (the representation of) partial models.

4.3.1 3-Valued Logic

In this section 3-valued logic [Kle+52; RSW04] is used to explicitly represent unspecified or unknown properties of the models with a third ½ truth value (beside 1 and 0 which means a value must be true

or false). During a refinement, ½ properties are gradually refined to either 0 or 1. This refinement is defined by an information ordering relation $X \sqsubseteq Y$, which specifies that either $X = ½$ and Y is refined to a more specific 1 or 0, or $X = Y$.

$$X \sqsubseteq Y := (X = ½) \vee (X = Y)$$

Information ordering $X \sqsubseteq Y$ has two important properties: first, if we know that $X = 1$ then it can be deduced that Y must be 1, and secondly, if $Y = 1$ then $X \geq ½$ (i.e. 1 or ½). Those two properties will be used to approximate possible values of a concrete model by checking the property on a partial model.

4.3.2 Signature of interpreted equivalence and existence

In Section 2.1 symbols $\sim$ and $\exists$ are introduced as part of logic. In this section, we extend the signature and by explicitly represent the existence (ε) and equivalence ($\sim$) of objects as relations. This will later allow us to represent richer structures, where the interpretation of ε will control the evaluation of $\exists v : \varphi$ and $\forall v : \varphi$ expressions, and the interpretation of $\sim$ will control the evaluation of $v_1 \sim v_2$ and $distinct(v_1, \ldots, v_n)$ expressions.

Definition 17 (Signature of Interpreted $\sim$ and ε Symbols) *Given a signature $\langle \Sigma; a \rangle$. An extended signature with interpreted equivalence and existence symbols means a signature (denoted with $\langle \Sigma^{\sim\varepsilon}, a^{\sim\varepsilon} \rangle$), where*

$$\Sigma^{\sim\varepsilon} := \Sigma \cup \{\sim, \varepsilon\} \text{ and } a^{\sim\varepsilon} := a, (\sim \mapsto 2), (\varepsilon \mapsto 1).$$

So, in a logic problem derived from a DSL specification the signature would be

$$\Sigma^{\sim\varepsilon} = \{\sim, \varepsilon, \mathsf{C}_1, \ldots, \mathsf{C}_n, \mathsf{R}_1, \ldots, \mathsf{R}_m, \mathsf{Q}_1, \ldots, \mathsf{Q}_o\},$$

where a unary predicate symbol C_i ($1 \leq i \leq n$) is defined for each *EClass*, and a binary predicate symbol R_j ($1 \leq j \leq m$) is derived for each *EReference* (or *EAttribute*) and Q_k for each graph pattern p_k ($1 \leq k \leq o$). Moreover, interpreted $\sim$ introduce an *equivalence relation* over objects, and ε predicate to denote the *existence relation* of an object in a given model.

4.3.3 Partial models with 3-valued logic and interpreted $\sim$ and ε symbols

This section introduces a partial modeling formalism based on interpreted 3-valued logic and interpreted $\sim$ and ε symbols.

Definition 18 (Partial Model) *A partial model over a signature $\langle \Sigma^{\sim\varepsilon}, a^{\sim\varepsilon} \rangle$ is a 3-valued logic structure $P = \langle O_P, \mathcal{I}_P \rangle$, where:*

- *O_P is the nonempty base set of individuals in the model (i.e. the symbolic objects)*
- *$\mathcal{I}_P(R) : O_P^{a(R)} \to \{0, 1, ½\}$ provides a 3-valued interpretation function for all $R \in \Sigma^{\sim\varepsilon}$.*

For the newly introduced $\sim$ and ε symbols, however, we assume some regularity properties:

Definition 19 (Regular Structures with Interpreted ~ and ε) *A logic structure $M = \langle O_M, \mathcal{I}_M \rangle$ defined over a signature $\langle \Sigma^{\sim_\varepsilon}, a^{\sim_\varepsilon} \rangle$ is regular, if $\mathcal{I}_M$ satisfies the following constraints:*

R1: $\forall o \in O_M : \mathcal{I}_M(\sim)(o, o) > 0$ *(~ is reflexive).*

R2: $\forall o_1, o_2 \in O_M : \mathcal{I}_M(\sim)(o_1, o_2) = \mathcal{I}_M(\sim)(o_2, o_1)$ *(~ is symmetric)*

R3: $\forall o_1, o_2 \in O_M : (o_1 \not\equiv o_2) \Rightarrow (\mathcal{I}_M(\sim)(o_1, o_2) < 1)$ *(~ respects transitivity)*

R4: $\forall o \in O_M : \mathcal{I}_M(\varepsilon)(o) > 0$ *(model does not contain non-existing objects)*

This implies that if there are no ½ values in a regular structure O_M, then ~ will be the same as ≡, and ε is always 1.

Theorem 2 (Unique Interpretation of ~ and ε in Regular Structures) *If a regular $M = \langle O_M, \mathcal{I}_M \rangle$ of $\langle \Sigma^{\sim_\varepsilon}, a^{\sim_\varepsilon} \rangle$ is 2-valued structure, then $\mathcal{I}_M(\sim)$ and $\mathcal{I}_M(\varepsilon)$ are defined uniquely up to isomorphism:*

$$\mathcal{I}_M(\sim)(o_1, o_2) := \begin{cases} 1 & \text{if } o_1 \equiv o_2 \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \mathcal{I}_M(\varepsilon)(o) := 1$$

Therefore, for regular structures without ½ values, there is a one-to-one mapping between regular logic structures of interpreted and uninterpreted equivalence and existence.

In the context of DSLs, 3-valued partial modeling enables advanced modeling features:

- *Uncertain Types.* $\mathcal{I}_P$ gives a 3-valued interpretation to each *EClass* symbol C_i in Σ: $\mathcal{I}_P(\mathsf{C}_i) : O_P \to \{1, 0, ½\}$, where an ½ value represents a case where it is unknown if an object has a type C or not.
- *Uncertain References.* $\mathcal{I}_P$ gives a 3-valued interpretation to each *EReference* symbol R_j in Σ: $\mathcal{I}_P(\mathsf{R}_j) : O_P \times O_P \to \{1, 0, ½\}$. An uncertain ½ value represent possible references.
- *Uncertain Equivalence.* $\mathcal{I}_P$ gives a 3-valued interpretation for the equivalence relation $\mathcal{I}_P(\sim) : O_P \times O_P \to \{1, 0, ½\}$. An uncertain ½ value relation between two objects means that the object might be equals and they can be potentially merged. For an object o where $o \sim_P o = ½$ it means that the object may represent multiple different objects, and can be split later on.
- *Uncertain Existence.* $\mathcal{I}_P$ gives a 3-valued interpretation for the existence relation $\mathcal{I}_P(\varepsilon) : O_P \to \{1, 0, ½\}$, where an ½ value represents objects that may be removed from the model.

Example 8. Figure 4.3 illustrates three partial models, where P_1 shows a submodel of the coffee machine from Figure 6.1. The objects are represented with nodes labelled with a unique name of its class. Solid and dashed lines represent references with 1 value and ½ references respectively, and missing edges represent 0 values. For example, in $P1$ state *Init* must be the target of transition $t1$, and Filling and Finish are potential targets. Uncertain ½ equivalences are also marked by dashed line with an = symbol. In $P1$ this means that state *Init* may be merged to states Filling and Finish, or $t2$ may be split into multiple objects between Filling and Finish.

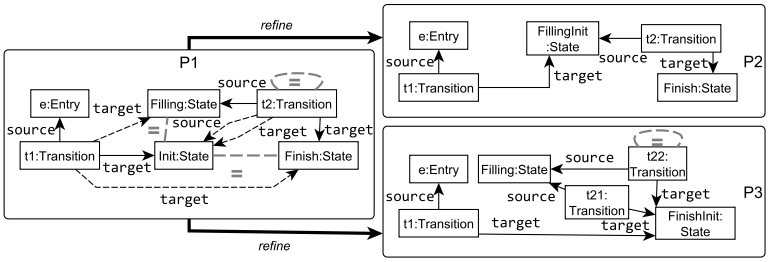


Figure 4.3: Example 3-valued partial model with refinements

4.3.4 Refinement and concretization

By resolving some uncertainty, a partial model P can be refined to a more concrete partial model Q.

Definition 20 (Refinement) *A refinement from $P = \langle O_P, \mathcal{I}_P \rangle$ to $Q = \langle O_Q, \mathcal{I}_Q \rangle$ over signature $\langle \Sigma^{\sim\varepsilon}; a^{\sim\varepsilon} \rangle$ is defined by a refinement morphism function $ref : O_P \to 2^{O_Q}$. A valid refinement ref respects the information order for all relation symbols $R \in \Sigma^{\sim\varepsilon}, n = a^{\sim\varepsilon}(R)$:*

$$\forall p_1, \ldots, p_n \in O_P, q_1 \in ref(p_1), \ldots, q_n \in ref(p_n) : \mathcal{I}_P(R)(p_1, \ldots p_n) \sqsubseteq \mathcal{I}_Q(R)(q_1, \ldots q_n)$$

Existing objects in P must have non-empty refinements, and all objects in Q are refined from an object in P:

$$\forall p \in O_P : [\mathcal{I}_P(\varepsilon)(p) = 1] \Rightarrow [ref(p) \neq \emptyset], \qquad \forall q \in O_Q : \exists p \in O_P : q \in ref(p)$$

If there is such a refinement morphism, then Q is the refinement of P (denoted as $P \sqsubseteq Q$).

In the context of DSLs *ref* respects the information order of type, reference, equivalence and existence predicates.

Example 9. Figure 4.3 illustrates two partial models $P2$ and $P3$ as possible refinements of $P1$. $P2$ represents a refinement scenario where Init and Filling are mapped to the same objects FillingInit, and the equivalence between the two objects are refined to 1 from ½. Simultaneously, the possible equivalence between FillingInit and Finish must be refined to 0 to satisfy the information order, because $[\![v_1 \sim v_2]\!]^{P1}_{v_1 \mapsto \text{Filling}, v_2 \mapsto \text{Finish}}$ was 0. In $P2$ the equivalence on Transition $t2$ is refined to 1 from ½, and mapped to a single object. $P3$ represents another valid refinement, where the Init and Finish are merged, and $t2$ is refined into two different objects $t21$ and $t22$, where $t22$ still represents a set of objects.

If a refinement $P \sqsubseteq Q$ resolves all uncertainty, and there are no ½ values in a partial model $Q = \langle O_Q, \mathcal{I}_Q \rangle$, and Q is regular, then Q represents a concrete (simple) instance model $M = \langle O_M, \mathcal{I}_M \rangle$ where $O_M = O_Q$ and $\mathcal{I}_M = \mathcal{I}_Q$, which is called concretization and marked with P.

Example 10. As $P2$ in Figure 4.3 does not contain any ½ values, it can be interpreted as concretization of $P1$.

4.3.5 Evaluating predicates on 3-valued partial models

The main goal of partial models is to evaluate graph predicates on them in order to check possible evaluations on *all* possible concretizations. Evaluating a predicate over a partial model may have multiple outcomes: a predicate *may* (½), *must* (1) or *cannot* (0) have a match depending on whether the partial model can possibly be concretized in a way to fulfill the condition of the predicate.

For three-valued partial models we use restricted syntax of first order logic predicates, that does not uses *implicit equivalence checks*. Informally, implicit equivalence means that a match has to substitute the same value for each occurrence of the same variable, which is similar to unification found in logic programming languages (like Prolog [Wie+12] or Datalog [CGT90]).

Definition 21 (Implicit equivalence check) *An expression φ contains implicit equivalence check of two different variable occurrences o_1 and o_2, if (1) o_1 and o_2 are both used in expressions other than equivalence checks ($\cdot \sim \cdot$), and*

(2/a) both o_1 and o_2 are free in φ, or

(2/b) o_1 and o_2 are bound by the same quantified expression.

This restriction is necessitated by the introduction of interpreted equivalence symbols ($\sim$). However, this restriction is only a formal one, because as discussed in Section A.2, implicit equivalence checks can be easily eliminated from any first order predicate by introducing new variables.

Example 11. For example in predicate noOutgoing(e) in Figure 5.2 the expression from(t, e) $\wedge$ to(t, trg) implicitly states that the value of the two t is the same. Our technique requires the explicit notation of equivalences, which can be achieved by rewriting each variable occurrence (except for those in equality constraints) to a new variable, and explicitly defining the equivalence between the new variables, by creating a logically equivalent expression. For example, the previous expression is changed to from(t_1, e) $\wedge$ to(t_2, trg) $\wedge$ $t_1 \sim t_2$.

Otherwise, we do not restrict the language of 3-valued expressions

Definition 22 (Syntax of 3-valued relational logic) *If $\varphi(v_1, \ldots, v_n)$ is a 3-valued logic expression over a signature $\langle \Sigma^{\sim\varepsilon}, a^{\sim\varepsilon} \rangle$, if:*

- *$\varphi(v_1, \ldots, v_n)$ is a (standard) logic expression over $\langle \Sigma^{\sim\varepsilon}, a^{\sim\varepsilon} \rangle$, and*
- *$\varphi(v_1, \ldots, v_n)$ is free of implicit equivalence checks.*

Semantically, a 3-valued predicate $\varphi(v_1, \ldots, v_n)$ of $\Sigma^{\sim\varepsilon}; a^{\sim\varepsilon}$ can be *directly evaluated* on a partial model M along a variable binding Z, which is a mapping $Z : \{v_1, \ldots, v_n\} \to O_M$ from variables to objects in M. However, in this case, the range of the interpretation function $[\![\cdot]\!]$ is $\{0, ½, 1\}$, and the interpretation of $\sim$, $\exists$ and $\forall$ expressions are controlled by the interpretation of $\sim$ and ε in $\mathcal{I}_M$.

Definition 23 (Semantics of 3-valued relational logic) *The semantics of a 3-valued predicate $\varphi(v_1, \ldots, v_n)$ over a 3-valued logic structure M over signature $\langle \Sigma^{\sim\varepsilon}; a^{\sim\varepsilon} \rangle$ and variable binding Z is denoted by $[\![\varphi(v_1, \ldots, v_n)]\!]^M_Z$, and defined as follows (differences to standard 2-valued semantics are highlighted with ▹):*

$$
\begin{aligned}
[\![1]\!]^M_Z &:= 1 \\
[\![0]\!]^M_Z &:= 0 \\
[\![R_i(v_i,\ldots,v_j)]\!]^M_Z &:= \mathcal{I}_M(R_i)(Z(v_i),\ldots,Z(v_j)) \\
\triangleright\quad [\![v_1 \sim v_2]\!]^M_Z &:= \mathcal{I}_M(\sim)(Z(v_1),Z(v_2)) \\
[\![distinct(v_1,\ldots,v_n)]\!]^M_Z &:= min\{[\![\neg(v_i \sim v_j)]\!]^M_Z : 1 \le i < j \le n\} \\
[\![\neg\varphi]\!]^M_Z &:= 1 - [\![\varphi]\!]^M_Z \\
[\![\varphi_1 \wedge \varphi_2]\!]^M_Z &:= min([\![\varphi_1]\!]^M_Z, [\![\varphi_2]\!]^M_Z) \\
[\![\varphi_1 \vee \varphi_2]\!]^M_Z &:= max([\![\varphi_1]\!]^M_Z, [\![\varphi_2]\!]^M_Z) \\
[\![\varphi_1 \Rightarrow \varphi_2]\!]^M_Z &:= [\![\neg\varphi_1 \vee \varphi_2]\!]^M_Z \\
[\![\varphi_1 \Leftrightarrow \varphi_2]\!]^M_Z &:= [\![(\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)]\!]^M_Z \\
\triangleright\quad [\![\exists v : \varphi]\!]^M_Z &:= max\{[\![\varepsilon(v) \wedge \varphi]\!]^M_{Z,v\mapsto o} : o \in O_M\} \\
\triangleright\quad [\![\forall v : \varphi]\!]^M_Z &:= min\{[\![\neg\varepsilon(v) \vee \varphi]\!]^M_{Z,v\mapsto o} : o \in O_M\} \\
\triangleright\quad [\![\varepsilon(v)]\!]^M_Z &:= \mathcal{I}_M(\varepsilon)(Z(v)) \\
[\![R^+(v_1,v_2)]\!]^M_Z &:= \begin{array}{l} max([\![R(v_1,v_2)]\!]^M_Z, \\ \quad max\{[\![\exists m_1,\ldots,m_n : R(v_1,m_1) \wedge \ldots \wedge R(m_n,v_2)]\!]^M_Z : n \in \mathbb{N}^+\}) \end{array}
\end{aligned}
$$

Representing equivalence and existence in a logic structure allows us to represent partial structures. However, in case of regular 2-valued structures, all predicates yields the same truth value.

Theorem 3 (Equivalent Semantics of Interpreted and Uninterpreted ~ and ε) *Let $M = \langle O_M, \mathcal{I}_M\rangle$ be a logic structure over $\langle\Sigma, a\rangle$, and a regular (3-valued) logic structure $M' = \langle O_{M'}, \mathcal{I}_{M'}\rangle$ over $\langle\Sigma^{\sim\varepsilon}, a^{\sim\varepsilon}\rangle$ with no ½ values, where $O_M = O_{M'}$ and $\mathcal{I}_M(s) = \mathcal{I}_{M'}(s)$ for each $s \in \Sigma$. Then, for each predicate $\varphi(v_1,\ldots,v_n)$ and variable binding $Z : \{v_1,\ldots,v_n\} \to O_M$ (which is also a valid binding M') the truth value of φ is equal:*

$$[\![\varphi(v_1,\ldots,v_n)]\!]^M_Z = [\![\varphi(v_1,\ldots,v_n)]\!]^{M'}_Z$$

Additionally, the truth value of the expression follows the $\sqsubseteq$ information ordering (proof in Section A.3) , which has two important consequences:

Theorem 4 (Forward concretization) *Let φ be a logic expression, and P and Q partial models where $P \sqsubseteq Q$. If $[\![\varphi]\!]^P = 1$, then $[\![\varphi]\!]^Q = 1$. Similarly, if $[\![\varphi]\!]^P = 0$, then $[\![\varphi]\!]^Q = 0$.*

Consequently, in each M concretization of a partial model, where $P \sqsubseteq M$, and $[\![\varphi]\!]^M = 1$ ($[\![\varphi]\!]^M = 0$) then $P \sqsubseteq M$.

Theorem 5 (Backward concretization) *Let φ be a logic expression, and P and Q partial models where $P \sqsubseteq Q$. If $[\![\varphi]\!]^Q = 1$, then $[\![\varphi]\!]^P \ge ½$. Similarly, if $[\![\varphi]\!]^Q = 0$ then $[\![\varphi]\!]^P \le ½$.*

Therefore, if an error predicate evaluates to 1, then it identifies an invalid partial model that cannot be repaired in any concretization. If it evaluates to ½ it highlights possible ways to inject errors. A 0 value can prove that an error cannot occur in the concretizations. This approach provides a conservative approximation for 1 and 0 values, where inaccurate cases are considered as ½. In other words, the match result is approximated in the direction of ½, which also includes the unknown cases. That is a safe compromise in many application areas such as model validation.

4.4 Rewriting predicates

In the previous section we defined the resolution rules for evaluating a graph predicate over a 3-valued partial model, which can result in three possible values: 1, ½, or 0. However, traditional query engines support only 2-valued pattern evaluation on 2-valued structures. Therefore, to use efficient graph pattern matching engines like introduced in [Ber+11], we introduce a predicates rewriting technique to calculate 3-valued predicate using two 2-valued predicates called *must* and *may* predicates, and combining them into 3 truth values. A predicate $must[\varphi]$ is a must predicate of φ, if $[\![must[\varphi]]\!]_Z^P = 1$ when $[\![\varphi]\!]_Z^P = 1$, otherwise $[\![must[\varphi]]\!]_Z^P = 0$. Similarly a predicate $may[\varphi]$ is a may predicate of φ, if $[\![may[\varphi]]\!]_Z^P = 1$ when $[\![\varphi]\!]_Z^P \geq ½$, otherwise $[\![may[\varphi]]\!]_Z^P = 0$.

4.4.1 Atomic must and may expressions

Atomic expressions $\mathsf{C}(v)$, $\mathsf{R}(v_1, v_2)$, $\varepsilon(v)$ and $v_1 \sim v_2$ are replaced with $(\varphi \geq ½)$ and $(\varphi = 1)$ 2-valued predicates in order to round ½ values up or down for maximizing the result for $may[\varphi]$ predicates, or to minimize the result for $must[\varphi]$ predicates. For relation symbols $R \in \Sigma^{\sim\varepsilon}$ $(R(v_1, \ldots, v_n) \geq ½)$ and $(R(v_1, \ldots, v_n) = 1)$ expressions are evaluated trivially on a partial model by numerical comparison:

$$R \in \Sigma^{\sim\varepsilon}, \in [\![R(v_1, \ldots, v_n) \geq ½]\!]_Z^P := \begin{cases} 1 & \text{if } [\![R(v_1, \ldots, v_n)]\!]_Z^P \geq ½ \\ 0 & \text{otherwise} \end{cases}$$

$$R \in \Sigma^{\sim\varepsilon}, [\![R(v_1, \ldots, v_n) = 1]\!]_Z^P := \begin{cases} 1 & \text{if } [\![R(v_1, \ldots, v_n)]\!]_Z^P = 1 \\ 0 & \text{otherwise} \end{cases}$$

4.4.2 Complex must and may expressions

Finally, *may* and *must* predicates are constructed from complex expression φ by recursively rewriting all subexpressions. It is important to highlight that the rewriting rule of the negated expressions $\neg\varphi$ (and $\varphi_1 \Rightarrow \varphi_2$) changes the modality of the inner expression from *may* to *must* and vica versa.

Definition 24 (Rewriting of *must*[·] and *may*[·] predicates) *Let $\varphi(v_1, \ldots, v_n)$ be a 3-valued logic predicate over signature $\langle \Sigma^{\sim\varepsilon}, a^{\sim\varepsilon} \rangle$. Then $must[\varphi(v_1, \ldots, v_n)]$ and $may[\varphi(v_1, \ldots, v_n)]$ is constructed according to the following rewriting rules. An atomic expression φ is replaced with numerical comparisons as follows:*

$$\begin{array}{rlrl} may[R(v_1, \ldots, v_n)] & ::= R(v_1, \ldots, v_n) \geq ½ & must[R(v_1, \ldots, v_n)] & ::= R(v_1, \ldots, v_n) = 1 \quad R \in \Sigma \\ may[\varepsilon(v)] & ::= \varepsilon(v) \geq ½ & must[\varepsilon(v)] & ::= \varepsilon(v) = 1 \\ may[v_1 \sim v_2] & ::= (v_1 \sim v_2) \geq ½ & must[v_1 \sim v_2] & ::= (v_1 \sim v_2) = 1 \end{array}$$

Next, complex expressions can be rewritten recursively:

$$\begin{aligned} may[distinct(v_1, \ldots, v_n)] &::= \neg \textstyle\bigwedge_{1 \leq i < j \leq n} must[v_1 \sim v_2] \\ must[distinct(v_1, \ldots, v_n)] &::= \neg \textstyle\bigwedge_{1 \leq i < j \leq n} may[v_1 \sim v_2] \end{aligned}$$

$$
\begin{aligned}
may[\neg\varphi] &::= \neg must[\varphi] & must[\neg\varphi] &::= \neg may[\varphi] \\
may[\varphi_1 \wedge \varphi_2] &::= may[\varphi_1] \wedge may[\varphi_2] & must[\varphi_1 \wedge \varphi_2] &::= must[\varphi_1] \wedge must[\varphi_2] \\
may[\varphi_1 \vee \varphi_2] &::= may[\varphi_1] \vee may[\varphi_2] & must[\varphi_1 \vee \varphi_2] &::= must[\varphi_1] \vee must[\varphi_2] \\
may[\varphi_1 \Rightarrow \varphi_2] &::= must[\varphi_1] \Rightarrow may[\varphi_2] & must[\varphi_1 \Rightarrow \varphi_2] &::= may[\varphi_1] \Rightarrow must[\varphi_2] \\
may[\varphi_1 \Leftrightarrow \varphi_2] &::= may[\varphi_1] \Leftrightarrow may[\varphi_2] & must[\varphi_1 \Leftrightarrow \varphi_2] &::= must[\varphi_1] \Leftrightarrow must[\varphi_2] \\
may[\exists v : \varphi] &::= \exists v : may[\varepsilon(v)] \wedge may[\varphi] & must[\exists v : \varphi] &::= \exists v : must[\varepsilon(v)] \wedge must[\varphi] \\
may[\forall v : \varphi] &::= \forall v : may[\neg\varepsilon(v)] \vee may[\varphi] & must[\forall v : \varphi] &::= \forall v : must[\neg\varepsilon(v)] \vee must[\varphi]
\end{aligned}
$$

$$
may[R^+(v_1, v_2)] ::= R^+_{may}(v_1, v_2), \text{ with } R_{may}(a, b) ::= may[R(a, b)]
$$

$$
must[R^+(v_1, v_2)] ::= R^+_{must}(v_1, v_2), \text{ with } R_{must}(a, b) ::= must[R(a, b)]
$$

Example 12. The following example illustrates the rewriting steps of an example graph previously introduced in Figure 5.2 into a may predicate.

original pattern: $\mathsf{noOutgoing}(e) := \mathsf{Entry}(e) \wedge \neg\exists t, trg : \mathsf{from}(t, e) \wedge \mathsf{to}(t, trg)$

may pattern: $may[\mathsf{noOutgoing}(e)] := may[\mathsf{Entry}(e)] \wedge may[\neg\exists t, trg : \mathsf{from}(t, e) \wedge \mathsf{to}(t, trg)] =$

$may[\mathsf{Entry}(e)] \wedge \neg\exists t, trg : must[\varepsilon(t)] \wedge must[\varepsilon(trg)] \wedge must[\mathsf{from}(t, e)] \wedge must[\mathsf{to}(t, trg)] =$
$(\mathsf{Entry}(e) \geq ½) \wedge \neg\exists t, trg : (\varepsilon(t) = 1) \wedge (\varepsilon(trg) = 1) \wedge (\mathsf{from}(t, e) = 1) \wedge (\mathsf{to}(t, trg) = 1)$

must[·] and *may*[·] predicates are constructed in a way that they always under- and over-approximate the truth-value of a predicate.

Theorem 6 (Relation of *must*[·] and *may*[·]) *For each 3-valued expression φ and partial model P the following statements hold:*

$$
[\![\varphi]\!]^P_Z = 1 \quad \Leftrightarrow \quad [\![must[\varphi]]\!]^P_Z = 1
$$

$$
[\![\varphi]\!]^P_Z \geq ½ \quad \Leftrightarrow \quad [\![may[\varphi]]\!]^P_Z = 1
$$

As a consequence:

$$
[\![may[\varphi]]\!]^P_Z \quad \Rightarrow \quad [\![must[\varphi]]\!]^P_Z
$$

Finally, *may*[φ] and *must*[φ] predicates are traditional 2-valued predicates, which can be combined to encode 3 possible truth values:

- If $[\![must[\varphi]]\!]^P_Z = 1$ and $[\![may[\varphi]]\!]^P_Z = 1$ then $[\![\varphi]\!]^P_Z = 1$
- If $[\![must[\varphi]]\!]^P_Z = 0$ and $[\![may[\varphi]]\!]^P_Z = 1$ then $[\![\varphi]\!]^P_Z = ½$
- If $[\![must[\varphi]]\!]^P_Z = 0$ and $[\![may[\varphi]]\!]^P_Z = 0$ then $[\![\varphi]\!]^P_Z = 0$

4.5 Transforming MAVO uncertainty to 3-valued partial models

MAVO uncertainty (which stands for May-Abstract-Variable-Open world) is a well-known and user-friendly partial modeling formalism [FSC12a; SFC12; SCG12] with several use-cases and tool support [BCE15]. In the following we present a mapping of MAVO partial models to 3-valued partial models, enabling the evaluation of graph constraints on it.

MAVO specifies partial models with a concrete instance model B called base model, and introduces uncertainty annotations on the objects and references of B. The transformation starts with the mapping of the base model, then the annotations are transformed separately.

Base Model. First, a logic structure $P = \langle O_P, \mathcal{I}_P \rangle$ of Σ is created from the base model B, where $O_P := O_B$, and $\mathcal{I}_P := \mathcal{I}_B$.

Mapping of May. In MAVO, *may* annotation marks uncertainty about the *existence* of an object or reference. For each object o marked by *may*, uncertain existence can be expressed by $\mathcal{I}_P(\varepsilon)(o) := ½$. For each reference R, it holds that if a link between o_1 and o_2 is marked by *may*, then $\mathcal{I}_P(\mathrm{R})(o_1, o_2) := ½$.

Mapping of Abstract. Abstract objects marked by *set* annotation marks uncertainty about the number of elements represented by an object. In 3-valued partiality, this can be represented by uncertain equivalence: for each object o marked by *set*, $\mathcal{I}_P(\sim)(o, o) := ½$.

Mapping of Variable. *var* annotation marks uncertainty about the distinctness of an object from another (which is not necessarily marked by *var*). In MAVO, objects with the same type are compatible for merging. Additionally, a *var* annotation implicitly specifies that the incoming and outgoing references of the compatible objects may be added to each other. For example, in the partial model in Figure 6.1, each incoming reference to *Init* may be redirected to *Filling* upon a merge. So for each object o_1 marked by *var*, and for each object o_2 with the same class predicate values ($\mathcal{I}_P(\mathrm{C})(o_1) = \mathcal{I}_P(\mathrm{C})(o_2)$ for each C) holds that:

- $\mathcal{I}_P(\sim)(o_1, o_2) := ½$, meaning that o_1 and o_2 may be merged
- for each incoming reference R from another object *src* to o_1 holds that: if $\mathcal{I}_P(\mathrm{R})(src, o_2) = 0$ then $\mathcal{I}_P(\mathrm{R})(src, o_2) := ½$. The outgoing references are handled similarly.

Mapping of Open. *open* is a *global property* of a MAVO partial model which marks uncertainty about the completeness of the model. If a model is *open*, then it can be extended by new objects and references in a refinement. Otherwise, only the existing elements can be resolved. In 3-valued partial models, this can be represented in the following way:

- a new object *other* is added to O_P, which represents the new objects.
- $\mathcal{I}_P(\sim)(other, other) = ½$, so *other* represent a set of objects.
- $\mathcal{I}_P(\varepsilon)(other) = ½$, so new objects are not necessarily added.
- for each class C: $\mathcal{I}_P(\mathrm{C})(other) = ½$, so *other* represents all types.
- for each reference R and each object pair o_1, o_2: if $\mathcal{I}_P(\mathrm{R})(o_1, o_2) = 0$, then $\mathcal{I}_P(\mathrm{R})(o_1, o_2) := ½$. Therefore new references can be added.

Cleaning of the Partial Model. During the translation of uncertainty annotations, new ½ references are added to the partial model without considering the structural constraints imposed by the target metamodel. Therefore, in order to exclude malformed instances from the analysis, when a ½ reference is added during the translation, (1) the ending types, (2) the multiplicity, (3) the containment hierarchy and (4) possible inverse relations are checked. If a possible reference would violate a structural constraint, then it is not added to P, so the precision of the approach can be increased by excluding invalid extensions only.

Local Search		#Obj = 157 #Ref= 604				#Obj=347 #Ref=1340				#Obj=1765 #Ref=6904			
Incremental		Open		Closed		Open		Closed		Open		Closed	
		Valid	Invalid	Valid	Invalid	Valid	Invalid	Valid	Invalid	Valid	Invalid	Valid	Invalid
Connected-Segments	must	1.40	1.36	1.39	1.37	1.79	1.96	1.73	2.07	28.41	68.97	27.96	68.71
	may	1.47	-	57.62	-	1.93	-	-	-	-	-	-	-
RouteSensor	must	1.40	1.30	1.35	1.45	1.72	1.92	1.74	2.20	25.28	70.70	26.79	70.23
	may	1.45	1.48	1.46	1.64	1.62	7.53	1.82	14.60	15.88	-	19.20	-
Semaphore-Neighbor	must	1.67	1.54	1.68	1.68	4.18	3.77	4.19	3.18	-	-	-	-
	may	1.49	-	46.93	-	2.80	-	-	-	-	-	-	-
SwitchSet	must	1.79	1.69	1.81	1.68	8.50	4.66	8.87	4.41	-	-	-	-
	may	1.88	8.86	4.14	-	8.62	-	117.79	-	-	-	-	-
Switch-Monitored	must	1.21	1.26	1.22	1.34	1.41	1.70	1.55	1.70	14.63	32.50	16.72	35.71
	may	1.13	1.11	1.06	1.12	1.27	1.30	1.31	1.30	12.55	31.30	12.46	29.22

Table 4.1: Evaluation time of validation patterns on partial models (in sec)

4.6 Scalability evaluation

We carried out an initial scalability evaluation[1] of our approach using 3 models (with 157, 347 and 1765 objects, respectively) and 5 queries available from the open TrainBenchmark [Szá+17]. We generated randomly assigned MAVO annotations for 5% of the model elements (e.g. with 7, 17, 88 uncertainties respectively). We evaluated the performance of (1) each graph query individually for (2) both may- and must-patterns (*may/must*) using (3) two pattern matching strategies (*incremental/local-search*) with (4) open world or closed world assumption (*open/closed*) after an optional (5) fault injection step (*valid/invalid*) to introduce some constraint violations. We measured the execution time for evaluating the queries in seconds with a timeout of 2 minutes using a laptop computer (CPU: Intel Core-i5-m310M, MEM: 16GB, OS: Windows 10 Pro). Our experiments were executed 10 times and the median of execution time is reported in Table 4.1 (table entries with a dash denote a timeout).

Our main observations can be summarized as follows:

- *Pattern matching over partial models is complex.* To position our experimental results, it is worth highlighting that most solutions of the Train Benchmark [Szá+17] evaluate *graph queries for regular models very fast* (scales up to millions of objects) for all these cases thus pattern matching over partial models must likely be in a different complexity class.

- *Fast inconsistency detection for must-matches.* The detection of a must-match over partial models is fast for both case of closed world and with open world assumption, especially, when using local-search graph pattern matching. It is also in line with previous observations in [J2] using SMT-solvers.

- *Scalable detection of may-matches with closed world assumption.* Our approach may identify potential inconsistencies (i.e. may-matches) over partial models with closed world semantics containing more than 1500 objects using incremental pattern matching. It is more than one order of magnitude increase compared to previous results reported in [Fam+13; FSC12a] using Alloy.

[1] A detailed description at `https://github.com/FTSRG/publication-pages/wiki/Graph-Constraint-Evaluation-over-Partial-Models-by-Constraint-Rewriting`.

- *Full match set of may-matches and open world is impractical.* As a negative result, calculating the full match set of graph patterns for may-matches *and* open world assumption frequently resulted in a timeout for models over 160 objects due to the excessively large size of the match set. For practical analysis, we believe that *open* annotation in MAVO should be restricted to be defined in the context of specific model elements.
- *Selection of graph pattern matching strategy.* In case of timeouts, we observed that large match sets caused problems for an incremental evaluation strategy while the lack of matches caused problems for local-search strategy.

4.7 Related work

Analysis of Uncertain/Partial Models. Uncertain models [FSC12a] provide user-friendly languages for defining partial models. Such models document semantic variation points generically by annotations on a regular instance model. Most analysis of uncertain models focuses on the generation of possible concrete models or the refinement of partial models. Potential concrete models compliant with an uncertain model can be synthesized by the Alloy Analyzer and its back-end SAT solvers [SFC12; SCG12], or refined by graph transformation rules [Sal+15].

Approaches [FSC12b; Fam+13] analyse possible matching and execution of model transformation rules on partial models by using a SAT solver (MathSAT4) or by automated graph approximation (referred to as "lifting"), or by graph query engines [C5]. The main difference is that their approach inspects possible partitions of a finite concrete model while we instead aim at (potentially infinite number of) extensions of a partial model. As a further difference, we use existing graph query engine instead of a SAT solver, which has a very positive effect on scalability (17 objects and 14 may annotations reported in [Fam+13] vs. over 1700 objects with 88 MAVO annotations in our paper).

Verification of Model Transformations. There are several formal methods that aim to evaluate graph patterns on abstract graph models (by either abstract interpretation [RD06; RZ12], or predicate abstraction [RSW04]) in order to detect possibly invalid concretizations. Those techniques typically employ techniques called pre-matching to create may-matches that are further analyzed. In [Rad+15] graph constraints are mapped to a type structure in order to differentiate objects that satisfy a specific predicate from objects that do not which could be used in our technique to further increase the precision of the matches.

In the previous cases an abstract graph similarly represents a range of possible models, and graph patterns are evaluated on abstract models to analyze their concretization. However, all of those technique expect a restricted structure in the abstract model, which is not available in partial models that are created by the developer.

Logic Solver Approaches. There are several approaches that map a (complete) initial instance model and WF constraints into a logic problem, which are solved by underlying CSP/SAT/SMT-solvers. In principle, the satisfaction of well-formedness constraints over a partial model (i.e. may- and must-matches) could be reformulated also using these techniques, although the same challenge has not been addressed so far. Complete frameworks with standalone specification languages include Formula [JLB11] (which uses Z3 SMT- solver [MB08]), Alloy [Jac02] (which relies on SAT solvers) and Clafer [Bak+16] or a combination of solvers [J2].

There are several approaches to validate models enriched with OCL constraints [GBR05] by relying upon different back-end logic-based approaches such as constraint logic programming [CCR07;

CCR08], SAT-based model finders (like Alloy) [SAB09; KHG11], first-order logic [BKS02] or higher-order logic [BW07]. As a common issue of such SAT/SMT-based approaches, the scalability is limited to small models.

4.8 Conclusion

In this chapter, I proposed a technique to evaluate graph queries capturing constraints over partial models. Since a partial model may be extended by the designer in future refinement steps, we defined may- and must-matches of a graph query correspondingly to denote potential and real violations of constraints. We also defined conservative approximations of may- and must-matches by rewriting of graph patterns in accordance with MAVO semantics.

Our initial scalability evaluation using the open Train Benchmark [Szá+17] shows that (1) finding real constraint violations over partial models is fast; (2) identifying potential inconsistencies with either open world or closed world assumption may scale for partial models with over 1500 model elements (which is one order of magnitude larger than reported in previous papers).

Although we motivated our work to check well-formedness constraints over uncertain models, our current results provide a key milestone for model generation, which aims at the automated generation of scalable and consistent domain-specific graph models (aka a graph-based model finder). Since the actual validation of complex graph constraints consumes significant amount of time in existing SAT/SMT-solvers, our current approach (which exploits efficient checking of graph constraints) nicely complements traditional logic solving techniques on complex graph structures.

Chapter 5

A Graph Solver for the Automated Generation of Models

5.1 Introduction

This chapter aims to provide a model generation technique to automatically generate well-formed graph models of a specification defined by (1) a metamodel (graph schema), (2) a set of well-formedness (WF) constraints expressed in first-order graph logic with transitive closure and optionally (3) an initial model fragment. Existing approaches like [Ana+10; JS07; KHG11; CCR08; BEC12; Bak+16][J2] map the instance generation problem of consistent graph models into logic solvers such as Alloy [TJ07; Mil+15], SMT-solvers [MB08], SAT-solvers or constraint solvers when the efficiency of graph model generation depends on the scalability and performance of back-end logic solvers, which primarily excel in *finding inconsistencies in complex specifications*. However, the generation of models as a side-effect of the proof construction is much less efficient.

In fact, these solvers guarantee neither scalability [C10] nor diversity [JSS13] when they need to generate well-formed graph instances of a specification – regardless of how smart the mapping is from a high-level graph model to the underlying logic solver. From a practical perspective, while the specification of complex industrial modeling tools may contain hundreds of classes (in their metamodel) and WF constraints, no existing model generation technique could derive a consistent graph that contains at least one object from each class.

We propose a novel automatic generation technique to derive consistent domain-specific graph models for specifications by exploiting and innovatively combining a multitude of advanced graph-based and core SAT-solving techniques.

1. We formulate model generation as a *refinement of partial models* [Sal+15][C5] where initial abstract model fragments are gradually refined and concretized during exploration.

2. We provide *partial model refinement rules* as *decision* and *unit propagation* steps by following core SAT-solving techniques.

3. We use incremental graph query evaluation of the VIATRA engine [Ujh+15] to efficiently *evaluate violations of constraints over partial models* during model generation [C5].

4. We integrate *shape analysis as state encoding* [RD06; Ren04; RSW04] for graphs to efficiently detect if two partial models should be treated as equivalent during exploration.

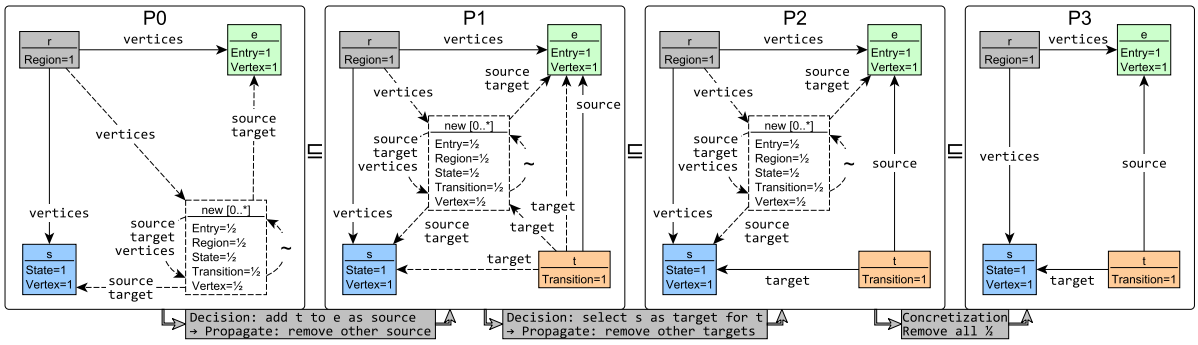


Figure 5.1: Sample partial models with uncertain elements and their refinement

5. We exploit rule-based design space exploration [HHV15] to *drive the generation process directly over graph shapes* using an objective function approximating the distance from a solution.

We *evaluate the scalability of our approach* using 6 tests sets of four domains (including industrial DSLs) and compare its performance with the well-known Alloy Analyzer [TJ07].

To our best knowledge, our framework is one of the first attempts to automatically generate consistent models by operating natively over (typed and attributed) graphs. Moreover, according to our scalability experiments, it is capable of *generating consistent graph models of 1-2 orders of magnitude larger* (with 500-6000 nodes) compared to models derived by Alloy and the generated model suite is also more diverse [C10]. As such, our graph solver can serve as a back-end where Alloy was used previously for model generation purposes in testing and benchmarking scenarios.

The chapter is structured as follows: Section 5.2 reviews the conceptual background for refinement of partial models. Our model generation framework is discussed in Section 5.3. Section 5.4 provides scalability measurements and compares our approach to Alloy, a popular solver-based model generator. Finally, Section 5.5 collects the related work and Section 5.6 concludes the chapter. This chapter is based on paper [C6] and book chapter [B14]. Proofs for this chapter are collected in Section A.4.

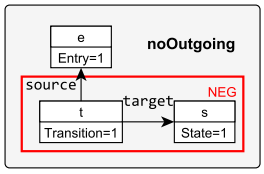
5.2 Modeling preliminaries

5.2.1 Partial models

Our model generation technique will be illustrated by automatically generating test inputs for Yakindu Statecharts Tools [Yak]. For the generation, we use 3-valued partial models as introduced in Chapter 4.

Example 13.

Four partial models are illustrated in Figure 5.1. As a notation guide, (1) the truth value of a *type predicate* is denoted by labels on nodes, where missing labels are treated as 0 values, while (2) *reference predicate* values 1 and ½ are represented by edges with solid and dashed lines (respectively), while missing edges between two objects represent 0 values for a predicate, (3) *existence predicate* values 1 and ½ are represented by nodes with solid and dashed borders, respectively, while objects with 0 existence values are simply not depicted. Finally, (4) uncertain ½ equivalences are marked by dashed line with an ~ symbol. Otherwise, each node represents a single, unique object (i.e. for



$\text{noOutgoing}(e) :=$
$\text{Entry}(e) \wedge \neg\exists t_1, t_2, e_1, s : \text{source}(t_1, e_1) \wedge \text{target}(t_2, s) \wedge e \sim e_1 \wedge t_1 \sim t_2$

Figure 5.2: Sample statechart WF constraint as graph query

all object o: $[\![o \sim o]\!] = 1$ and for all different objects o_1 and o_2: $[\![o_1 \sim o_2]\!] = 0$).

In P_0 (on the left side of Figure 5.1), object r is of type Region but not of type State: $[\![\text{Region}(v)]\!]^{P_0}_{v \mapsto r} = 1$ and $[\![\text{State}(r)]\!]^{P_0}_{v \mapsto r} = 0$. In case of object *new*, all type predicate are ½, which means that the object may represent any type of objects. In P_0 there is a certain vertices reference between r and s and r and e, and a possible reference between r and *new* or as a self-loop of *new*. Nodes r,e and s represent objects that must exist, and *new* represent possible objects which may exist or they might be removed later from the model. Node *new* may also represent multiple objects (note the self-loop edge ∼), which can later be refined into multiple distinct model elements.

During the refinement from partial model P to Q (denoted by $P \sqsubseteq Q$) uncertainty of a partial model is resolved gradually. Finally, if a 3-valued partial model P only contains 1 and 0 values (and no ½ values), and P is regular and there are no ∼ relations between different objects (i.e. all equivalent nodes are merged), then P represents a traditional *instance model*.

Example 14. Figure 5.1 illustrates two refinement steps from P_0 to P_2. In P_1 object *new* is split into two different objects: *new* and t of P_1, where $[\![v_1 \sim v_2]\!]^{P_0}_{v_1 \mapsto new, v_2 \mapsto new} = ½$ is refined to $[\![v_1 \sim v_2]\!]^{P_0}_{v_1 \mapsto new, v_2 \mapsto t} = 0$, $[\![v_1 \sim v_2]\!]^{P_0}_{v_1 \mapsto t, v_2 \mapsto t} = 1$ and $[\![v_1 \sim v_2]\!]^{P_0}_{v_1 \mapsto new, v_2 \mapsto new} = ½$. Additionally, $[\![\varepsilon(v)]\!]^{P_0}_{v \mapsto new} = ½$ is refined to $[\![\varepsilon(v)]\!]^{P_1}_{v \mapsto t} = 1$, and $[\![\text{Transition}(v)]\!]^{P_0}_{v \mapsto t} = ½$ is refined to $[\![\text{Transition}(v)]\!]^{P_1}_{v \mapsto t} = 1$, thus creating a new Transition object t, while all other ½ type predicates are refined to 0. Finally, source predicates are refined to 1 with e as target, and to 0 with all other objects as target.

In step P_1 to P_2, possible target predicates are refined to 1 with s as target object, and to 0 with e and *new* as target objects. Note that refinement step $P_1 \sqsubseteq P_2$ will illustrate the decision rule while $P_2 \sqsubseteq P_3$ will illustrate the unit propagation rule later in Section 5.3.3. P_3 is a concretization of P_2, which is also a refinement $P_2 \sqsubseteq P_3$.

5.2.2 Defining constraints over partial models

As illustrated in Section 3.2, domain-specific WF constraints are captured either by standard OCL constraints [Ocl] or by graph patterns (GP), which are translated to first order logic.

Example 15. The Yakindu documentation states several constraints for statecharts that can be formalized as graph predicates [C10]. For instance, constraint noOutgoing(e) in Figure 5.2 (depicted as a graph pattern and a graph predicate) detects an entry state e without an outgoing transition. The explicit use of equality constraints $e \sim e_1$ and $t_1 \sim t_2$ is responsible for performing a natural join operation over the edges as predicates. As a result, the same formula can be evaluated with both 2-valued and 3-valued interpretation.

Using these properties, we define a monotonous derivation sequence of valid partial models which (1) starts from the most abstract partial model where all constraints are evaluated to ½, which partial model (2) is gradually refined into more and more concrete partial models (with less number of predicates evaluating to ½). Refinement steps are continued until a concretized graph model of a designated scope eventually satisfies all WF constraints with 2-valued interpretation. Our graph generation approach will derive instance models along refinements. As such, partial models will gradually become more and more concrete after each refinement step which implies that checking WF constraints on partial models also becomes more precise. The practical benefit compared to consecutive calls to backend solvers [C10] is that the complex model finding problem can be divided into a sequence of small decisions while WF constraints can be checked (approximately) on intermediate solutions.

5.3 Automated graph generation

In this chapter, we propose a general and automated graph model generation approach which takes a domain specified by (I_1) a vocabulary Σ defined by a metamodel, and (I_2) a theorem $\mathcal{T}$ defined by a set of well-formedness constraints $\{\neg\varphi_1^{WF}, \ldots, \neg\varphi_n^{WF}\}$, and ($I_3$) a search scope (i.e. minimal and maximal number of nodes in a solution graph) and (O) generates a consistent (valid) concrete logic structure M where $M \models \mathcal{T}$ as output.

The model generation framework gradually refines partial models by rule-based design space exploration (DSE) [HHV15] into a well-formed instance model which complies to the metamodel and all WF constraints are satisfied, if such a concrete model exists within the given search scope. During exploration, our framework simultaneously operates on (concrete) instance models and WF constraints as well as (abstract) partial models and approximated WF constraints introduced in Chapter 3 by applying refinement rules. *Refinement rules* are defined as graph transformation rules [ERK99][C5] manipulating directly over partial models with 3-valued interpretations by concretizing a single atomic uncertain ½ value in each step.

Refinement rules are grouped into two categories: (1) *Decision rules* are derived from Σ to reduce the number of valid concretizations of a partial model (i.e. new information is added) while (2) *Unit propagation rules* are derived from $\mathcal{T}$ to propagate the consequences of previous decisions in order to simplify a solution candidate without excluding potential solutions.

Model generation is initiated from an *initial partial model* provided as input by an engineer, or from the *most abstract partial model* $P_0 = \langle O_{P_0}, \mathcal{I}_{P_0} \rangle$, where all predicates are unknown, i.e. (1) there is a single (abstract) object $O_{P_0} = \{new\}$; (2) $\mathcal{I}_{P_0}(\varepsilon)(new) = ½$ and $\mathcal{I}_{P_0}(\sim)(new, new) = ½$ thus this object may represent multiple possible objects of the concrete models; (3) for all class relation symbols $R \in \Sigma$ (i.e. all C classes, all R references and all A attributes in a DSL) $\mathcal{I}_{P_0}(R)(new, \ldots, new) = ½$.

5.3.1 Refinement operations for partial models

We define *refinement operations* Op to refine partial models by simultaneously growing the size of the models while reducing uncertainty in a way that each finite and consistent instance model is guaranteed to be derived in finite steps.

- *concretize*(p, val): if the atomic predicate p (which is either $\mathrm{C}_i(o)$, $\mathrm{R}_j(o_k, o_l)$ or $o_k \sim o_l$) has a ½ value in the pre-state partial model P, then it can be refined in the post-state Q to val which is either a 1 or 0 value. As an effect of the rule, the level of uncertainty will be reduced.

- *splitAndConnect*$(o, mode)$: if o is an object with $[\![o \sim o]\!]^P = ½$ in the pre-state, then a new object *new* is introduced in the post state by splitting o in accordance with the semantics defined by the following two modes:
  - *at-least-two*: $[\![new \sim new]\!]^Q = ½$, $[\![o \sim o]\!]^Q = ½$, $[\![new \sim o]\!]^Q = 0$, $[\![\varepsilon(new)]\!]^Q = 1$;
  - *at-most-two*: $[\![new \sim new]\!]^Q = 1$, $[\![o \sim o]\!]^Q = 1$, $[\![new \sim o]\!]^Q = ½$, $[\![\varepsilon(new)]\!]^Q = 1$;

  In each case, $\mathcal{O}_Q = \mathcal{O}_P \cup \{new\}$, and we copy all incoming and outgoing binary relations of o to *new* in Q by keeping their original values in P. Furthermore, all class predicates remain unaltered.

On the technical level, these refinement operations could be easily captured by means of algebraic graph transformation rules [Ehr+06] over typed graphs. However, for efficiency reasons, several elementary operations may need to be combined into compound rules. Thererfore, specifying refinement operations by graph transformation rules will be investigated in a future paper.

Example 16. Refinement $P_4 \sqsubseteq P_5$ (in Figure 5.3) is a result of applying refinement operation *splitAndConnect*$(o, mode)$ on object *new*3 and in *at-least-two* mode, splitting *new*3 to e and *new*4 copying all incoming and outgoing references. Next, in P_6, the type of object e is refined to Entry and Vertex, the ½ equivalence is refined to 1, and references incompatible with Entry or Vertex are refined to 0. Note that in P_6 it is ensured that Region r has an Entry, thus satisfying WF constraint *noEntryInRegion*. In P_7 the type of object *new*4 is refined to Transition, the incompatible references are removed similarly, but the ½ self equivalence remain unchanged. Therefore, in P_8 object *new*4 can split into two separate Transitions: $t1$ and $t2$ with the same source and target options. Refinement $P_8 \sqsubseteq P_9 \sqsubseteq P_{10}$ denotes a possible refinement path, where the target of $t1$ is directed to an Entry, thus violating WF constraint *incomingToEntry*. Note that this violation can be detected earlier in an unfinished partial model P_9. Refinement $P_{11} \sqsubseteq P_{12}$ denotes the consecutive application of six *concretize*(p, val) operations on uncertain source and target edges leading out of $t1$ and $t2$ in P_{11}, resulting in a valid model.

Note that these refinement operations may result in a partial model that is unsatisfiable. For instance, if all class predicates evaluate to 0 for an object o of the partial model P, i.e. $[\![\mathrm{C}(o)]\!]^P = 0$, then no instance models will correspond to it as most metamodeling techniques require that each element has exactly or at least one type. Similarly, if we violate the reflexivity of $\sim$, i.e. $[\![o \sim o]\!]^P = 0$, then the partial model cannot be concretized into a valid instance model. But at least, one can show that these refinement operations ensure a refinement relation between the partial models of its pre-state and post-state.

Theorem 7 (Refinement operations ensure refinement) *Let P be a partial model and op be a refinement operation. If Q is the partial model obtained by executing op on P (formally, $P \xrightarrow{op} Q$) then $P \sqsubseteq Q$.*

5.3.2 Consistency of model generation by refinement operations

Next we formulate and prove the consistency of model generation when it is carried out by a sequence of refinement steps from the most generic partial model P_0 using the previous refinement operations. We aim to show soundness (i.e. if a model is derivable along an open derivation sequence then it

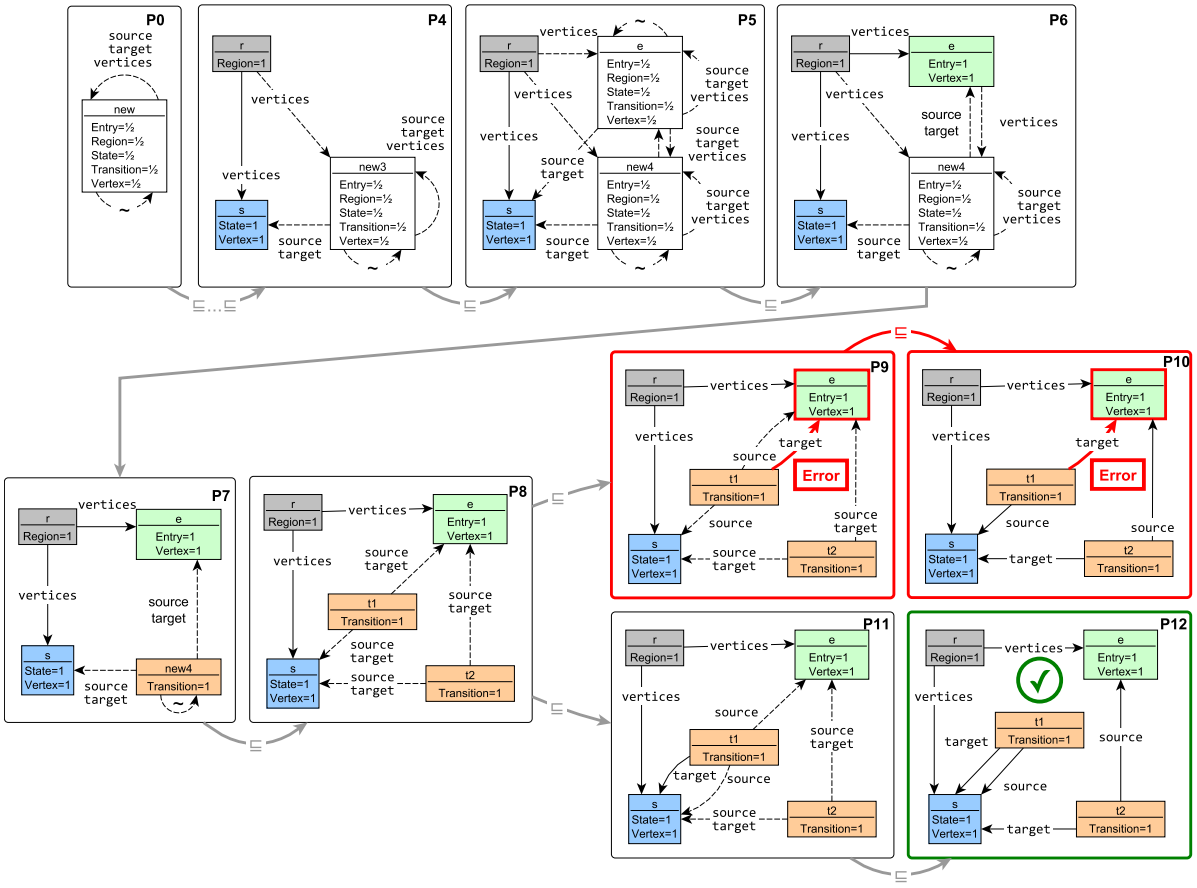


Figure 5.3: Refinement of partial models

is consistent), finite completeness (i.e. each finite consistent model can be derived along some open derivation sequence), and a concept of incrementality.

Many tableaux based theorem provers build on the concept of closed branches with a contradictory set of formulae. We adapt an analogous concept for closed derivation sequences over graph derivations in [Ehr+06]. Informally, refinement is not worth being continued as a WF constraint is surely violated due to a match of a graph pattern in case of a closed derivation sequence. Consequently, all consistent instance models will be derived along open derivation sequences.

Definition 25 (Closed vs. open derivation sequence) *A finite derivation sequence of refinement operations $op_1; \ldots; op_k$ leading from the most generic partial model P_0 to the partial model P_k (denoted as $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$) is closed wrt. a graph predicate φ if φ has a match in P_k, formally, $[\![\varphi]\!]^{P_k} = 1$.*
A derivation sequence is open if it is not closed, i.e. P_k is a partial model derived by a finite derivation sequence $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ with $[\![\varphi]\!]^{P_k} \leq ½$.

Note that a single match of φ makes a derivation sequence to be closed, while an open derivation sequence requires that $[\![\varphi]\!]^{P_k} \leq ½$ which, by definition, disallows a match with $[\![\varphi]\!]^{P_k} \leq 1$.

Example 17. Derivation sequence $P_0 \overset{\cdots}{\longrightarrow} P_9$ depicted in Figure 5.3 is closed for $\varphi = incomingToEntry(v)$ as the corresponding graph pattern has a match in P_9, i.e. $[\![incomingToEntry(v)]\!]^{P_9}_{v \mapsto e} = 1$. Therefore, P_{10} can be avoided as the same match would still exist. On the other hand, derivation sequence $P_0 \overset{\cdots}{\longrightarrow} P_{11}$ is open for $\varphi = incomingToEntry(v)$ as $incomingToEntry(v)$ is evaluated to ½ in all partial models $P_0, \ldots, P_{11}$.

As a consequence of approximations, an open derivation sequence ensures that any prefix of the same derivation sequence is also open.

Corollary 1 (Prefixes of open derivation sequences are open) *Let $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ be an open derivation sequence of refinement operations wrt. φ. Then for each $0 \leq i \leq k$, $[\![\varphi]\!]^{P_i} \leq ½$.*

The soundness of model generation informally states that if a concrete model M is derived along an open derivation sequence then M is consistent, i.e. no graph predicate of WF constraints has a match.

Corollary 2 (Soundness of model generation) *Let $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ be a finite and open derivation sequence of refinement operations wrt. φ. If P_k is a concrete instance model M (i.e. $P_k = M$) then M is consistent (i.e. $[\![\varphi]\!]^M = 0$).*

Effectively, once a concrete instance model M is reached during model generation along an open derivation sequence, checking the WF constraints on M by using traditional (2-valued) graph pattern matching techniques ensures the soundness of model generation as 3-valued and 2-valued evaluation of the same graph pattern should coincide.

Next, we show that any finite instance model can be derived by a finite derivation sequence.

Theorem 8 (Finiteness of model generation) *For any finite instance model M, there exists a finite derivation sequence $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ of refinement operations starting from the most generic partial model P_0 leading to $P_k = M$.*

Our completeness theorem states that any consistent instance model is derivable along open derivation sequences where no constraints are violated (under-approximation). Thus it allows to eliminate all derivation sequences where an graph predicate φ evaluates to 1 on any intermediate partial model P_i as such partial model cannot be further refined to a well-formed concrete instance model due to the properties of under-approximation. Moreover, a derivation sequence leading to a consistent model needs to be open wrt. all constraints, i.e. refinement can be terminated if any graph pattern has a match.

Theorem 9 (Completeness of model generation) *For any finite and consistent instance model M with $[\![\varphi]\!]^M = 0$, there exists a finite open derivation sequence $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ of refinement operations wrt. φ starting from the most generic partial model P_0 and leading to $P_k = M$.*

Unsurprisingly, graph model generation still remains undecidable in general as there is no guarantee that a derivation sequence leading to P_k where $[\![\varphi]\!]^{P_k} = ½$ can be refined later to a consistent instance model M. However, the graph model finding problem is decidable for a finite scope, which

is an a priori upper bound on the size of the model. Informally, since the size of partial models is gradually growing during refinement, we can stop if the size of a partial model exceeds the target scope or if a constraint is already violated.

Theorem 10 (Decidability of model generation in finite scope) *Given a graph predicate φ and a scope $n \in \mathbb{N}$, it is decidable to check if a concrete instance model M exists with $|O_M| \leq n$ where $[\![\varphi]\!]^M = 0$.*

This finite decidability theorem is analogous with formal guarantees provided by the Alloy Analyzer [TJ07] that is used by many mapping-based model generation approaches. Alloy aims to synthesize small counterexamples for a relational specification, while our refinement calculus provides the same for typed graphs without parallel edges for the given refinement operations.

However, our construction has extra benefits compared to Alloy (and other SAT-solver based techniques) when exceeding the target scope. First, all candidate partial models (with constraints evaluated to ½) derived up to a certain scope are reusable for finding consistent models of a larger scope, thus search can be incrementally continued. Moreover, if a constraint violation is found with a given scope, then no consistent models exist at all.

Corollary 3 (Incrementality of model generation) *Let us assume that no consistent models M^n exist for scope n, but there exists a larger consistent model M^m of size m (where $m > n$) with $[\![\varphi]\!]^{M^m} = 0$. Then M^m is derivable by a finite derivation sequence $P_i^n \xrightarrow{op_{i+1};\ldots;op_k} P_k^m$ where $P_k^m = M^m$ starting from a partial model P_i^n of size n.*

Corollary 4 (Completeness of refutation) *If all derivation sequences are closed for a given scope n, but no consistent model M^n exists for scope n for which $[\![\varphi]\!]^{M^n} = 0$, then no consistent models exist at all.*

While these theorems aim to establish the theoretical foundations of a model generator framework, it provides no direct practical insight on the exploration itself, i.e. how to efficiently provide derivation sequences that likely lead to consistent models. Nevertheless, we have an initial prototype implementation of such a model generator which is also used as part of the experimental evaluation.

5.3.3 Decision rules

Next, we define decision rules for the efficient generation of models.

Decision rules (see Figure 5.4) define various refinement operations to concretize information in partial models to construct possible solutions. They are derived from the vocabulary Σ of the metamodel, where each predicate symbol C_i and R_j represents a *Class* and *Reference*. In general, decision rules are responsible for (1) introducing new objects by splitting the abstract *new* object or (2) rewriting an ½ value to 1 as detailed by (the scheme of) four decision rule classes.

- Rule addRoot(C) selects a non-abstract class C (see the precondition in the left hand side) if no other roots have been created (denoted by NEG) to ensure that the model has a single root (as required by EMF). Its effect (prescribed by the right hand side) is to split the initial *new* object by creating a new *root* as an instance of C, which acts as a root element for the containment hierarchy and all self-loop references on *new* are extended to both objects.

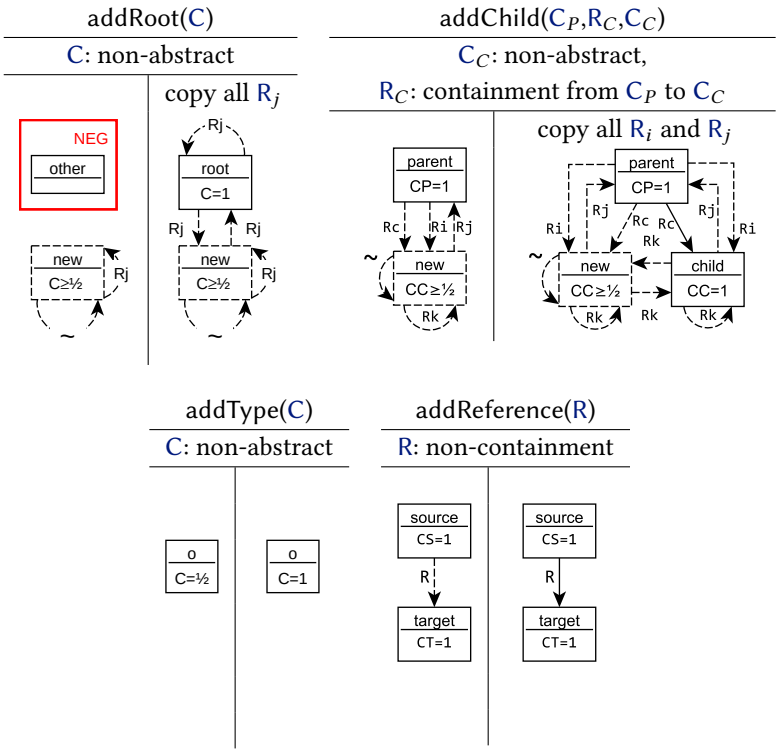


Figure 5.4: Decision rules for graph model generation

- Rule addChild(C_P,R_C,C_C) selects an existing *parent* object of type C_P, the *new* object with a non-abstract type C_C, and a containment reference R_C from *parent* to *new*. Upon execution, it splits *new* into a new object *child* of type C_C connected to *parent* via R_C, thus unfolding a new object along the containment hierarchy. During the unfolding step, all outgoing R_i, incoming R_j and loop R_k references of *new* are extended (copied) to *child*.
- Finally, two rules addType(C) and addReference(R) refine uncertain ½ classes and references in the partial model. In the latter case, the rule requires that the types of the endpoints are already fixed appropriately.

> **Example 18.** The refinement step $P_0 \sqsubseteq P_1$ in Figure 5.1 introduces a new object t by applying the *decision rule* addChild (of Figure 5.4), which changes ½ values of transition(t) and source(t, e) to 1.

5.3.4 Unit propagation rules

Unit propagation rules are responsible for refining unspecified elements in a partial model without excluding any valid solution to simplify the partial model propagating the consequences of previously applied decision rules. Unit propagation rules are applied repeatedly right after a decision rule is applied. They are derived from the structural constraints (1)-(6) introduced in Section 5.2.2. In general, unit propagation rules rewrite ½ elements to 0 (or 1) if a 1 (or 0) value would contradict to a constraint. Figure 5.5 illustrates (the scheme of) unit propagation rules used in this chapter.

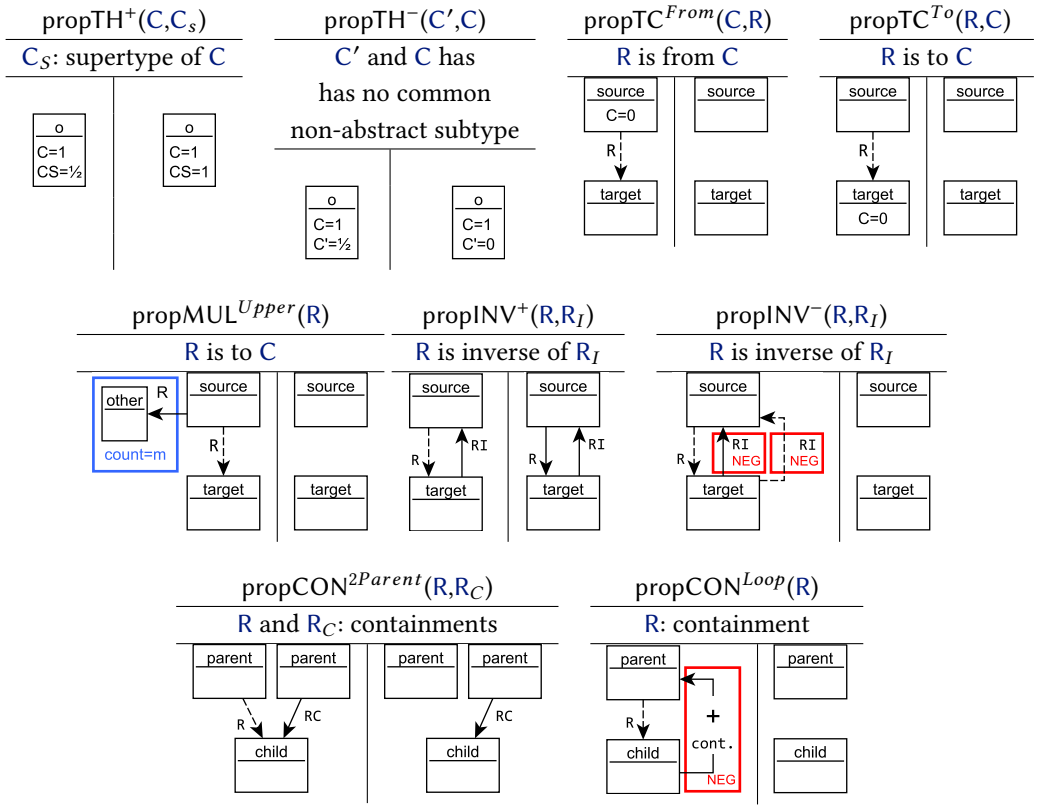


Figure 5.5: Unit propagation rules for graph model generation

- *Type hierarchy (TH)* is maintained by two rules: propTH^{+}(C,C$_s$) propagates a positive (1) type predicate of C to a supertype C$_s$; propTH^{-}(C′,C) rewrites a ½ type predicate value to 0 for type C if the object already has an incompatible type C′ where C and C′ do not have a common subclass (all classes are considered to be subclasses of themselves).
- *Type compliance (TC)* is checked by two rules: propTC^{From}(C,R) and propTC^{To}(R,C). Both rules remove possible references if the types of the reference end-points are incompatible.
- Rule propMUL^{Upper}(R) checks the *upper multiplicity (MUL)* of a reference, and removes all possible additional links if the upper limit is reached.
- The *Inverse (INV)* structural constraint is checked two rules: propINV^{+}(R,R$_I$) and propINV^{-}(R,R$_I$) which set a R predicate value to 1 or 0 if an inverse R$_I$ value is already set.
- To ensure *Containment hierarchy (CON)*, first decision rules enforce that each non-root object has a parent. Then, additional possible incoming containment references are removed by unit propagation rule propCON^{2Parent}(R,R$_C$). Finally, propCON^{Loop}(R) removes possible reference predicates that would create a loop in the containment hierarchy.

Decision and unit propagation rules are in close analogy with the DLL62 algorithm of traditional SAT-solvers [DLL62]: values of variables are graph elements in our case (instead of Boolean values),

complex graph predicates are evaluated (instead of conjunctive normal formulae), and the state space of graphs needs to be continuously stored during exploration (instead of a search tree).

> **Example 19.** Refinement step $P_1 \sqsubseteq P_2$ is a result of a *decision rule* addChild to set the ½ value of target(t, s) to 1 followed by a *unit propagation rule* propMUL^{Upper}(target) which aims to prevent the partial model from violating an upper multiplicity constraint by setting ½ values of target(t, e) and target(t, new) to 0 as a direct consequence of the previous decision rule.

As a preprocessing step, decision and propagation rules are derived. Moreover, *under-approximated (must) predicates* are synthesized from graph predicates in accordance with [C5] to detect unresolvable WF constraints early in a partial solution.

5.3.5 Exploration

During exploration (see Figure 5.6), refinement rules are repeatedly applied driven by an objective function and a rule selection strategy. As such, the size of partial models is continuously growing up to the designated scope, while the number of uncertainties and constraint violations in these partial models are decreasing to ensure that the process converges to consistent instance models. Now we discuss the steps of the model generation process.

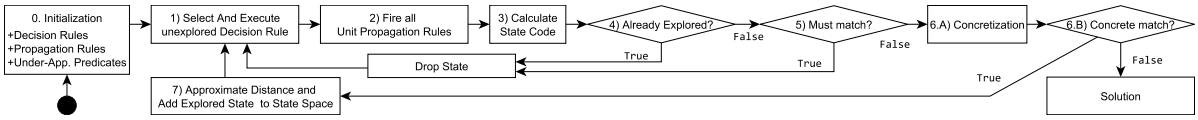


Figure 5.6: Exploration strategy for automated model generation

(1) After initializing the search with an initial partial model, *an unexplored decision rule is selected and applied* to derive a new (refined) partial model along a partial model refinement step (Section 4.3). The role of these refinement rules is in direct analogy with the decision steps in SAT solvers.

(2) After executing a decision rule, our framework *executes all possible unit propagation rules on the partial solution* to propagate the consequence of the decision, thus further refining the partial model. This step is again in direct analogy with SAT solvers, but it is carried out by incremental, change-driven model transformation rules [Ber+12] to improve efficiency.

(3) To prevent traversing the same (graph) state twice, a *state code* is calculated and stored for the new partial model by using graph isomorphism checks over *graph shapes* [Ren04]. Graph shapes abstract from node identities, but they efficiently identify if two graphs can be distinguished by the neighborhood (i.e. incoming and outgoing edges) of a node.

(4) Had the new state been already explored, the partial model is dropped and a new refinement rule is applied **(1)**.

(5) If a new partial model is reached, then our framework *checks if it satisfies all under-approximated (must) constraints* by partial evaluation of these constraints [C5] using an incremental graph query engine [Ujh+15]. If an under-approximated constraint is violated by the partial model

then an inconsistency is detected, thus the partial model can never be refined into a well-formed instance model so it can be dropped **(1)**.

(6) Next the partial model is *concretized into an instance model* by removing all uncertainties and all the original WF constraints are checked on this candidate model by the incremental graph query engine to detect inconsistencies. If no violations are found, then the instance model is stored as a solution, and the exploration may terminate (if designated model scope is reached) or continue to find other solutions **(1)**. Note that checking the original WF constraints on a concretized model guarantees the correctness of our solver.

(7) Finally, our framework approximates the *distance for a solution* by an objective function, adds the current partial model to the exploration path and continues refinement from a new unexplored decision refinement. For each partial model, this objective function is calculated as the sum of constraint violations and the number of missing objects wrt. a designated size.

For selecting the next match where a decision rule is to be applied, we use a combined exploration strategy with *best-first search* heuristic, *backtracking*, *backjumping* and *random restarts* in an advanced design space exploration framework [HHV15]. The search selects the best candidate wrt. the objective function, then it randomly fires (with uniform distribution) an enabled decision rule, subsequently, it fires all possible unit propagation rules. If no further decision rules can be applied, then it backtracks to continue along the previous partial model candidate. At each step, the exploration may backjump to the best model candidate found so far during the exploration. Finally, the search is occasionally restarted from a randomly chosen intermediate model candidate. Such a random restart is a common technique in SAT-solvers to avoid local optima.

Our framework operates directly over graph models and the exploration itself is driven on such a high-level. Thus, it *combines the advantages of multiple advanced graph-based techniques with core SAT-solving techniques* to tackle the scalability problems of existing mapping-based approaches:

- The approximated and the original of WF constraints are efficiently evaluated over partial models (in **(5)** and **(6.B)**) using incremental graph query evaluation techniques [Ujh+15].
- Refinement rules are divided into decision **(1)** and unit propagation rules **(2)** as facilitated by core SAT-solving algorithms.
- Isomorphic states are detected during exploration **(3)** by combining shape analysis techniques [Ren04; RSW04].
- Our framework has full control over the graph generation process **(1,7)** via rule-based DSE techniques [HHV15].

Example 20. Figure 5.3 depicts two sequences of partial model refinement steps deriving two instance models P_{10} and P_{12}:

$$P_0 \sqsubseteq \ldots \sqsubseteq P_4 \sqsubseteq P_5 \sqsubseteq P_6 \sqsubseteq P_7 \sqsubseteq P_8 \sqsubseteq P_9 \sqsubseteq P_{10} \text{ and}$$

$$P_0 \sqsubseteq \ldots \sqsubseteq P_4 \sqsubseteq P_5 \sqsubseteq P_6 \sqsubseteq P_7 \sqsubseteq P_8 \sqsubseteq P_{11} \sqsubseteq P_{12}.$$

Taking refinement step $P_4 \sqsubseteq P_5$ as an illustration, object $new3$ (in P_4) is refined into e and $new4$ (in P_5) where $[\![v_1 \sim v_2]\!]^{P_5}_{v_1 \mapsto e, v_2 \mapsto new4} = 0$ to represent two different objects in the concrete in-

stance models. Moreover, all incoming and outgoing edges of *new*3 are copied in e and *new*4. The final refinement step $P_{11} \sqsubseteq P_{12}$ concretizes uncertain source and target references into concrete references.

On the other hand derivation sequence $P_0 \sqsubseteq \ldots \sqsubseteq P_9$ depicted in Figure 5.3 leads to an unsatisfiable partial model, where well-formedness constraint is violated: $[\![incomingToEntry(v)]\!]^{P_9}_{v \mapsto e} = 1$ (it must exist). Therefore, P_{10} can be avoided by dropping P_9 as the same match would still exist.

5.3.6 Strengths and limitations

Our approach operates on connected sparse graphs with edges as relations (i.e. no edge identities and no parallel edges of a type) as underlying data model, which is less expressive than full relational algebra in case of Alloy. As a current technical limitation, our graph generation approach is showcased for EMF metamodels and models, which are widely used in industrial modeling tools, but it could be easily adapted to other graph formalisms. The expressive power of graph predicates used for capturing WF constraints is equivalent to first order logic with transitive closure over binary predicates.

Our solver efficiently handles complex structural graph constraints defined in first order logic with transitive closure. However, it includes only enumerations as attribute values but excludes strings, integers, etc. Such attribute values could be handled in the future by calling external solvers (e.g. SMT-solvers) during the exploration or as a post-processing step.

While the decision procedure of our graph solver provides stronger completeness guarantees than Alloy within a bounded scope, it does not provide an unsatisfiable core (i.e. minimal contradictory set of formulae) to highlight contradiction between WF constraints, which is supported by many SAT and SMT-solvers.

5.4 Experimental evaluation

We carried out an experimental evaluation of generating consistent instance models to address the following research questions:

RQ1 How does our graph solver scale (in time and model size) when generating consistent models of increasing size?

RQ2 How does our approach scale (in time and model size) compared to the widely used model finder Alloy [TJ07]?

RQ3 How do the different steps of the exploration influence performance of the graph solver?

Selected DSLs for evaluation. As model generation for DSLs still lacks systematically constructed performance benchmarks, we evaluated our approach in the context of 6 test sets of four different domains. First (1) a small File System (*FS*) example was taken from the Alloy documentation [All]. *Ecore*, the meta-metamodeling language of EMF [Emf], has been used as a case study by different approaches [Ana+10; Büt+12; KHG11; Soe+10][J2][C10] using Alloy as a background solver for model generation purposes. Our measurements also cover two DSLs that were developed in industrial projects, namely, (3) *Yakindu* [Yak] and (4) Functional Architecture Model (*FAM*) developed for avionics [Heg+16]. Due to their complexity, domains (3) and (4) are split into two cases: we generate models first with only general metamodel constraints (*w/o WF*), and then in the presence of extra WF constraints (*with WF*). In addition to their direct practical relevance, these DSLs have already been used in the context of model generation in numerous papers [Gon+12][J2][C10] in the past.

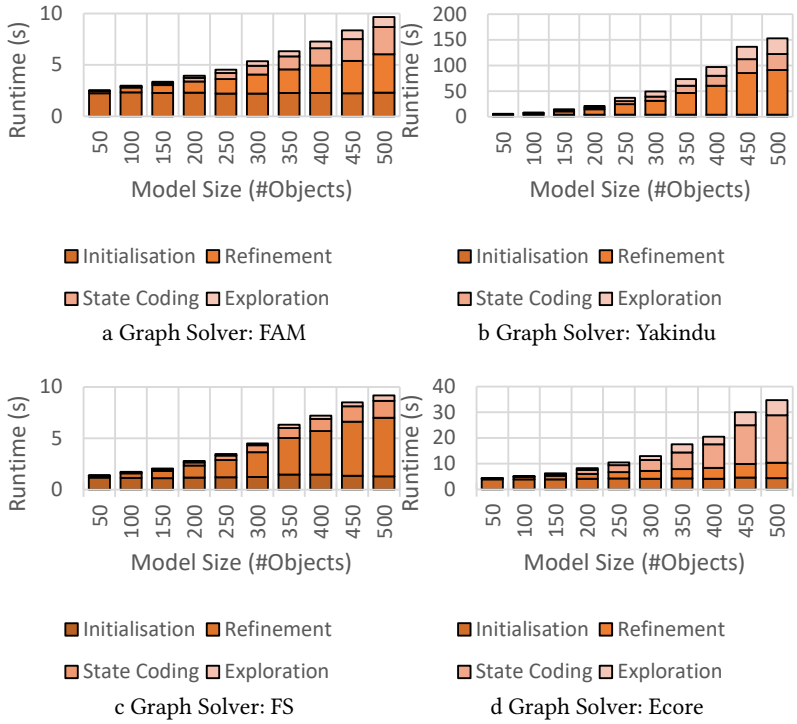


Figure 5.7: Distribution of generation time

	Problem size			Largest model (#Objects)		
	#Class	#Ref	#WF.	GS	Sat4J	MiniSat
FAM+WF	9	15	23	6250	58	61
FAM-WF	9	15	15	7000	87	92
Yak+WF	10	6	25	1000	–	–
Yak-WF	10	6	5	7250	86	90
FS	4	4	7	4750	87	89
Ecore	19	33	24	2000	38	41

Table 5.1: Comp.: Maximal model size

Benchmarking setup. To measure scalability, we set up a timeout of 3 minutes for each model generation run with increasing model size. For each measurement point, model generation was executed 30 times and the median of the runs were taken. To account for warm-up effects and memory handling of the Java 8 virtual machine, we added an extra 20 runs before the actual measurements and called the garbage collector explicitly between runs. As a baseline of comparison, Alloy Analyzer V4.2 (the latest stable version available at the Alloy download site) was used with two underlying SAT solver libraries: Sat4J (default in Alloy) and MiniSAT (recommended by Alloy). All measurements were executed on an average desktop computer[1] with 12 GB heap size.

Experimental results. For **RQ1** we evaluate the execution time of our approach for the four domains by increasing the target model size from 50 to 500 objects (with a step size of 50 new objects),

[1]CPU: Intel Core-i5-m310M, MEM: 16GB, OS: Windows 10 Pro.

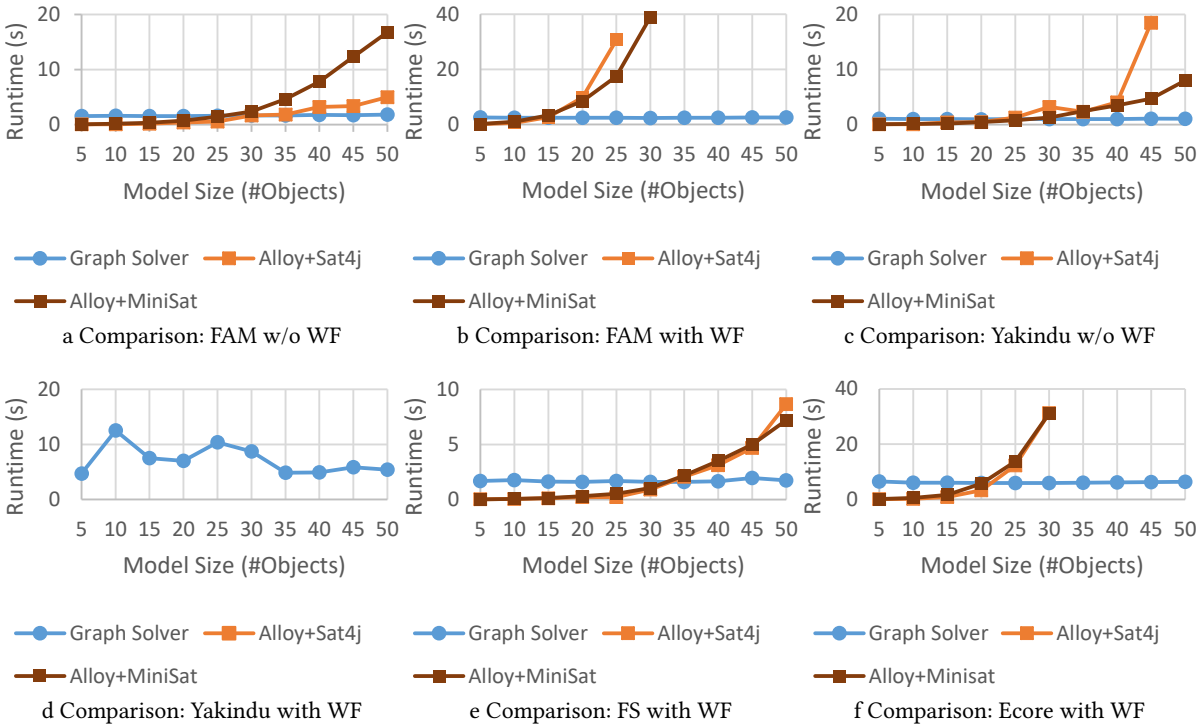


Figure 5.8: Runtime comparison with increasing model size

and measuring (in 5.7a– 5.7d) the total execution time. As a key observation, our approach is able to generate consistent models with 500 elements for all four domains within 10 seconds for FAM and FS, within 40 seconds for Ecore and 3 minutes for Yakindu. As a stress test, we also managed to generate even larger consistent models (1000 objects for Yakindu, 7000 objects for FAM, 4750 objects for FS and 2000 objects for Ecore, see Table 5.1) in 20 minutes (as a median of 10 measurements).

For **RQ2**, we compare model generation time of our Graph Solver with Alloy for small model sizes (from 5 to 50 objects, step size of 5 new objects) for the 6 test cases. As a baseline, we use a state-of-the-art EMF-to-Alloy mapping technique [Büt+12; KHG11; Soe+10][J2][C10] and tool to obtain Alloy specifications for the Yakindu, FAM and Ecore domains, and the original Alloy specification is used for FS. According to the results (see 5.8a–5.8f and Table 5.1), our approach scales much better as it generates models 1-2 orders of magnitude larger than Alloy could handle regardless of the back-end SAT solver which only had little impact on scalability. This is in line with previous measurements for Alloy in [J2][C10]). Alloy dominantly ran out of memory when mapping input specification into a SAT problem which results in over 6 million variables and several million clauses when aiming to generate a model with 40-90 objects.

Note that the Alloy Analyzer is not primarily targeted to generate models but to check the consistency of a relational specification within a given scope and synthesize small counterexamples. In fact, Alloy had a smaller runtime for very small models, thus the warm-up cost of our graph solver is higher. However, our graph solver is able to generate much larger graph models even for all four domains with similar consistency guarantees as Alloy.

For **RQ3**, we also measured (in 5.7a– 5.7d) how much time is spent in the different phases of model generation by our graph solver (see Figure 5.6) such as initialization, partial model refinement, state encoding and exploration. The preprocessing phase (1.5 seconds for FS, 4 seconds for Ecore, 2

seconds for FAM and 4 seconds for Yakindu) is a one-time penalty which is is proportional to the complexity of the metamodel and the WF constraints, thus we expect it to be negligible for model generation in case of other domains. Refinement is the dominant phase in the Yakindu and FS cases, while state encoding is dominant for Ecore. These results show that future research should primarily improve on refinement by providing a better transformation engine or refinement rules.

Quality of generated models. The quality of the generated models can be investigated from different aspects. To ensure *correctness*, all WF constraints were checked on each generated graph instance by using an external tool, the VIATRA graph query engine [Var+16]. To assess *diversity* when a sequence of models are generated by the proposed graph solver (within similar scope), each model is guaranteed to be non-isomorphic by the state codes (Step **(4)** in Figure 5.6) or by a distance metric [C11] in case of repeated calls to the solver, which offers increased diversity compared to Alloy [C11]. Systematically assessing the *realistic* nature of models is a more complex task [Szá+16] which necessitates to obtain a large set of real models authored by engineers. Our graph solver ensures that only enumeration values can be isolated nodes in a graph otherwise all graphs are connected by default, i.e. all regular nodes are arranged into a containment hierarchy. In order to assure connectedness, Alloy requires an extra constraint to capture this concept, thus by default, our solution appears to be more realistic. In addition, [Szá+16][B14] contain an in-depth investigation of realistic models for the Yakindu domain. All generated models are available at [Vs].

Threats to validity. In order to strengthen *internal validity*, our experiments include an extensive warm-up phase prior to the actual measurements to decrease the fluctuation of runtime results caused by the JVM (instead of the natural fluctuation of solver runtimes). We used default setups for running Alloy and our graph solver, i.e. no extra hints and performance optimizations were provided in the two approaches. Domain-specific fine tunings may improve scalability in some cases but it would simultaneously decrease the general-purpose nature of these solvers.

To address *external validity*, our measurements cover 6 test cases including 3 industrial domains (Ecore, Yakindu, FAM) with *complex structural WF constraints*, thus our experimental scalability results for our graph solver are likely generalizable to other domains of similar size and complexity within the limitations of Section 5.3.6. In case of simple WF constraints, the difference between the performance characteristics of Alloy and our graph solver may be smaller. Since the performance of Alloy depends on the backend SAT-solver, our measurements already included two state-of-the-art solvers (SAT4J and MiniSAT). Thus the large scalability difference in the size of generated models can likely be attributed to our graph solver.

Summary. Our graph solver provides a strong platform for generating consistent graph models which are 1-2 orders of magnitude larger (with similar or higher quality) than derived by mapping based approaches using Alloy with an underlying SAT-solver. Such a difference in scalability can only partly be dedicated to our conceptually different approach which combines several advanced graph techniques to improve performance instead of fine-tuning a mapping. However, it likely indicates fundamental shortcomings of existing mapping based approaches. Based on in-depth profiling we suspect that representing each potential edge between a pair of nodes as a separate Boolean variable blows up the state space for sparse graph with only linear number of edges. Moreover, SAT-solvers have major problems in evaluating complex predicates over larger graph models [B14] where graph query evaluation was particularly efficient [Ujh+15].

		Logic Solvers	Uncertain Models	Rule-Based	Iterative	Symbolic
Inputs	Partial Snapshot	+	++	-	+	-
	Local Constraints	+	-	+	+	+
	Global Constraints	+	-	-	+	+
Outputs	Metamodel	+	+	+	+	+
	Well-formed	+	-	-	+	+
	Scalable	-	-	++	+/-	-
	Decidability	-	+	+	-	+/-

Table 5.2: Comparison of related approaches

5.5 Related work

We compare our solution with existing model generation techniques with respect to the characteristics of *inputs* and *output results* in Table 6.1. As for *inputs*, the model generation can be (1) initiated from a *partial snapshot*. Additionally, an approach may support (2) *local* and (3) *global constraints* as WF constraints: a local constraint accesses only the attributes and the outgoing references of an object, while a global constraint specifies a complex structural pattern. Local constraints are frequently attached to objects (e.g. in UML class diagrams), while global constraints are widely used in DSLs. As *outputs*, the generated models may (i) be *metamodel-compliant* (ii) satisfy all *well-formedness* constraints of the language. We consider a technique (iii) *scalable* if there is no hard limit on the model size (as demonstrated in the respective papers). Finally, a model generation approach may be (iv) *decidable* which always terminates with a result. Our comparison excludes approaches like which do not guarantee metamodel- compliance of generated instance models.

Logic Solver Approaches. Several approaches map a model generation problem into a logic problem, which is solved by underlying SAT/SMT-solvers. Complete frameworks with standalone specification languages include Formula [JLB11] (which uses Z3 SMT- solver [MB08]), Alloy [Jac02] (which relies on SAT solvers like Sat4j [LBP10]) and Clafer [Bak+16] (using reasoners like Alloy).

There are several approaches aiming to validate standardized engineering models enriched with OCL constraints [GBR05] by relying upon different back-end logic-based approaches such as constraint logic programming [CCR07; CCR08; BC12], SAT-based model finders (like Alloy) [SAB09; Ana+10; Büt+12; KHG11; Soe+10][J2][C10], CSP solvers [Gon+12] first-order logic [BKS02], constructive query containment [Que+12], higher-order logic [BW07; GRR09], or rewriting logics [CE08]. Partial snapshots and WF constraints can be uniformly represented as constraints [J2]. Growing models are supported in [JS07][C10] for a limited set of constraints.

Scalability of all these approaches are limited to small models / counter-examples. Furthermore, these approaches are either a priori bounded (where the search space needs to be restricted explicitly) or they have decidability issues. As our approach is independent from the actual mapping of constraints to logic formulae, it could potentially be integrated with most of the above techniques by complementing or replacing the back-end solvers.

Uncertain Models. Partial models are similar to uncertain models, which offer a rich specification language [FSC12a; SC15] amenable to analysis. They a more user-friendly language compared to 3-valued interpretations, but without handling additional WF constraints. Potential concrete models compliant with an uncertain model can be synthesized by the Alloy Analyzer [SFC12], or refined by graph transformation rules [Sal+15]. Each concrete model is derived in a single step, thus their approach is not iterative like ours. Scalability analysis is omitted from these papers, but refinement of uncertain models is always decidable, thus termination is guaranteed.

Approaches like [Fam+13] analyze possible matches and executions of model transformation rules on partial models by using a SAT solver (MathSAT4) or by automated graph approximation (referred to as "lifting"), or by graph query engines with [C5]. As a key difference, our approach carries out model refinement while simultaneously evaluating graph query evaluation.

Rule-based Instance Generators. A different class of model generators relies on rule-based synthesis driven by randomized, statistical or metamodel coverage information for testing purposes [Bro+06; FSB04a; Ali+16]. Some approaches support the calculation of effective metamodels [Sen+09], but partial snapshots are excluded from input specifications. Moreover, WF constraints are restricted to local constraints evaluated on individual objects while global constraints of a DSL are not supported. On the positive side, these approaches guarantee the diversity of models and scale well in practice [SSB17; Ali+16].

Iterative Approaches. Iterative approaches generate models by multiple solver calls. In Chapter 6 models are generated in by calling Alloy in multiple steps, where each step extends the instance model by a few elements. This approach scaled up to 50 object in 45s for generating valid Yakindu Statecharts. An iterative approach is proposed *specifically for allocation problems* in [KJS11] based on Formula. Models are generated in two steps to increase diversity of results by first creating non-isomorphic submodels from an effective metamodel fragment followed by a problem-specific symmetry-breaking predicate [Cra+96] to ensures that no isomorphic models are generated twice while constraint checks are postponed to the final stage. An iterative, counter-example guided synthesis is proposed for higher-order logic formulae in [Mil+15], but the size of models is fixed and smaller than 50 objects.

Symbolic Model Generation Techniques. Certain techniques use abstract (or symbolic) graphs for analysis purposes. A tableau-based reasoning method is proposed for graph properties [SLO17; Pen08; ADW16], which automatically refine solutions based on well-formedness constraints, and handle state space in the form of a resolution tree. As a key difference, our approach refines possible solutions in the form of partial models, while [SLO17; Pen08] resolves the graph constraints to a concrete solution. Therefore our approach is able to exploit efficient graph query engines to evaluate partial solutions, while those techniques are demonstrated on small (< 10 objects) graphs or with no scalability evaluation at all.

Additionally, different approaches use abstract interpretation [RD06; Ren04], or predicate abstraction [RSW04] for partial modeling. In those approaches, concretization is used to materialize (typically small) counter-examples for designated safety properties in a graph transformation system. However, their focus is to support model checking of abstract graph transformation systems, which can evaluate complex trajectories, but do not scale in the size of the models.

5.6 Conclusion

We presented a novel graph solver to generate consistent models of a designated size from a specification defined by a metamodel and a set of WF constraints. Unlike existing approaches which map the model generation problem to logic solvers (dominantly SAT or SMT-solvers), we address the model generation problem of consistent instances directly over graphs by combining advanced graph-based techniques with core SAT-solving rules. Our approach is fully automated and available as an open source tool [Vs].

Our experimental evaluation carried out over three industrial domains confirmed that our solver is able to synthesize consistent graph models with over 500-6000 objects with similar quality guarantees as provided by the popular relational model finder Alloy. The scalability of our solver is 1-2 orders

of magnitude better than existing mapping based approaches using Alloy with a SAT-solver in the background. Such a difference in scalability likely indicates not only the benefits of our approach but also the inherent problems of mapping based model generation approaches deriving and solving a SAT problem. Thus our solver can serve as the output of mappings that previously used Alloy for model generation purposes. Altogether, our technique has the potential to be used in many testing scenarios including validation of large industrial DSLs, but its scalability is not yet sufficient for benchmarking purposes.

Chapter 6

Incremental Graph Model Generation with Logic Solvers

6.1 Introduction

The generation of sample instance models of Domain-Specific Language specifications has become an active research line due to its increasing industrial relevance for engineering complex modeling tools by using large metamodels (MM) and complex well-formedness (WF) constraints [Mou+09]. Existing approaches dominantly use either a logic solver or a rule-based instance generator in the background.

- *Model finding using logic solvers* [Jac02] (like SMT or SAT-solvers) is an effective technique (1) to identify inconsistencies of a DSL specification or (2) to generate well-formed sample instances of a DSL. This approach handles *complex global WF constraints* which necessitates to access and query several model elements during evaluation. Model generation for graph structures needs to satisfy complex structural global constraints (which is typical characteristic for DSLs), which restricts the direct use of logical numerical and constraint solvers despite the existence of various encodings of graph structures into logic formulae.

  As the metamodel of an industrial DSL may contain hundreds of model elements, any realistic instance model should be of similar size. Unfortunately, this cannot currently be achieved by a single direct call to the underlying solver [JLB11][J2], thus existing logic based model generators *fail to scale*. Furthermore, logic solvers tend to retrieve simple *unrealistic models* consisting of unconnected islands of model fragments and many isolated nodes, which is problematic in an industrial setting.

- *Rule-based instance generators* [Bro+06; FSB04a; Sen+09] are effective in generating larger model instances by independent modifications to the model by randomly applying mutation rules. Such a rule-based approach offers *better scalability* for complex DSLs. These approaches may incorporate *local WF constraints* which can be evaluated in the context of a single model element (or within its 1-context). However, they *fail to handle global WF constraints* which require to access and navigate along a complex network of model elements. Since constraint evaluation is typically the final step of the generation process, the synthesized models may violate several WF constraints of the DSL in an industrial setting.

In this chapter, I propose an iterative process for incrementally generating valid instance models by calling existing logic solvers as black-box components using various abstractions and approxima-

tions to improve overall scalability. (1) First, we apply enhanced metamodel pruning [Sen+09] and partial instance models [J2] to reduce the complexity of model generation subtasks and the retrieved partial solutions initiated in each step. (2) Then we propose an (over-)approximation technique for well-formedness constraints in order to interpret and evaluate them on partial (pruned) metamodels. (3) Finally, we define a workflow that incrementally generates a sequence of instance models by refining and extending partial models in multiple steps, where each step is an independent call to the underlying solver. We carried out experiments using the state-of-the-art Alloy Analyzer [Jac02] to assess the scalability of our approach.

Our approach increases the size of generated models by carefully controlling the information fed into and retrieved back from logic solvers in each step via abstractions. Each generated model (1) increases in size by only a handful number of elements, (2) satisfies all WF constraints (on a certain level of abstraction). The incremental derivation of the result set provides graceful degradation, i.e. if the back-end solver fails to synthesize models of size N (due to timeout), all previous model instances are still available. From a practical viewpoint, the DSL engineer can influence or assist the instance generation process by selecting the important fragment of the analyzed metamodel (so called *effective metamodel* [Bro+06]). This is also common practice for testing model transformations or code generators.

The chapter is structured as follows. Section 6.2 introduces some preliminaries for reviewing metamodels, constraints and partial snaptshots. The approach is presented in Section 6.3 followed by an experimental evaluation in Section 6.4. Related work is assessed in Section 6.5 while Section 6.6 concludes this chapter. The content of this chapter is based on paper [C10].

6.2 Preliminaries

In this section we present an overview of model generation with logic solvers with a running case study of Yakindu statecharts.

> **Example 21.** A sample statechart is illustrated in Figure 6.1. Yakindu provides two types of synchronization mechanisms: explicit synchronization nodes (marked as black rectangles) and event-based synchronization (i.e. raising and consuming events).
>
> Validation is crucial for domain-specific modelling tools to detect conceptual design flaws early and ensure that malformed models does not processed by tooling. Therefore missing validation rules are considered as bugs of the editor. While Yakindu is a stable modeling tool, it was still easy to develop model instances as corner cases which satisfy all (implemented) well-formedness constraints of the language but crashes the simulator or code generator due to synchronization issues. One of such problems is depicted in Figure 6.1 where (1) after 5 seconds a (2) timeout event raised in region timer, but (3) it cannot be accepted in wait in the simulator and in the generated code.
>
> Our goal is to systematically synthesize such model instances by using logic solvers in the background by mapping DSL specifications to a logic problem [JLB11][J2]. Such model generation approach usually takes three inputs: (1) a *metamodel of the domain*, (2) a set of *well-formedness constraints* of the language, and optionally (3) a *partial snapshot* serving as an initial seed which generated models need to contain.

In this chapter, we derive logic theory from the metamodel, the well-formedness constraints, derived features, and optionally from a partial snapshot as described in Chapter 3:

$$DSLMM \wedge WF \wedge DF \wedge PS.$$

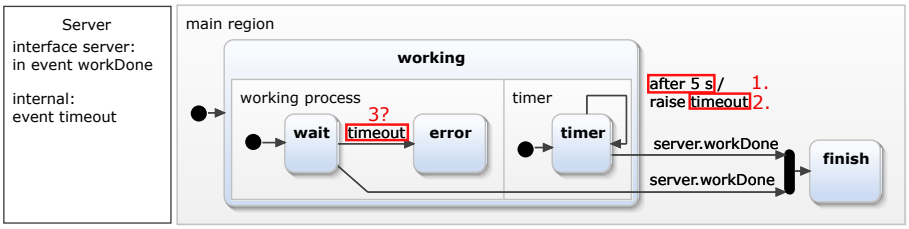


Figure 6.1: Example Yakindu statechart with synchronisations.

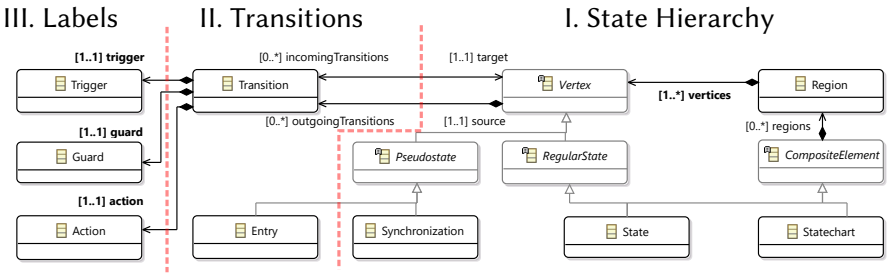


Figure 6.2: Metamodel extract from Yakindu state machines

Example 22. For this chapter, we use the same state-graph metamodel fragment as introduced in Chapter 3, but here, as illustrated in Figure 6.2, we partition it into three parts: State Hierarchy, Transitions and Labels The Yakindu documentation states several constraints for statecharts including the following ones regulating the use of synchronization states.

Source states of a synchronization have to be contained in different regions!

$\Phi_1 := \forall syn, s_1, s_2, t_1, t_2, r_1, r_2 :$
$(\text{Synchronization}(syn) \wedge \text{outgoing}(s_1, t_1) \wedge \text{outgoing}(s_2, t_2) \wedge \text{target}(t_1, syn) \wedge$
$\text{target}(t_2, syn) \wedge \text{vertices}(r_1, s_1) \wedge \text{vertices}(r_2, s_2) \wedge s_1 \neq s_2) \Rightarrow r_1 \nsim r_2$

- Source states of a synchronization are contained in the same parent state!

  $\Phi_2 := \forall syn, s_1, s_2, t_1, t_2, r_1, r_2 \exists p :$
  $(\text{Synchronization}(syn) \wedge \text{outgoing}(s_1, t_1) \wedge \text{outgoing}(s_2, t_2) \wedge \text{target}(t_1, syn) \wedge$
  $\text{target}(t_2, syn) \wedge \text{vertices}(r_1, s_1) \wedge \text{vertices}(r_2, s_2) \wedge s_1 \nsim s_2)$
  $\Rightarrow (\text{regions}(p, r_1) \wedge \text{regions}(p, r_2))$

- Target states of a synchronization have to be contained in different regions!

  $\Phi_3 := \forall syn, s_1, s_2, t_1, t_2, r_1, r_2 :$
  $(\text{Synchronization}(syn) \wedge \text{incoming}(s_1, t_1) \wedge \text{incoming}(s_2, t_2) \wedge \text{source}(t_1, syn) \wedge$
  $\text{source}(t_2, syn) \wedge \text{vertices}(r_1, s_1) \wedge \text{vertices}(r_2, s_2) \wedge s_1 \nsim s_2) \Rightarrow r_1 \nsim r_2$

- Target states of a synchronization are contained in the same parent state!

  $\Phi_4 := \forall syn, s_1, s_2, t_1, t_2, r_1, r_2 \exists p :$
  $(\text{Synchronization}(syn) \wedge \text{incoming}(s_1, t_1) \wedge \text{incoming}(s_2, t_2) \wedge \text{source}(t_1, syn) \wedge$
  $\text{source}(t_2, syn) \wedge \text{vertices}(r_1, s_1) \wedge \text{vertices}(r_2, s_2) \wedge s_1 \nsim s_2)$
  $\Rightarrow (\text{regions}(p, r_1) \wedge \text{regions}(p, r_2))$

- A synchronization shall have at least two incoming or outgoing transitions!
  $\Phi_5 := \forall syn : \text{Synchronization}(syn) \Rightarrow \exists t_1, t_2 : t_1 \nsim t_2 \wedge ($
  $(\text{incoming}(t_1, syn) \wedge \text{incoming}(t_2, syn)) \vee (\text{outgoing}(t_1, syn) \wedge \text{outgoing}(t_2, syn)))$

6.3 Incremental model generation by approximations

Despite the precise definition of logic formulae for our statechart language using existing mappings [J2], a major practical drawback is that a direct (single step) model generation only terminates for a limited sizes (which is very small in case of Z3 or Alloy). If we aim to improve scalability by omitting certain constraints, the synthesized models are no longer well-formed thus they cannot be fed into Yakindu as sample models.

To increase the size of synthesized models while still keeping them well-formed, we propose an incremental model generation approach (Section 6.3.3) by iterative calls to backend solvers exploiting two enabling techniques of metamodel pruning (Section 6.3.1) and constraint approximation (Section 6.3.2).

6.3.1 Metamodel pruning

Metamodel pruning [FSB04a; Sen+09] takes a metamodel MM as input and derives a simplified (pruned) metamodel $p(MM)$ as output by removing some *EClasses*, *EReferences* and *EAttributes*. When removing a class from a metamodel, we need to remove all subclasses, all attributes and incoming or outgoing references to obtain a consistent pruned metamodel.

- **Pruning**: $Cls_{p(MM)} \subseteq Cls_{MM}$, $Ref_{p(MM)} \subseteq Ref_{MM}$, $Attr_{p(MM)} \subseteq Attr_{MM}$ and $Data_{p(MM)} \subseteq Data_{MM}$.
- **EReference**: if $R \in Ref_{p(MM)}$ then:

$$src^{ref}_{p(MM)}(R) \in Cls_{p(MM)} \text{ and } trg^{ref}_{p(MM)}(R) \in Cls_{p(MM)}$$

$$\forall R \in Ref_{p(MM)} : src^{ref}_{p(MM)}(R) = src^{ref}_{MM}(R) \wedge trg^{ref}_{p(MM)}(R) = trg^{ref}_{MM}(R)$$

$$\forall R_1, R_2 \in Ref_{p(MM)} : inv_{p(MM)}(R_1, R_2) \Leftrightarrow inv_{MM}(R_1, R_2)$$

- **EAttributes**: if $A \in Attr_{p(MM)}$ then:

$$src^{attr}_{p(MM)}(A) \in Cls_{p(MM)} \text{ and } trg^{attr}_{p(MM)}(A) \in Data_{p(MM)}$$

$$\forall A \in Attr_{p(MM)} : src^{attr}_{p(MM)}(A) = src^{attr}_{MM}(A) \wedge trg^{attr}_{p(MM)}(A) = trg^{attr}_{MM}(A)$$

- **EClasses**: if $C \in Cls_{p(MM)}$ then:

$$\forall S \in Cls_{MM} : sup_{MM}(C, S) \Rightarrow (S \in Cls_{p(MM)} \wedge sup_{p(MM)}(C, S))$$

$$\forall C \in Cls_{p(MM)} : abs_{MM}(C) \Leftrightarrow abs_{p(MM)}(C)$$

- Other functions mul^{min}_{MM}, mul^{max}_{MM} and $cont_{MM}$ in $p(MM)$ remain the same with limited domain.

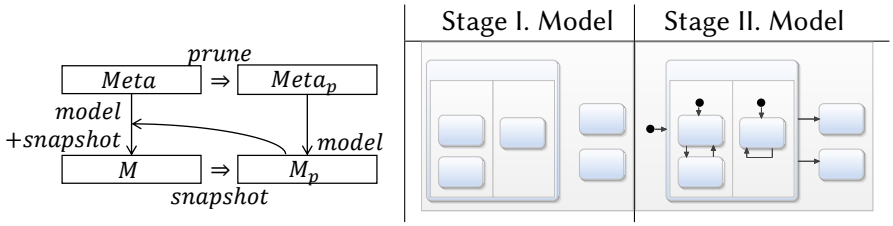


Figure 6.3: Metamodel pruning with overapproximation

Example 23. We prune our statechart metamodel in two phases (see the slices in Figure 6.2): classes Trigger, Guard and Action are omitted together with incoming references (Stage II), and then classes Transition, Pseudostate, Entry and Synchronization are removed (Stage I).

By using metamodel pruning, we first aim to generate valid instance models for the pruned metamodel and then extend them to valid instance models of the original larger metamodel. For that purpose, we exploit a property we call the *overapproximation property of metamodel pruning* (see Figure 6.3), which ensures that if there exist a valid instance model M for a metamodel MM (formally, $M \models MM$) then there exists a valid instance model M_P for the pruned metamodel $p(MM)$ (formally, $M_P \models Meta_P$) such that M_P is a partial snapshot of M ($M_P \subseteq M$). Consequently, if a model generation problem is unsatisfiable for the pruned metamodel, then it remains unsatisfiable for the larger metamodel. However, we may derive a pruned instance model M_P which cannot be completed in the full metamodel MM, which is called a *false positive*.

Example 24. The statechart model in the middle of Figure 6.3 corresponds to the pruned metamodel after Stage II. In our example, it can be extended by adding transitions and entry states to the model illustrated in the right side of Figure 6.3, which now corresponds to the pruned metamodel of Stage I.

6.3.2 Constraint pruning and approximation

When removing certain metamodel elements by pruning, related structural constraints (such as multiplicity, inverse, etc.) of $p(MM)$ are be automatically removed, which trivially fulfills the overapproximation property. Moreover, $p(MM)$ is defined in a limited signature $\langle p(\Sigma), p(a)\rangle$

$$MM \Rightarrow p(MM), p(\Sigma) \subseteq \Sigma$$

However, the treatment of additional well- formedness constraints needs special care since simple automated removal would significantly increase the rate of false positives in a later phase of model generation to such an extent that no intermediate models can be extended to a valid final model.

Based on some first-order logic representation of the constraints (derived e.g. in accordance with Chapter 3 and [C9][J2]), we propose to maintain approximated versions of constraint sets during metamodel pruning. In order to investigate the interrelations of constraints, we assume that logical consequences of a constraint set can be derived manually by experts or automatically by theorem provers [KV09]. Given a DSL specification with a metamodel MM and a set of WF constraints $WF = \{\varphi_1, \ldots, \varphi_n\}$, let φ be a formula derived as a theorem $MM, WF \models \varphi$.

Now an *overapproximation* of formula φ over metamodel MM for a pruned metamodel $p(MM)$ is a formula $p(\varphi)$ such that (1) $\varphi \Rightarrow p(\varphi)$, (2) φ_P contains symbols only from $p(\Sigma)$. The details of approximation are illustrated in the followinf definition where R denotes a relation symbol derived for class or reference predicates in accordance with the metamodel. While more precise approximations can possibly be defined in the future, the current approximation is logically correct as if a model generation problem is unsatisfiable for an approximated set of constraints (over the pruned metamodel) then it remains unsatisfiable for the original set of constraints.

Definition 26 (Over- and under-approximation by pruning) *Let $p(MM)$ denote the pruned theory of of MM, which uses the pruned signature $\langle p(\Sigma), p(a)\rangle$ of $\langle \Sigma, a\rangle$. Then the over- and under-approximation of a predicate $\varphi(v_1, \ldots, v_n)$ of $\langle \Sigma, a\rangle$ is a predicate of $\langle p(\Sigma), p(a)\rangle$ denoted with $p^O[\varphi(v_1, \ldots, v_n)]$ and $p^U[\varphi(v_1, \ldots, v_n)]$, and calculated as follows:*

$$p^U[R(v_1,\ldots,v_n)] ::= \begin{cases} R(v_1,\ldots,v_n) & \text{if } R \in p(\Sigma) \\ 0 & \text{otherwise} \end{cases}$$

$$p^O[R(v_1,\ldots,v_n)] ::= \begin{cases} R(v_1,\ldots,v_n) & \text{if } R \in p(\Sigma) \\ 1 & \text{otherwise} \end{cases}$$

$$p^U[v_1 \sim v_2] ::= v_1 \sim v_2 \qquad p^O[v_1 \sim v_2] ::= v_1 \sim v_2$$

$$p^U[distinct(v_1,\ldots,v_n)] ::= distinct(v_1,\ldots,v_n) \qquad p^O[distinct(v_1,\ldots,v_n)] ::= distinct(v_1,\ldots,v_n)$$

$$p^U[\neg\varphi] ::= \neg p^O[\varphi] \qquad p^O[\neg\varphi] ::= \neg p^U[\varphi]$$

$$p^U[\varphi_1 \wedge \varphi_2] ::= p^U[\varphi_1] \wedge p^U[\varphi_2] \qquad p^O[\varphi_1 \wedge \varphi_2] ::= p^O[\varphi_1] \wedge p^O[\varphi_2]$$

$$p^U[\varphi_1 \vee \varphi_2] ::= p^U[\varphi_1] \vee p^U[\varphi_2] \qquad p^O[\varphi_1 \vee \varphi_2] ::= p^O[\varphi_1] \vee p^O[\varphi_2]$$

$$p^U[\varphi_1 \Rightarrow \varphi_2] ::= p^U[\varphi_1] \Rightarrow p^O[\varphi_2] \qquad p^O[\varphi_1 \Rightarrow \varphi_2] ::= p^U[\varphi_1] \Rightarrow p^O[\varphi_2]$$

$$p^U[\varphi_1 \Leftrightarrow \varphi_2] ::= p^U[\varphi_1] \Leftrightarrow p^U[\varphi_2] \qquad p^O[\varphi_1 \Leftrightarrow \varphi_2] ::= p^O[\varphi_1] \Leftrightarrow p^O[\varphi_2]$$

$$p^U[\exists v : \varphi] ::= \exists v : p^U[\varphi] \qquad p^O[\exists v : \varphi] ::= \exists v : p^O[\varphi]$$

$$p^U[\forall v : \varphi] ::= \forall v : p^U[\varphi] \qquad p^O[\forall v : \varphi] ::= \forall v : p^O[\varphi]$$

$$p^U[R^+(v_1,v_2)] ::= \begin{cases} R^+(v_1,\ldots,v_n) & \text{if } R \in p(\Sigma) \\ 0 & \text{otherwise} \end{cases}$$

$$p^O[R^+(v_1,v_2)] ::= \begin{cases} R^+(v_1,\ldots,v_n) & \text{if } R \in p(\Sigma) \\ 1 & \text{otherwise} \end{cases}$$

Example 25. Based on the set of WF constraints $\{\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5\}$ defined in Section 6.2, a prover can derive the following formula as a theorem over the metamodel of Stage II: $\varphi_{syncout} \vee \varphi_{syncin}$, where $\varphi_1, \varphi_5 \models \varphi_{syncout} \vee \varphi_{syncin}$. The generated theorem $\varphi_{syncout}$ (and φ_{syncin}) restricts the number of outgoing (ingoing) transitions from (to) a synchronization as follows:

$\varphi_{syncout} = \forall syn \exists \underline{t_1}, \underline{t_2}, s_1, r_1, r_2, p : \mathsf{Synchron}(syn) \Rightarrow$
$(\underline{\mathsf{outgoing}(syn, t_1)} \wedge \underline{\mathsf{target}(t_1, s_1)} \wedge \underline{\mathsf{outgoing}(syn, t_2)} \wedge \underline{\mathsf{target}(t_2, s_2)} \wedge s_1 \nsim s_2 \wedge$
$\mathsf{vertices}(r_1, s_1) \wedge \mathsf{vertices}(r2, s2) \wedge r_1 \nsim r_2 \wedge \mathsf{regions}(p, r1) \wedge \mathsf{regions}(p, r2))$

The variables and relations approximated in this phase are underlined: in Stage I the generation is restricted to the model by omitting transitions. The result of overapproximation states that if a model contains a synchronization, then needs to contain at least two regions:

$\varphi^O_{syncout} \vee \varphi^O_{syncin} = \forall syn \exists s_1, r_1, r_2, p : \mathsf{Synchron}(syn) \Rightarrow$

$(s_1 \nsim s_2 \wedge \mathsf{vertices}(r_1, s_1) \wedge \mathsf{vertices}(r2, s2) \wedge r_1 \nsim r_2 \wedge \mathsf{regions}(p, r1) \wedge \mathsf{regions}(p, r2))$

Applying the approximation rules directly on $\{\varphi_1, \varphi_5\}$ would lead to $\varphi_1^O : \mathit{true}$ and $\varphi_5^O : \mathit{true}$. These constraints are too coarse overapproximations providing no useful information to the model generator at this phase.

Over- and under-approximation by pruning are proper approximations:

Theorem 11 (Relation between over- and under-approximation by pruning) *For each pruning $p(MM)$ and predicate φ:*

$$p^U[\varphi(v_1, \ldots, v_n)] \Rightarrow \varphi(v_1, \ldots, v_n) \Rightarrow p^O[\varphi(v_1, \ldots, v_n)].$$

6.3.3 Incremental Model Generation by Iterative Solver Calls

By using metamodel pruning, we first aim to generate valid instance models for the pruned metamodel, which is a simplified problem for the underlying logic solver. Instance models of increasing size will be gradually generated by using valid models of the pruned metamodel as partial snapshots (i.e. initial seeds) for generating instances for a larger metamodel. Therefore, an incremental model generation task is also given with a target size s and a target metamodel MM, but with an additional partial snapshot M_P. M_P is a valid instance of pruned metamodel $p(MM)$. M_P has s_P number of objects ($s_P \leq s$).

From a logic perspective, the partial snapshot defines a partial interpretation of relations for model generation, which may simplify the task of the solver compared to using fully uninterpreted relations. In order to exploit this additional information, the relations in the logic problem are partitioned into two sets of interpreted and uninterpreted symbols. $O_P = \{o_1, \ldots, o_{s_P}\}$ are the objects in the partial snapshot. The extra objects to be generated in this step are denoted by $O_N = \{o_{s_P+1}, \ldots, o_s\}$. The relations are partitioned according to the following rules:

- **Classes**: Each class predicate $\mathsf{C}(\cdot)$ in MM is separated into two: a fully interpreted $\mathsf{C_O}(\cdot)$ predicate for the objects in the partial snapshot O_P, and an uninterpreted $\mathsf{C_N}(\cdot)$ for the newly generated objects O_N. Therefore an object o is instance of a class C in the generated model if $\mathsf{C_O}(o) \vee \mathsf{C_N}(o)$ is satisfied. If the class is not in the pruned metamodel ($C \notin \mathit{Cls}_{p(MM)}$) then $\mathsf{C_O}(o)$ is to be omitted, and if no new elements are created from a class then $\mathsf{C_N}(o)$ can be omitted.

- **References**: Each reference predicate $\mathsf{R}(\cdot, \cdot)$ is separated into four categories: a fully interpreted $\mathsf{R_{OO}}(\cdot, \cdot)$ between the objects of the partial snapshot (O_P), an uninterpreted $\mathsf{R_{NN}}(\cdot, \cdot)$ between the objects of the newly created objects (O_N), and two additional uninterpreted relations $\mathsf{R_{ON}}(\cdot, \cdot)$ and $\mathsf{R_{NO}}(\cdot, \cdot)$ connecting the elements of the partial snapshot with the newly created elements (relations over $O_O \times O_N$ and $O_N \times O_O$ respectively). Therefore a reference $R(o, t)$ exists in the generated model if $\mathsf{R_{OO}}(o, t) \vee \mathsf{R_{NN}}(o, t) \vee \mathsf{R_{NO}}(o, t) \vee \mathsf{R_{ON}}(o, t)$. If the relation is not in the pruned metamodel ($R \notin \mathit{Ref}_{p(MM)}$) then $\mathsf{R_{OO}}(o, t)$ can be omitted, and if no new elements are created from a class then $\mathsf{R_{NN}}(o, t)$, $\mathsf{R_{NO}}(o, t)$ and $\mathsf{R_{ON}}(o, t)$ can also be omitted.

- **Attributes**: Attribute predicates are separated into a fully interpreted $\mathsf{A_{OO}}(\cdot, \cdot)$ for the objects in the partial snapshots O_P, and an uninterpreted relation $\mathsf{A_{NO}}(\cdot, \cdot)$ for the newly created elements O_N. An object o has an attribute value v ($\mathsf{A}(o, v)$) if $\mathsf{A_{OO}}(o, v) \vee \mathsf{A_{NO}}(o, v)$. Attribute predicates are treated as reference predicates for omission.

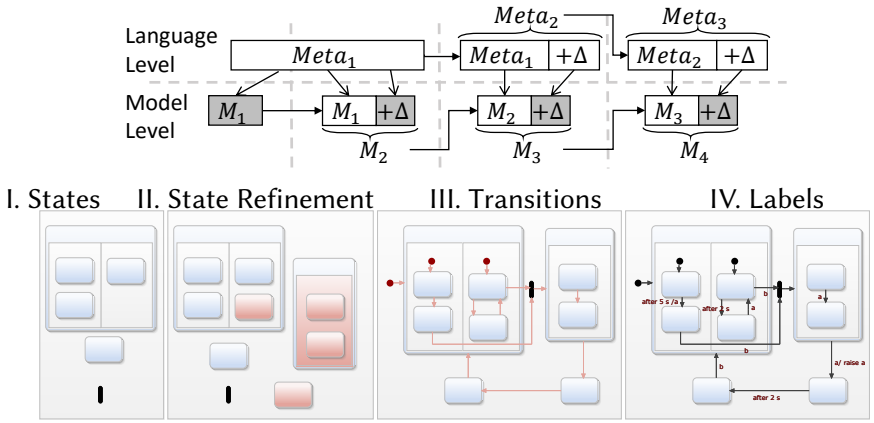


Figure 6.4: Model generation iterations

The level of incrementality is still unfortunately limited from an important aspect. The background solvers typically provide no direct control over the simultaneous creation of new elements, i.e. we cannot provide domain- specific hints to the solver when the creation of an object always depends on the creation or existence of another object. This can still cause issues when a multitude of WF constraints are defined.

> **Example 26.** In our running example, the instance models are generated in four steps, which is illustrated in Figure 6.4. First, initial seeds are generated for the state hierarchy (M_1 over MM_1), which are extended in the second step to model M_2 with the same metamodel elements. Then the metamodel is extended to MM_2, and the transitions and the initial states are added to model M_3. Finally, triggers, guards and actions can be added to the model to obtain M_4.

6.4 Measurements

In order to assess the effectiveness of incremental model generation using constraint approximation for synthesizing well-formed instance models for domain-specific languages, we conducted some initial experiments using the Alloy Analyzer as background solver. We were interested in the following questions:

- Is incremental model generation with metamodel pruning and constraint approximation effective in increasing the size of models, the success rate or decreasing the runtime of the solver?
- Is incremental model generation still effective if metamodel pruning or constraint approximation is excluded?

6.4.1 Configurations

We conducted measurements on two versions of the Yakindu statechart metamodel: Phase 1 and Phase 2 (see Figure 6.2). The pruned metamodel of Phase 1 (MM_1) contains 8 classes and 2 references, and no well- formedness constraints by default. The metamodel of Phase 2 (MM_2) contains 10 classes,

				MM1			MM2		
				#CLS:X	#REF:Y	#WF:Z	#CLS:X	#REF:Y	#WF:Z
				8	2	0 + 2	10	4	8
	Incre-mental	MM Pruning	Constraint Approx	Runtime (ms)	Model size (#)	Success rate (%)	Runtime (ms)	Model size (#)	Success rate (%)
Base	No	No	No	18349	60	100%	39040	12	0%
				Timeout	70	N/A	Timeout	16	N/A
W/o Prune	Yes	No	Yes	7327 + 11176	50+50	100%	Timeout	16	N/A
W/o Approx	Yes	Yes	No	12600+34804	50+50	100%	230 + 183465	20+30	0%
Full	Yes	Yes	Yes	7327 + 11176	50+50	100%	1644 + 44362	20+30	100%

Figure 6.5: Measurement results

4 references and 8 constraints (including the 5 WF constraints listed in the chapter and 3 more for restricting entry states).

- As a **base** configuration, the Alloy Analyzer is executed separately for the two problems with 1 minute timeout. We record two cases: the largest model derived and a slightly larger model size where timeout was observed.
- Next, we run the solver incrementally with an initial model of size N and an increment of size K denoted as $N+K$ in Figure 6.5 **without constraint approximation** but with metamodel pruning. Moreover, instance models derived for Phase 1 are used as partial snapshots for Phase 2.
- Then we run the solver incrementally with constraint approximation but **without metamodel pruning**. For that purpose, the constraint set for Phase 1 contains two approximated constraints: (1) Each region has a state where the entry state will point, and (2) There are orthogonal states in the model. Again, instance models derived for Phase 1 are used as partial snapshots for Phase 2, but the full metamodel is considered in Phase 2.
- Finally we configure the solver for **full** incrementally with constraint approximation and metamodel pruning by reusing instances of Phase 1 as partial snapshots in Phase 2.

6.4.2 Measurement setup

Each model generation task was executed on the DSL presented in this chapter 5 times using the Alloy Analyzer (with SAT4j- solver), then the median of the execution times was calculated. The measures are executed with one minute timeout on an average personal computer[1]. We measure the *runtime* of model generation, the *model size* denoting the maximal number of elements the derived model may contain, and the *success rate* denoting the percentage of cases when a well-formed model was derived, which satisfy all WF constraints within the given search scope.

6.4.3 Measurement results

Results of our measurements are summarized in Figure 6.5. We summarize our observations below.

- **Base:** For MM_1, Alloy was able to generate models with up to 60 objects. As there are no constraints at this level, many synchronizations are created (about half of the objects were synchronization and with only 5-10 states). Over 60 objects, the runtime grows rapidly as the SAT

[1]CPU: Intel Core-i5-m310M, MEM: 16GB but the back-end solver can use 4GB only, OS: Windows 10 Pro, Reasoner: Alloy Analyzer 4.2 with sat4j

solver runs out of the maximal 4 GB memory. For MM_2, Alloy was unable to create any models that satisfies all of the constraints as the search scope turned out to be too small to create valid models with synchronizations.

- **W/o approx** Alloy was able to generate models with 100 elements in two steps where each iterative step had comparable runtime. However, since no constraints are considered for MM_1, Alloyed failed to extend partial snapshots of MM_1 to well-formed models for MM_2 (success rate: 0%, although for this specific case, we executed over 100 runs of the solver due to the unexpectedly low success rate). Furthermore, we had to reduce the scope of search to 20 and 30 new elements with types taken from $MM_2 \setminus MM_1$ due to timeouts.
- **W/o prune** When metamodel pruning was excluded but approximated constraints were included for MM_1, model generation succeeded for 100 elements, but extending them to models of MM_2 failed (as in this case, new elements could take any elements from MM_2)
- **Full** With incremental model generation by combining metamodel pruning and constraint approximation, we were able to generate well-formed models for both MM_1 and MM_2, which was the only successful case for the latter.

6.4.4 Analysis of results

While we used a reasonably sized statechart metamodel extracted from a real modeling tool (including everything to model state machines, but excluding imports and namespacing), we avoid drawing generic conclusions for the exact scalability of our results. Instead, we summarize some negative results which are hardly specific to the chosen example:

- Mapping a model generation problem to Alloy and running the Alloy Analyzer in itself will likely fail to derive useful results for practical metamodels, especially, in the presence of complex well-formedness constraints. Our observation is that many objects need to be created at the same time in consistent way, which cannot be efficiently handled by the underlying solver (either the scope is too small or out-of-memory). Altogether, the Alloy Analyzer was more effective in finding consistent model instances than in proving that a problem is inconsistent, thus there are no solutions.
- An incremental approach with metamodel pruning but without constraint approximation will increase the overall size of the derived models, but the false positive rate would quickly increase.
- An incremental approach without metamodel pruning but with constraint approximation will likely have the same pitfalls as the original Alloy case: either the scope of search will become insufficient, or we run out of memory.
- Combining incremental model generation with metamodel pruning and constraint approximation is promising as a concept as it significantly improved wrt. the baseline case. But the underlying solver was still not sufficiently powerful to guarantee scalability with the Alloy back-end solver for complex industrial cases.

6.5 Related work

We compared our solution with existing model generation techniques with respect to the characteristics of *inputs* and *output results* in Table 6.1. As for *inputs*, the model generation can be (1) initiated

		Logic Solvers	Uncertain Models	Rule-Based Generators	Iterative Solver Call
Inputs	Partial Snapshot	+	++	-	+
	Effective Metamodel	-	-	+	+
	Local Constraints	+	-	+	+
	Global Constraints	+	-	-	+
Outputs	Metamodel-compliant	+	+	+	+
	Well-formed	+	-	-	+
	Diverse	-	-	+	?
	Scalable	-	-	++	+/-
	Decidability	-	+	+	- (graceful degradation)

Table 6.1: Comparison of related approaches

from a *partial snapshot*, (2) focused on an *effective metamodel*. Additionally, an approach may support (3) *local* and (4) *global constraints* well-formedness constraints: a local constraint accesses only the attributes and the outgoing references of an object, while a global constraint specifies a complex structural pattern. Local constraints are frequently attached to objects (e.g. in UML class diagrams), while global constraints are widely used in domain-specific modeling languages. As *outputs*, the generated models may (i) be metamodel-compliant (ii) satisfy all *well-formedness* constraints of the language. When generated models are intended to be used as test cases, some approaches may guarantee a certain level of coverage or (iii) *diversity*. We consider a technique (iv) *scalable* if there is no hard limit on the model size (as demonstrated in the respective papers). Finally, a model generation approach may be (v) *decidable* which always terminates with a result. Our comparison excludes approaches like which do not guarantee metamodel- compliance of generated instance models.

Logic Solver Approaches. Several approaches map a model generation problem (captured by a metamodel, partial snapshots, and a set of WF constraints) into a logic problem, which are solved by underlying SAT/SMT-solvers. Complete frameworks with standalone specification languages include Formula [JLB11] (which uses Z3 SMT- solver [MB08]), Alloy [Jac02] (which relies on SAT solvers like Sat4j[LBP10]) and Clafer [Bak+16] (using backend reasoners like Alloy).

There are several approaches aiming to validate standardized engineering models enriched with OCL constraints [GBR05] by relying upon different back-end logic-based approaches such as constraint logic programming [CCR07; CCR08; BC12], SAT-based model finders (like Alloy) [SAB09; Ana+10; Büt+12; KHG11; Soe+10], first-order logic [BKS02], constructive query containment [Que+12], higher-order logic [BW07; GRR09], or rewriting logics [CE08].

Partial snapshots and WF constraints can be uniformly represented as constraints [J2], but metamodel pruning is not typical. Growing models are supported in [JS07] for a limited set of constraints. Scalability of all these approaches are limited to small models / counter-examples. Furthermore, these approaches are either a priori bounded (where the search space needs to be restricted explicitly) or they have decidability issues.

The main difference of our current approach is its *iterative derivation of models* and the *approximative handling of metamodels and constraints*. However, our approach is independent from the actual mapping of constraints to logic formulae, thus it could potentially be integrated with most of the above techniques.

Uncertain Models. Partial models are also similarity to uncertain models, which offer a rich specification language [FSC12a; SC15] amenable to analysis. Uncertain models provide a more expressive

language compared to partial snapshots but without handling additional WF constraints. Such models document semantic variation points generically by annotations on a regular instance model, which are gradually resolved during the generation of concrete models. An uncertain model is more complex (or informative) than a concrete one, thus an a priori upper bound exists for the derivation, which is not an assumption in our case.

Potential concrete models compliant with an uncertain model can synthesized by the Alloy Analyzer [SFC12], or refined by graph transformation rules [Sal+15]. Each concrete model is derived in a single step, thus their approach is not iterative like ours. Scalability analysis is omitted from the respective papers, but refinement of uncertain models is always decidable.

Rule-based Instance Generators. A different class of model generators relies on rule-based synthesis driven by randomized, statistical or metamodel coverage information for testing purposes [Bro+06; FSB04a]. Some approaches support the calculation of effective metamodels [Sen+09], but partial snapshots are excluded from input specifications. Moreover, WF constraints are restricted to local constraints evaluated on individual objects while global constraints of a DSL are not supported. On the positive side, these approaches guarantee the diversity of models and scale well in practice.

Iterative approaches. An iterative approach is proposed *specifically for allocation problems* in [KJS11] based on Formula. Models are generated in two steps to increase diversity of results. First, non-isomorphic submodels are created only from an effective metamodel fragment. Diversity between submodels is achieved by a problem-specific symmetry-breaking predicate [Cra+96] which ensures that no isomorphic model is generated twice. In the second step the algorithm completes the different submodels according to the full model, but constraints are only checked at the very final stage. This is a key difference in our approach where an approximation of constraints is checked at each step, which reduces the number of inconsistent intermediate models. An iterative, counter-example guided synthesis is proposed for higher-order logic formulae in [Mil+15], but the size of derived models is fixed.

6.6 Conclusion

In the chapter, I proposed an incremental model generation approach which (1) iteratively calls black-box logic solvers to guarantee well-formedness by (2) feeding instance models obtained in a previous step as partial snapshots (compulsory model fragments) to a subsequent phase to limit the number of new elements, and using (3) various approximations of metamodels and constraints. Our experiments show that significantly larger model instances can be generated with the same solvers using such an incremental approach especially in the presence of complex well-formedness constraints. However, part of our experimental results are negative in the sense that the proposed iterative approach is still not scalable to derive large model instances of complex industrial languages with due to restrictions of the underlying Alloy Analyzer and the SAT solver libraries.

Chapter 7

Diverse Graph Model Generation With Logic Solvers

7.1 Introduction

The design of complex DSLs tools is a challenging. As the complexity of DSL tools increases, special attention is needed to validate the modeling tools themselves (e.g. for tool qualification purposes) to ensure that WF constraints and the preconditions of model transformation and code generation functionality [Bau+06][J2][C10] are correctly implemented in the tool. There are many approaches aiming to address the testing of DSL tools (or transformations) [BA05; Ara+15; Val+12] which necessitate *the automated synthesis of graph models* to serve as test inputs. Many best practices of testing (such as equivalence partitioning [Rei97], mutation testing [JH11]) recommends the synthesis of *diverse* graph models where any pairs of models are structurally different from each other to achieve high coverage or a diverse solution space.

While software diversity is widely studied [Bau+14], existing diversity metrics for graph models are much less elaborate [B14]. Model comparison techniques [Dif] frequently rely upon the existence of node identifiers, which can easily lead to many isomorphic models. Moreover, checking graph isomorphism is computationally very costly. Therefore practical solutions tend to use approximate techniques to achieve certain diversity by random sampling [JSS13], incremental generation [KJS11][C10], or using symmetry breaking predicates [TJ07]. Unlike equivalence partitions which capture diversity of inputs in a customizable way for testing traditional software, a similar diversity concept is still missing for graph models.

In this chapter, I propose *diversity metrics* to characterize a single model and a set of models. For that purpose, we innovatively reuse neighborhood graph shapes [RD06], which provide a fine-grained typing for each object based on the structure (e.g. incoming and outgoing edges) of its neighborhood. Moreover, we propose an *iterative model generation technique* to automatically synthesize a diverse set of models for a DSL where each model is taken from a different equivalence class wrt. graph shapes as an equivalence relation.

The chapter evaluates our diversity metrics and model generator in the context of mutation-based testing [MBT06] of WF constraints in an industrial DSL tool. We evaluate and compare the *mutation score* and *our diversity metrics* of test suites obtained by (1) an Alloy based model generator (using symmetry breaking predicates to ensure diversity), (2) an iterative graph solver based generator using neighborhood shapes, and (3) from real models created by humans. Our finding is that a diverse set of models derived along different neighborhood shapes has better mutation score. Furthermore, based

on a test suite with 4850 models, we found that high correlation between mutation score and our diversity metrics, which indicates that our metrics may be good predictors in practice for testing.

Up to our best knowledge, our method is one of the first studies on (software) model diversity. From a testing perspective, our diversity metrics provide a stronger characterization (with respect to granularity) of a test suite of models than traditional metamodel coverage which is used in many research papers. Furthermore, model generators using neighborhood graph shapes (that keep models only if they are surely non-isomorphic) provide increased diversity compared to symmetry breaking predicates (which exclude models if they are surely isomorphic).

After illustrating the technical challenge in the context of an industrial DSL tool we provide an overview of key underlying modeling techniques in Section 7.2. A conceptual overview of identifying missing or extra constraints is given in Section 7.3. An experimental evaluation of our approach is provided in Section 7.4, while related work is assessed in Section 7.5 before concluding in Section 7.6. This chapter is based on conference paper [C11].

7.2 Preliminaries

Core modeling concepts and testing challenges of DSL tools will be illustrated in the context of Yakindu Statecharts [Yak]

> **Example 27.** A simplified metamodel for Yakindu state machines is illustrated in Figure 7.1. A Statechart consists of Regions, which in turn contain states (called Vertices) and Transitions. In this chapter, we introduced more Vertices compared to the metamodel used in Chapter 3: an abstract state Vertex is further refined into RegularStates (like State or FinalState) and PseudoStates (like Entry, Exit or Choice). We excluded Synchronization scenarios.

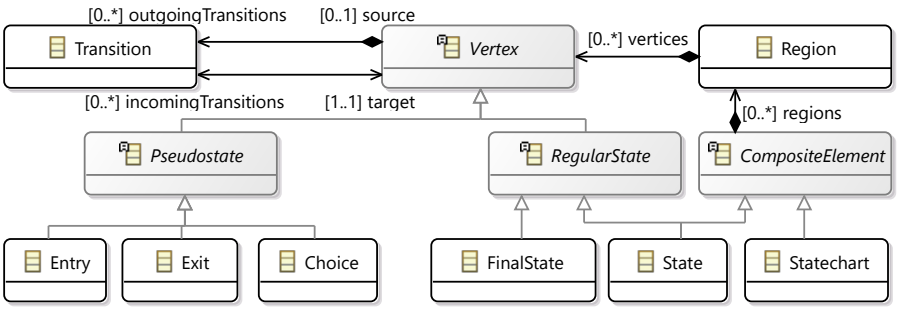


Figure 7.1: Metamodel extract from Yakindu state machines

As previously discussed in Chapter 3, an ordinary *instance model* can be represented as a logic structure $M = \langle O_M, \mathcal{I}_M \rangle$ where O_M is the finite set of objects (the size of the model is $|M| = |O_M|$), and $\mathcal{I}_M$ provides interpretation for all predicate symbols in Σ as follows:

- the interpretation of a unary predicate symbol C_i is defined in accordance with the types of the EMF model: $\mathcal{I}_M(C_i) : O_M \to \{1, 0\}$ An object $o \in O_M$ is an instance of a class C_i in a model M if $\mathcal{I}_M(C_i)(o) = 1$.

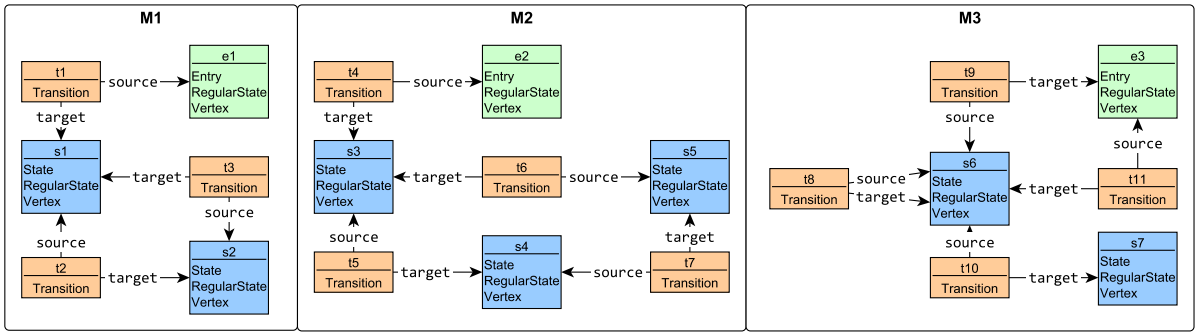


Figure 7.2: Example instance models (as directed graphs)

- the interpretation of a binary predicate symbol R_j is defined in accordance with the links in the EMF model: $\mathcal{I}_M(\mathsf{R}_j) : O_M \times O_M \to \{1, 0\}$. There is a reference R_j between $o_1, o_2 \in O_M$ in model M if $\mathcal{I}_M(\mathsf{R}_j)(o_1, o_2) = 1$.

Example 28. Figure 7.2 shows graph representations of three (partial) instance models. For the sake of clarity, Regions and inverse relations incomingTransitions and outgoingTransitions are excluded from the diagram. In M_1 there are two States ($s1$ and $s2$), which are connected to a loop via Transitions $t2$ and $t3$. The initial state is marked by a Transition $t1$ from an entry $e1$ to state $s1$. M_2 describes a similar statechart with three states in loop ($s3$, $s4$ and $s5$ connected via $t5$, $t6$ and $t7$). Finally, in M_3 there are two main differences: there is an incoming Transition $t11$ to an Entry state ($e3$), and there is a State $s7$ that does not have outgoing transition. While all these $M1$ and $M2$ are non-isomorphic, later we illustrate why they are not diverse.

In many industrial modeling tools, WF constraints are captured either by OCL constraints [Ocl] or graph patterns (GP) [Ujh+15]. Here we use a tool-independent logic predicate representation of those constraints as introduced in Section 3.2.

7.2.1 Motivation: testing of DSL tools

A code generator would normally assume that the input models are well-formed, i.e. all WF constraints are validated prior to calling the code generator. However, there is no guarantee that the WF constraints actually checked by the DSL tool are exactly the same as the ones required by the code generator. For instance, if the validation forgets to check a subclause of a WF constraint, then runtime errors may occur during code generation. Moreover, the precondition of the transformation rule may also contain errors. For that purpose, it is important that WF constraints and model transformations of DSL tools can be systematically tested. Alternatively, model validation can be interpreted as a special case of model transformation, where precondition of the transformation rules are fault patterns, and the actions place error markers on the model [Ujh+15].

A popular approach for testing DSL tools is mutation testing [MBT06; SBM09] which aims to reveal missing or extra predicates by (1) deriving a set of mutants (e.g. WF constraints in our case) by applying a set of mutation operators. Then (2) the test suite is executed for both the original and the mutant programs, and (3) their output are compared. (4) A mutant is killed by a test if different output is produced for the two cases (i.e. different match set). (5) The mutation score of a test suite is

calculated as the ratio of mutants killed by some tests wrt. the total number of mutants. A test suite with better mutation score is preferred [JH11].

7.2.2 Fault model and detection

As a fault model, we consider omission faults in WF constraints of DSL tools where some subconstraints are not actually checked. In our fault model, a WF constraint is given in a conjunctive normal form $\varphi_e = \varphi_1 \wedge \cdots \wedge \varphi_k$, all unbound variables are quantified existentially ($\exists$), and may refer to other predicates specified in the same form. Note that this format is equivalent to first order logic, and does not reduce the range of supported graph predicates. We assume that in a faulty predicate (a mutant) the developer may forget to check one of the predicates φ_i (Constraint Omission, CO), i.e.

$$\varphi_e{:=}[\varphi_1 \wedge \ldots \wedge \varphi_i \wedge \ldots \wedge \varphi_k] \text{ is rewritten to}$$

$$\varphi_f{:=}[\varphi_1 \wedge \cdots \wedge \varphi_{i-1} \wedge \varphi_{i+1} \wedge \cdots \wedge \varphi_k],$$

or may forget a negation (Negation Omission), i.e.

$$\varphi_e{:=}[\varphi_1 \wedge \ldots \wedge (\neg\varphi_i) \wedge \ldots \wedge \varphi_k] \text{ is rewritten to}$$

$$\varphi_f{:=}[\varphi_1 \wedge \ldots \wedge \varphi_i \wedge \ldots \wedge \varphi_k].$$

Given an instance model M, we assume that both $[\![\varphi_e]\!]^M$ and the faulty $[\![\varphi_f]\!]^M$ can be evaluated separately by the DSL tool. Now a test model M detects a fault if there is a variable binding Z, where the two evaluations differ, i.e. $[\![\varphi_e]\!]^M Z \neq [\![\varphi_f]\!]^M Z$.

Example 29. Two WF constraints checked by the Yakindu environment can be captured by graph predicates as follows:

$$\varphi : \mathit{incomingToEntry}(E){:=}\exists T : \mathsf{Entry}(E) \wedge \mathsf{target}(T, E)$$

$$\phi : \mathit{noOutgoingFromEntry}(E){:=}\mathsf{Entry}(E) \wedge \neg(\exists T : \mathsf{source}(T, E))$$

According to our fault model, we can derive two mutants for *incomingToEntry* as predicates

$$\varphi_{f_1}{:=}\mathsf{Entry}(E) \text{ and } \varphi_{f_2}{:=}\exists t : \mathsf{target}(T, E).$$

Constraints φ and ϕ are satisfied in model M_1 and M_2 as the corresponding graph predicates have no matches, thus $[\![\varphi]\!]^{M_1} Z = 0$ and $[\![\phi]\!]^{M_1} Z = 0$. As a test model, both M_1 and M_2 is able to detect the same omission fault both for φ_{f_1} as $[\![\varphi_{f_1}]\!]^{M_1} = 1$ (with $E \mapsto e1$ and $E \mapsto e2$) and similarly φ_{f_2} (with $s1$ and $s3$). However, M_3 is unable to kill mutant φ_{f_1} as (φ had a match $E \mapsto e3$ which remains in φ_{f_1}), but able to detect others.

7.3 Model diversity metrics for testing DSL tools

As a general best practice in testing, a good test suite should be diverse, but the interpretation of diversity may differ. For example, equivalence partitioning [Rei97] partitions the input space of a program into equivalence classes based on observable output, and then select the different test cases

of a test suite from different execution classes to achieve a diverse test suite. However, while software diversity has been studied extensively [Bau+14], model diversity is much less covered.

In existing approaches [CCR07; CCR08; Val+12; Sch+13; Bro+06; BA05] for testing DSL and transformation tools, a test suite should provide full *metamodel coverage* [WKC06], and it should also guarantee that any pairs of models in the test suite are non-isomorphic [JSS13; TJ07]. In [B14], the diversity of a model M_i is defined as the number of (direct) types used from its MM, i.e. M_i is more diverse than M_j if more types of MM are used in M_i than in M_j. Furthermore, a model generator Gen deriving a set of models $\{M_i\}$ is diverse if there is a designated distance between each pairs of models M_i and M_j: $dist(M_i, M_j) > D$, but no concrete distance function is proposed.

Below, we propose diversity metrics for a single model, for pairs of models and for a set of models based on neighborhood shapes [RD06], a formal concept known from the state space exploration of graph transformation systems [Ren06]. Our diversity metrics generalize both metamodel coverage and (graph) isomorphism tests, which are derived as two extremes of the proposed metric, and thus it defines a finer grained equivalence partitioning technique for graph models.

7.3.1 Neighborhood shapes of graphs

A neighborhood Nbh_i describes the local properties of an object in a graph model for a range of size $i \in \mathbb{N}$ [RD06]. The neighbourhood of an object o describes all unary (class) and binary (reference) relations of the objects within the given range. Informally, neighbourhoods can be interpreted as richer types, where the original classes are split into multiple subclasses based on the difference in the incoming and outgoing references.

Definition 27 (Neighborhood descriptors) *Neighborhood descriptors are defined recursively over a signature $\langle \Sigma, a \rangle$:*

- *For range $i = 0$, Nbh_0 is a subset of class symbols:*

$$Nbh_0 \subseteq 2^{\{\mathrm{C}_1,\ldots,\mathrm{C}_n\}}$$

- *A neighbor Ref_i for $i > 0$ is defined by a reference symbol and a neighborhood:*

$$Ref_i \subseteq \{\mathrm{R}_1, \ldots, \mathrm{R}_m\} \times Nbh_{i-1}.$$

- *For a range $i > 0$ neighborhood Nbh_i is defined by a previous neighborhood and two sets of neighbor descriptors (for incoming and outgoing references separately):*

$$Nbh_i \subseteq Nbh_{i-1} \times 2^{Ref_i} \times 2^{Ref_i}.$$

Then, a shaping function calculates the shape of a given model.

Definition 28 (Shaping) *Shaping function $nbh_i : O_M \to Nbh_i$ maps each object in a model M to a neighborhood with range i. If $i = 0$, then:*

$$nbh_0(o) := \{\mathrm{C} \mid [\![\mathrm{C}(v)]\!]^M_{v \mapsto o} = 1\}$$

otherwise, if $i > 0$ then:

$$nbh_i(o) := \langle nbh_{i-1}(o), in, out \rangle, \text{ where:}$$

$$in = \{\langle \mathsf{R}, n\rangle | \exists o' \in O_M : [\![\mathsf{R}(v', v)]\!]^M_{v\mapsto o, v'\mapsto o'} \wedge n = nbh_{i-1}(o')\}$$

$$out = \{\langle \mathsf{R}, n\rangle | \exists o' \in O_M : [\![\mathsf{R}(v, v')]\!]^M_{v\mapsto o, v'\mapsto o'} \wedge n = nbh_{i-1}(o')\}$$

A shape of a model M for range i (denoted as $S_i(M)$) is a set of neighborhood descriptors of the model:

$$S_i(M) := \{x | \exists o \in O_M : nbh_i(o) = x\}.$$

A shape can be interpreted and illustrated as a as a type graph: after calculating the neighborhood for each object, each neighborhood is represented as a node in the graph shape. Moreover, if there exist at least one link between objects in two different neighborhoods, the corresponding nodes in the shape will be connected by an edge. We will use the size of a shape $|S_i(M)|$ which is the number of shapes used in M.

Example 30. We illustrate the concept of graph shapes for model M_1. For range 0, objects are mapped to class names as neighborhood descriptors:

$nbh_0(e) = \{\mathsf{Entry}, \mathsf{PseudoState}, \mathsf{Vertex}\}$

- $nbh_0(t1) = nbh_0(t2) = nbh_0(t3) = \{\mathsf{Transition}\}$
- $nbh_0(s1) = nbh_0(s2) = \{\mathsf{State}, \mathsf{RegularState}, \mathsf{Vertex}\}$

For range 1, objects with different incoming or outgoing types are further split, e.g. the neighborhood of $t1$ is different from that of $t2$ and $t3$ as it is connected to an Entry along a source reference, while the source of $t2$ and $t3$ are States.

$nbh_1(t1) = \langle\{\mathsf{Transition}\}, \emptyset, \{\langle\mathsf{source}, \{\mathsf{Entry}, \mathsf{PseudoState}, \mathsf{Vertex}\}\rangle,$
$\langle\mathsf{target}, \{\mathsf{State}, \mathsf{RegularState}, \mathsf{Vertex}\}\rangle$

- $nbh_1(t2) = nbh_1(t3) = \langle\{\mathsf{Transition}\}, \emptyset, \{\langle\mathsf{source}, \{\mathsf{State}, \mathsf{RegularState}, \mathsf{Vertex}\}\rangle,$
  $\langle\mathsf{target}, \{\mathsf{State}, \mathsf{RegularState}, \mathsf{Vertex}\}\rangle$

For range 2, each object of M_1 would be mapped to a unique element. In Figure 7.3, the neighborhood shapes of models M_1, M_2, and M_3 for range 1, are represented in a visual notation adapted from [RD06; RSW04] (without additional annotations e.g. multiplicities or predicates used for verification purposes). The trace of the concrete graph nodes to neighbourhood is illustrated on the right. For instance, $e1$ and $e2$ in $M1$ and M_2 Entries are both mapped to the same neighbourhood $n1$, while $e3$ can be distinguished from them as it has incoming reference from a transition, thus creating a different neighbourhood $n5$.

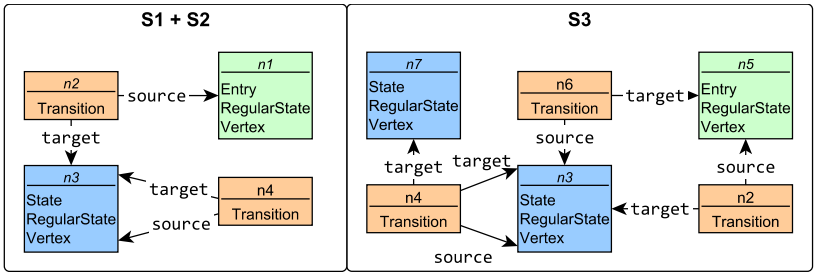
The theoretical foundations of graph shapes [RD06; RSW04] prove several key semantic properties which are exploited:

P1 There are only a *finite number of graph shapes in a certain range*, and a smaller range reduces the number of graph shapes, i.e. $|S_i(M)| \leq |S_{i+1}(M)|$.

P2 $|S_i(M_j)| + |S_i(M_k)| \geq |S_i(M_j \cup M_k)| \geq |S_i(M_j)|$ and $|S_i(M_k)|$.

7.3.2 Metrics for model diversity

We define two metrics for model diversity based upon neighborhood shapes. *Internal diversity* captures the diversity of a single model, i.e. it can be evaluated individually for each and every generated model. As neighborhood shapes introduce extra subtypes for objects, this model diversity metric



o	$nbh_1(o)$
$e1-2$	$n1$
$e3$	$n5$
$s1-6$	$n3$
$s7$	$n7$
$t1, 4, 11$	$n2$
$t2, 3, 5-8, 10$	$n4$
$t9$	$n6$

Figure 7.3: Sample neighborhood shapes of M_1, M_2 and M_3

measures the number of neighborhood types used in the model with respect to the size of the model. *External diversity* captures the distance between pairs of models. Informally, this diversity distance between two models will be proportional to the number of different neighborhoods covered in one model but not the other.

Definition 29 (Internal model diversity) *For a range i of neighborhood shapes for model M, the internal diversity of M is the number of shapes wrt. the size of the model: $d_i^{int}(M) = |S_i(M)|/|M|$.*

The range of this internal diversity metric $d_i^{int}(M)$ is $[0..1]$, and a model M with $d_1^{int}(M) = 1$ (and $|M| \geq |MM|$) *guarantees full metamodel coverage* [WKC06], i.e. it surely contains all elements from a metamodel as types. As such, it is an appropriate diversity metric for a model in the sense of [B14]. Furthermore, given a specific range i, the number of potential neighborhood shapes within that range is finite, but it grows superexponentially. Therefore, for a small range i, one can derive a model M_j with $d_i^{int}(M_j) = 1$, but for larger models M_k (with $|M_k| > |M_j|$) we will likely have $d_i^{int}(M_j) \geq d_i^{int}(M_k)$. However, due to the rapid growth of the number of shapes for increasing range i, for most practical cases, $d_i^{int}(M_j)$ will converge to 1 if M_j is sufficiently diverse.

Definition 30 (External model diversity) *Given a range i of neighborhood shapes, the external diversity of models M_j and M_k is the number of shapes contained exclusively in M_j or M_k but not in the other, formally, $d_i^{ext}(M_j, M_k) = |S_i(M_j) \oplus S_i(M_k)|$ where $\oplus$ denotes the symmetric difference of two sets.*

External model diversity allows to compare two models. One can show that this metric is a (pseudo)-distance in the mathematical sense [AF12], and thus, it can serve as a diversity metric for a model generator in accordance with [B14].

Definition 31 (Pseudo-distance) *A function $d : \mathcal{M} \times \mathcal{M} \to \mathbb{R}$ is called a pseudo-distance, if it satisfies the following properties:*

- *d is non-negative: $d(M_j, M_k) \geq 0$*
- *d is symmetric $d(M_j, M_k) = d(M_k, M_j)$*

- *if M_j and M_k are isomorphic, then $d(M_j, M_k) = 0$*
- *triangle inequality: $d(M_j, M_l) \leq d(M_k, M_j) + d(M_j, M_l)$*

Theorem 12 (Exterrnal model diversity) *External model diversity $d_i^{ext}(M_j, M_k)$ is a pseudo-distance between models M_j and M_k for any i.*

During model generation, we will exclude a model M_k if $d_i^{ext}(M_j, M_k) = 0$ for a previously defined model M_j, but *it does not imply that they are isomorphic*. Thus our definition allows to avoid graph isomorphism checks between M_j and M_k which have high computation complexity. Note that external diversity is a dual of symmetry breaking predicates [TJ07] used in the Alloy Analyzer where $d(M_j, M_k) = 0$ implies that M_j and M_k are isomorphic (and not vice versa).

Definition 32 (Coverage of model set) *Given a range i of neighborhood shapes and a set of models $MS = \{M_1, \ldots, M_k\}$, the coverage of this model set is defined as $cov_i\langle MS\rangle = |S_i(M_1) \cup \cdots \cup S_i(M_k)|$.*

The coverage of a model set is not normalised, but its value monotonously grows for any range i by adding new models. Thus it corresponds to our expectation that adding a new test case to a test suite should increase its coverage.

Example 31. Let us calculate the different diversity metrics for M_1, M_2 and M_3 of Figure 7.2. For range 1, they have the shapes illustrated in Figure 7.3. The internal diversity of those models are $d_1^{int}(M_1) = 4/6$, $d_1^{int}(M_2) = 4/8$ and $d_1^{int}(M_3) = 6/7$, thus M_3 is the most diverse model among them. As M_1 and M_2 has the same shape, the distance between them is $d_1^{ext}(M_1, M2) = 0$. The distance between M_1 and M_3 is $d_1^{ext}(M_1, M3) = 4$ as M_1 has 1 different neighbourhoods ($n1$), and M_3 has 3 ($n5$, $n6$ and $n7$). The set coverage of M_1, M_2 and M_3 is 7 altogether, as they have 7 different neighbourhoods ($n1$ to $n7$).

Now we aim at generating a diverse sequence of models $MS = \{M_1, M_2, \ldots, M_k\}$ for a given metamodel MM (and potentially, a set of constraints WF). First, model generation is an iterative process where previous solutions serve as further constraints [C10]. Second, it repeatedly calls a back-end graph solver [Vs][C6] to automatically derive consistent instance models which satisfy WF. As a key conceptual novelty, we enforce the structural diversity of models during the generation process using neighborhood shapes at different stages. Most importantly, if the shape $S_i(M_n)$ of a new instance model M_n obtained as a candidate solution is identical to the shape $S_i(M_j)$ for a previously derived model M_j for a predefined (input) neighborhood range i, the solution candidate is discarded, and iterative generation continues towards a new candidate.

7.4 Evaluation

In this section, we provide an empirical evaluation of our diversity metrics and model generation technique to address the following research questions:

RQ1: How effective is our technique in creating diverse models for testing?

RQ2: How effective is our technique in creating diverse test suites?

RQ3: Is there correlation between diversity metrics and mutation score?

Target Domain. In order to answer those questions, we executed model generation campaigns on a DSL extracted from Yakindu Statecharts (as proposed in [C10]). We used the partial metamodel describing the state hierarchy and transitions of statecharts (illustrated in Figure 6.2, containing 12 classes and 6 references). Additionally, we formalized 10 WF constraints regulating the transitions as graph predicates, based on the built-in validation of Yakindu.

For mutation testing, we used a constraint or negation omission operator (CO and NO) to inject an error to the original WF constraint in every possible way, which yielded 51 mutants from the original 10 constraints (but some mutants may never have matches). We checked both the original and mutated versions of the constraints for each instance model, and a model kills a mutant if there is a difference in the match set of the two constraints. The mutation score for a test suite (i.e. a set of models) is the total number of mutants killed that way.

Compared approaches. Our test input models were taken from three different sources. First, we generated models with our iterative approach using a graph solver (**GS**) with different neighborhoods for ranges **r=1** to **r=3**.

Next, we generated models for the same DSL using **Alloy**[TJ07], a well-known SAT-based relational model finder. For representing EMF metamodels we used traditional encoding techniques [Büt+12][J2]. To enforce model diversity, Alloy was configured with three different setups for symmetry breaking predicates: **s=0**, **s=10** and **s=20** (default value). For greater values the tool produced the same set of models. We used the latest 4.2 build for Alloy with the default Sat4j [LBP10] as back-end solver. All other configuration options were set to default.

Finally, we included 1250 manually created statechart models in our analysis (marked by **Human**). The models were created by students as solutions for similar (but not identical) statechart modeling homework assignments [B14] representing real models which were *not* prepared for testing purposes.

Measurement setup. To address **RQ1**-**RQ3**, we created a two-step measurement setup. In **Step I.** a set of instance models is generated with all **GS** and **Alloy** configurations. Each tool in each configuration generated a sequence of 30 instance models produced by subsequent solver calls, and each sequence is repeated 20 times (so 1800 models are generated for both **GS** and **Alloy**). In case of **Alloy**, we prevented the deterministic run of the solver to enable statistical analysis. The model generators was to create metamodel-compliant instances compliant with the structural constraints of Section 3.2 but ignoring the WF constraints. The target model size is set to 30 objects as Alloy did not scale with increasing size (the scalability and the details of the back-end solver is reported in [C6]). The size of **Human** models ranges from 50 to 200 objects.

In **Step II.**, we evaluate and the mutation score for all the models (and for the entire sequence) by comparing results for the mutant and original predicates and record which mutant was killed by a model. We also calculate our diversity metrics for a neighborhood range where no more equivalence classes are produced by shapes (which turned out to be $r = 7$ in our case study). We calculated the internal diversity of each model, the external diversity (distance) between pairs of models in each model sequence, and the coverage of each model sequence.

RQ1: Measurement Results and Analysis. Figure 7.4 shows the distribution of the number of mutants killed by at least one model from a model sequence (left box plot), and the distribution of internal diversity (right box plot). For killing mutants, **GS** was the best performer (regardless of the **r** range): most models found 36-41 mutants out of 51. On the other hand, **Alloy** performance varied based on the value of symmetry: for **s=0**, most models found 9-15 mutants (with a large number of positive outliers that found several errors). For **s=10**, the average is increased over 20, but the number of positive outliers simultaneously dropped. Finally, in default settings (**s=20**) **Alloy** generated similar

models, and found only a low number of mutants. We also measured the efficiency of killing mutants by **Human**, which was between **GS** and **Alloy**. None of the instance models could find more than 41 mutants, which suggests that those mutants cannot be detected at all by metamodel-compliant instances.

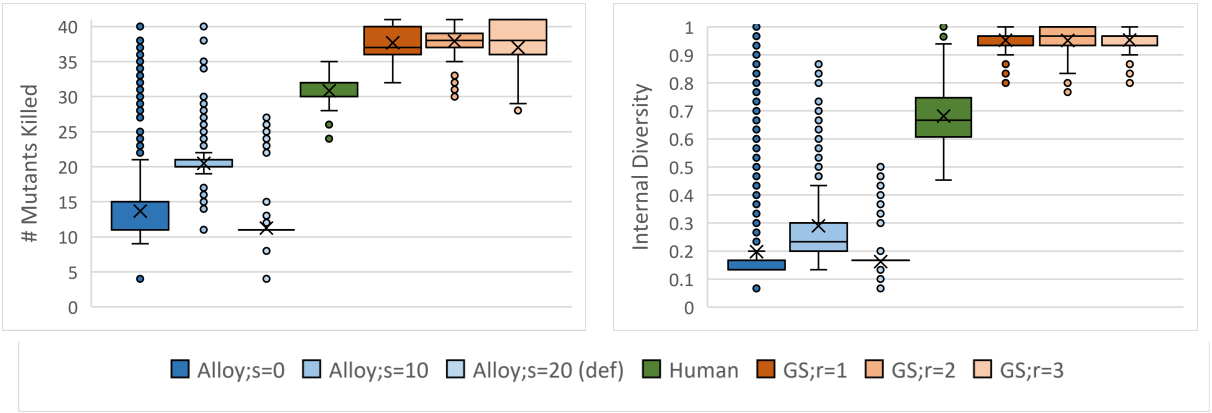
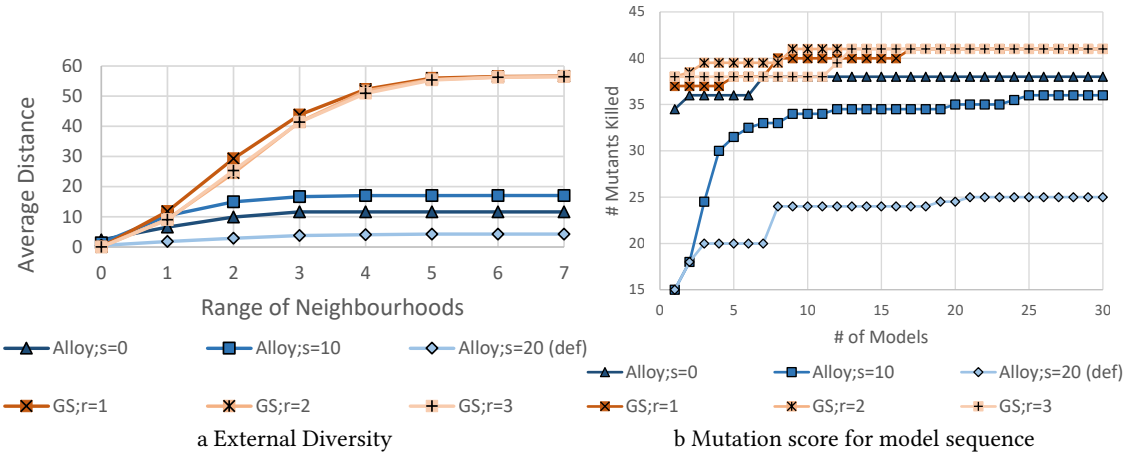


Figure 7.4: Mutation Score and Internal Diversity

The right side of Figure 7.4 presents the internal diversity of models measured as shape nodes/graph nodes (for fixpoint range 7). The result are similar: the diversity was high with low variance in **GS** with slight differences between ranges. In case of **Alloy**, the diversity is similarly affected by the symmetry value: **s=0** produced low average diversity, but a high number of positive outliers. With **s=10**, the average diversity increased with decreasing number of positive outliers. And finally, with the default **s=20** value the average diversity was low. The internal diversity of **Human** models are between **GS** and **Alloy**.

7.5a illustrates the average distance between all model pairs generated in the same sequence (vertical axis) for range 7. The distribution of external diversity also shows similar characteristics as Figure 7.4: **GS** provided high diversity for all ranges (56 out of the maximum 60), while the diversity



a External Diversity

b Mutation score for model sequence

Figure 7.5: External diversity and mutation score comparison

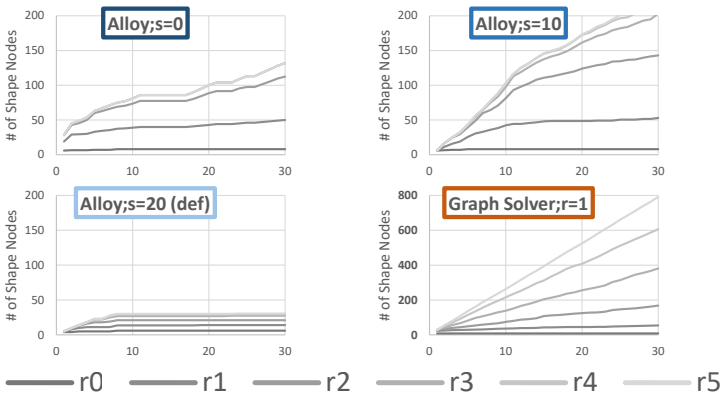


Figure 7.6: Model set coverage

between models generated by **Alloy** varied based on the symmetry value.

As a summary, our model generation technique consistently outperformed Alloy wrt. both the diversity metrics and mutation score for individual models.

RQ2: Measurement Results and Analysis. 7.5b shows the number of killed mutants (vertical axis) by an increasing set of models (with 1 to 30 elements; horizontal axis) generated by **GS** or **Alloy**. The diagram shows the *median* of 20 generation runs to exclude the outliers. **GS** found a large amount of mutants in the first model, and the number of killed mutants (36-37) increased to 41 by the 17th model, which after no further mutants were found. Again, our measurement showed little difference between ranges **r=1**, **2** and **3**. For **Alloy**, the result highly depends on the symmetry value: for **s=0** it found a large amount of mutants, but the value saturated early. Next, for **s=10**, the first model found significantly less mutants, but the number increased rapidly in the for the first 5 models, but altogether, less mutants were killed than for **s=0**. Finally, the default configuration (**s=20**) found the least number of mutants.

In Figure 7.6, the average coverage of the model sets is calculated (vertical axis) for increasing model sets (horizontal axis). The neighborhood shapes are calculated for $r = 0$ to 5, which after no significant difference is shown. Again, configurations of symmetry breaking predicates resulted in different characteristics for **Alloy**. However, the number of shape nodes investigated by the test set was significantly higher in case of **GS** (791 vs. 200 equivalence classes) regardless of the range, and it was monotonously increasing by adding new models.

Altogether, both mutation score and equivalence class coverage of a model sequence was much better for our model generator approach compared to Alloy.

RQ3: Analysis of Results. Figure 7.7 illustrates the correlation between mutation score (horizontal axis) and internal diversity (vertical axis) for all generated and human models in all configurations. Considering all models (1800 **Alloy**, 1800 **GS**, 1250 **Human**), mutation score and internal diversity shows a high correlation of 0.95 – while the correlation was low (0.12) for only **Human**.

Our initial investigation suggests that a high internal diversity will provide good mutation score, thus our metrics can potentially be good predictors in a testing context, but we cannot generalize to full statistical correlation.

Threats to Validity and Limitations. We evaluated more than 4850 test inputs in our measurement, but all models were taken from a single domain of Yakindu statecharts with a dedicated set of WF constraints. However, our model generation approach did not use any special property of the metamodel

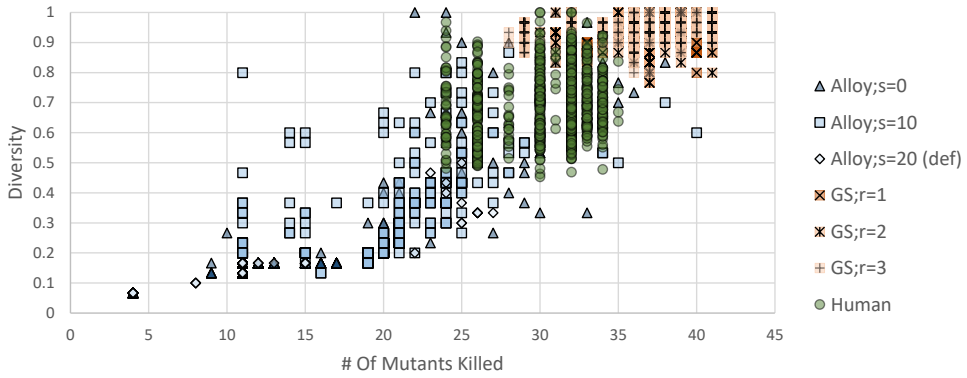


Figure 7.7: Model diversity and mutation score correlation

or the WF constraints, thus we believe that similar results would be obtained for other domains. For mutation operations, we checked only omission of predicates, as extra constraints could easily yield infeasible predicates due to inconsistency with the metamodel, thus further reducing the number of mutants that can be killed. Finally, although we detected a strong correlation between diversity and mutation score with our test cases, this result cannot be generalized to statistical causality, because the generated models were not random samples taken from the universe of models. Thus additional investigations are needed to justify this correlation, and we only state that if a model is generated by either **GS** or **Alloy**, a higher diversity means a higher mutation score with high probability.

7.5 Related work

Diverse model generation plays a key role in testing model transformations code generators and complete developement environments [RV16]. Mutation-based approaches [MBT06; DPV06; Ara+15] take existing models and make random changes on them by applying mutation rules. A similar random model generator is used for experimentation purposes in [BS16]. Other automated techniques [Bro+06; EKT09] generate models that only conform to the metamodel. While these techniques scale well for larger models, there is no guarantee whether the mutated models are well-formed. Approaches relying upon object identifiers (like [Dif]) may classify two graphs which are isomorphic to be different.

There is a wide set of model generation techniques which provide certain promises for test effectiveness. White-box approaches [Ara+15; BA05; GC14; GS15; Sch+13][J2] rely on the implementation of the transformation and dominantly use back-end logic solvers, which lack scalability when deriving graph models.

Scalability and diversity of solver-based techniques can be improved by iteratively calling the underlying solver [KJS11][C10]. In each step a partial model is extended with additional elements as a result of a solver call. Higher diversity is achieved by avoiding the same partial solutions. As a downside, generation steps need to be specified manually, and higher diversity can be achieved only if the models are decomposable into separate well-defined partitions.

Black-box approaches [Fle+07; Mot+15; Büt+12; GS15] can only exploit the specification of the language or the transformation, so they frequently rely upon contracts or model fragments. As a common theme, these techniques may generate a set of simple models, and while certain diversity can be achieved by using symmetry-breaking predicates, they fail to scale for larger sizes. In fact, the

effective diversity of models is also questionable since corresponding safety standards prescribe much stricter test coverage criteria for software certification and tool qualification than those currently offered by existing model transformation testing approaches.

Based on the logic-based Formula solver, the approach of [JSS13] applies stochastic random sampling of output to achieve a diverse set of generated models by taking exactly one element from each equivalence class defined by graph isomorphism, which can be too restrictive for coverage purposes. Stochastic simulation is proposed for graph transformation systems in [THR10], where rule application is stochastic (and not the properties of models), but fulfillment of WF constraints can only be assured by a carefully constructed rule set.

7.6 Conclusion

This chapter proposed a novel diversity metrics for models based on neighbourhood shapes [RD06], which are true generalizations of metamodel coverage and graph isomorphism used in many research papers. Moreover, we presented a model generation technique that to derive structurally diverse models by (i) calculating the shape of the previous solutions, and (ii) feeding back to an existing generator to avoid similar instances thus ensuring high diversity between the models.

We evaluated our approach in a mutation testing scenario for Yakindu Statecharts, an industrial DSL tool. We compared the effectiveness (mutation score) and the diversity metrics of different test suites derived by our approach and an Alloy-based model generator. Our approach consistently outperformed the Alloy-based generator for both a single model and the entire test suite. Moreover, we found high (internal) diversity values normally result in high mutation score, thus highlighting the practical value of the proposed diversity metrics.

Conceptually, our approach can be adapted to an Alloy-based model generator by adding formulae obtained from previous shapes to the input specification. However, our initial investigations revealed that such an approach does not scale well with increasing model size. While Alloy has been used as a model generator for numerous testing scenarios of DSL tools and model transformations [BA05; Büt+12; SBM09; Val+12][C10], our measurements strongly indicate that it is not a justified choice as (1) Alloy is very sensitive to configurations of symmetry breaking predicates and (2) the diversity and mutation score of generated models is problematic.

Chapter 8

Change Propagation of View Models with Logic Solvers

8.1 Introduction

View models are a key concept in domain-specific modeling tools to provide task-specific focus (e.g., power or communication architecture of a system) to engineers by creating a model which highlights only some relevant aspects of the system to help detect conceptual flaws. Typically multiple *view models* are defined for a given an underlying *source model*, which need to be refreshed automatically (or upon user request) upon changes in the source model.

However, such view models are read-only representations derived by a unidirectional transformation, and they cannot be changed directly. When a view model needs to be changed, the engineer is forced to edit and manually check the source model until the modified model corresponds to the expected view model. Additionally, the effects of a source change need to be observed in all other view models to avoid unintentional changes and to prevent the violation of structural well-formedness (WF) constraints. The fact that changes in the view model cannot be directly propagated back to a change in the source model hinders the use of view models in an industrial case setting.

To tackle this problem, we propose a technique to automatically calculate possible source model candidates for a set of changes in different view models. First, the possibly affected partition of the source model is identified by observing traceability links to restrict the impact of a view modification. Then the modified view models, the query-based view specification and the well-formedness constraints of the source model are transformed into logic formulae. By using an iterative technique [C10] over the Alloy Analyser [Jac02], our approach enumerates multiple (but not all) valid resolutions of the source model corresponding to the changes of view models and *the constraints of the source model*. As a result, source elements unaffected by the target change may still need to be added as a side effect to make the source model consistent. We illustrate our technique on a healthcare example. The current approach extends the conceptual overview of [C13] by presenting the technical contents in depth, and providing a first performance evaluation.

Our method provides advanced support for a class of bidirectional model transformations where each element in the view is defined unidirectionally by a declarative query [Deb+14; Gho+15]. Arbitrary changes of view models are supported and incrementally back-propagated without backward transformation rules. Our approach allows the engineer to select from multiple source candidates or to restrict the scope of considered source changes. Moreover, changes from multiple view models are merged into a consistent source model where the consistency criteria also includes the well-

formedness constraints of the source language.

Next, Section 8.2 overviews the concepts of view models and our past work on deriving view models by query-based unidirectional transformations. Then our backward change propagation approach (from view models to source models) is presented in details in Section 8.3. An initial experimental evaluation is provided in Section 8.4. Related work is overviewed in Section 8.5 while Section 8.6 concludes the chapter. This chapter is based on [C12] and [C13].

8.2 View models

In a domain-specific modeling tool, the underlying domain model is presented to the engineers in different views. These views are frequently represented as models themselves (called view models and denoted by M_V in the sequel), which are populated from the underlying domain model (called source model, M_S). One source model may populate multiple view models. In a general setting, view models can be detached from the source model to such an extent that they correspond to a different language, thus they need to be compliant with a *view metamodel* MM_V and satisfy *view-specific well-formedness constraints* WF_V.

A view model is derived from the source model by a unidirectional *forward transformation* $M_V := \mathsf{fwd}(M_S)$. This is a restricted class of model transformations where query-based declarative techniques are especially suitable [Gho+15; Deb+14]. Efficient live maintenance of a view model upon changes of the source model can be carried out by incremental transformation techniques [Deb+14; HLR06] even for multiple view models ($M_{Vi} := \mathsf{fwd}^i(M_S)$) or chains of view models ($M_V := \mathsf{fwd}^2(\mathsf{fwd}^1(M_S))$).

A forward transformation frequently creates and maintains a trace model $T = T_{obj} \cup T_{fea}$ between the source M_S and view M_V models. An *object trace* T_{obj} is a relation which connects activations of rules (queries) in the source model M_S to objects of the view model M_V. Similarly, a *feature trace* T_{fea} (i.e. reference or attribute trace) is a relation which connects rule activations in the source model M_S to references in the view model M_V.

While view models may immediately reflect live changes in the source model, view models were immutable by the engineers in our previous work [Deb+14], which restricts the use of view models in an industrial setting. In the current approach, we allow view models $M_{V1}, \ldots M_{VN}$ to be changed directly to $M'_{V1}, \ldots M'_{Vn}$ and present an approach for backward change propagation for view models to an updated source model M'_S using logic solvers.

8.2.1 Motivating scenario

Our change propagation technique will be illustrated on a case study of a remote health care system developed in the Concerto project [Con], which develops an environment for pulse and blood pressure measurement controlled by a smart phone.

> **Example 32.** An example environment is illustrated in the upper left part (1.) of Figure 8.1. Measurements of pulse and blood pressure is measured by the sensors of a mobile phone, which are executed periodically triggered daily by the phone timer. The completion event of measurements triggers the processing of sensor data: pressureDone and pulseDone. The result of the measurement is collected in reports pulseReport and pressureReport, and sent to the different hosts. In our case study, the blood pressure is sent to the general practitioner (gp) of the patient for logging, and signs of hearth failure is sent to hospitals (modeled by emergency).

Two view models are derived from this source model in our telecare example which are maintained as the source model changes. The *Dataflow* view (2.*A*) shows which Hosts will be notified about each InformationTypes, while *Event* layout (2.*B*) describes event sequences represented by Activation nodes with *after* references between them leading from an Init node to a Finish node.

Let us now assume that changes are made in views illustrated in 3.*A* and 3.*B*: 3.*A* represents a change where dataflow from Pulse to Emergency is redirected to the General Practitioner (denoted by «del» and «new»). In 3.*B*, the action dedicated to report the pulse is removed from the view, but the remaining report waits for the completion of both measurement (denoted similarly). Our technique will allow to automatically generate valid and well-formed source model candidates like (4.) that conforms to the current state of the view model.

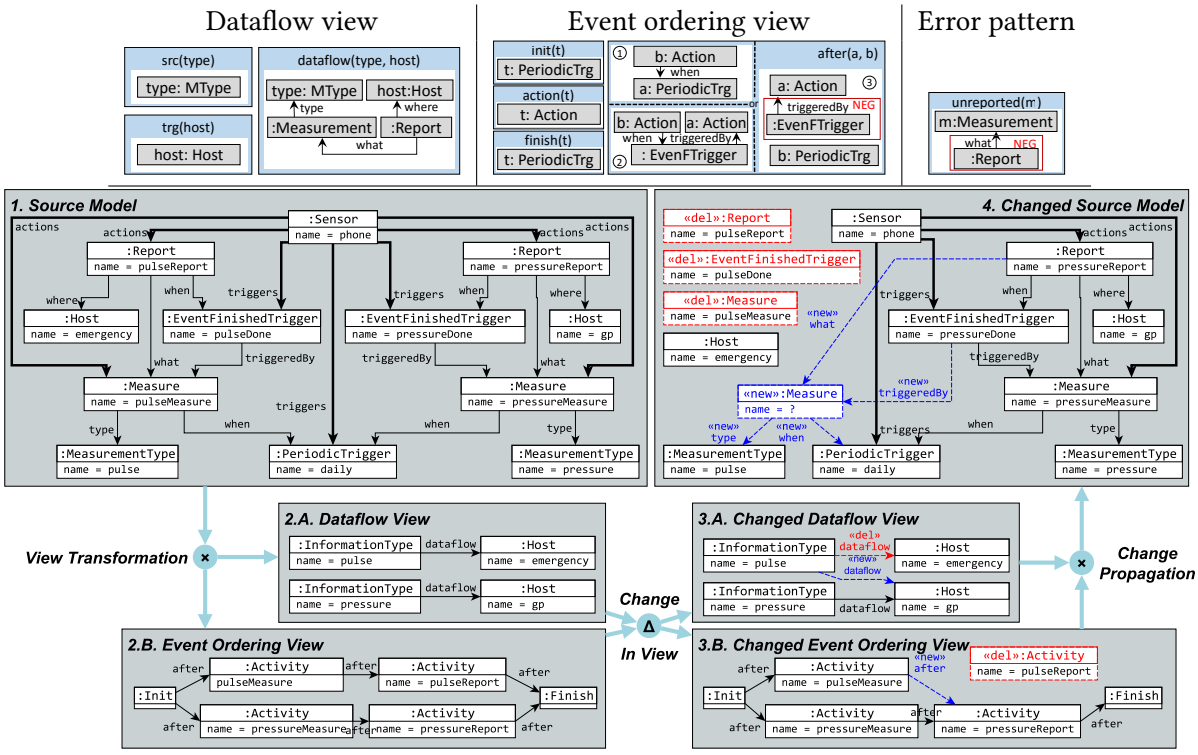


Figure 8.1: Motivating scenario on a healthcare example

8.2.2 Definition of view models

In [Deb+14] we proposed to use declarative queries as derivation rules to (i) specify new view model elements in the target M_V, and (ii) maintain a trace model between the source M_S and view M_V models based on the matching queries (patterns). A graph pattern describes structural conditions $\varphi(v_1, \ldots, v_n)$ on a model M with a combination of path and type expressions equivalent to first order logic predicate. A derivation rule consists of a pattern predicate φ and an action part where for each activation Z of predicate φ, the action part is fired. An activation Z is a function that maps all parameters $\{v_1, \ldots, v_n\}$ of predicate φ to an object of O_M in the source model $M_S = \langle O_{M_S}, \mathcal{I}_{M_S} \rangle$, $Z : \{v_1, \ldots, v_n\} \rightarrow O_{M_S}$.

To help navigation along traces, two (injective partial) lookup functions are introduced: $lookup_{VS}(v)$ maps a view object v to a predicate φ with its activation Z over the source model ($lookup_{VS}(v) = \langle \varphi, Z \rangle$ if $\langle \varphi, Z, v \rangle \in T$) and $lookup_{SV}(\varphi, Z)$ maps an activation to a view object v ($lookup_{SV}(\varphi, Z) = v$ if $\langle \varphi, Z, v \rangle \in T$). The following actions are used in derivation rules[Deb+14]:

AddObj(C, φ) Activation Z of precondition φ creates an entry $\langle \varphi, Z, v \rangle$ in the trace with a unique view object v in the view model with the corresponding type C. For each activation Z of precondition φ, there exists a unique $v \in O_{M_V}$ view object, where

$$[\![\varphi]\!]_Z^{M_S} = 1 \quad \Leftrightarrow \quad \langle \varphi, Z, v \rangle \in T \quad \Leftrightarrow \quad v \in O_{M_V}, [\![\mathrm{C}(x)]\!]_{x \mapsto v}^{M_V} = 1$$

AddRef(R, $\varphi_R, \varphi_s, \varphi_t Z_R, Z_s, Z_t$) Activations Z_R, Z_s, Z_t of preconditions $\varphi_R, \varphi_s, \varphi_t$ creates an entry $\langle \varphi_R, \varphi_s, \varphi_t, Z_R, Z_s, Z_t, \mathrm{R}, v_s, v_t \rangle \in T$ in the trace with a reference R in the view model from the source v_s to the target v_t, where $v_s = lookup_{SV}(\varphi_s, Z_s)$ and $v_t = lookup_{SV}(\varphi_t, Z_t)$.

$$\begin{array}{l}[\![\varphi_R]\!]_{Z_R}^{M_S} = [\![\varphi_s]\!]_{Z_s}^{M_S} = \\ [\![\varphi_t]\!]_{Z_t}^{M_S} = 1\end{array} \Leftrightarrow \begin{array}{c}\langle \varphi_R, \varphi_s, \varphi_t, Z_R, Z_s, Z_t, \mathrm{R}, v_s, v_t \rangle \in T \\ v_s = lookup_{SV}(\varphi_s, Z_s), \quad v_t = lookup_{SV}(\varphi_t, Z_t)\end{array} \Leftrightarrow [\![\mathrm{R}(x,y)]\!]_{x \mapsto v_s, y \mapsto v_t}^{M_V} = 1$$

AddAtt(A, $\varphi_A, \varphi_s, \varphi_t Z_A, Z_s$) Activations $Z_A, Z_s, v \mapsto t$ of preconditions $\varphi_A, \varphi_s, \varphi_t$ creates an entry $\langle \varphi_A, \varphi_s, \varphi_t, Z_R, Z_s, Z_t, \mathrm{A}, v_s, t \rangle \in T$ in the trace with an attribute A in the view model from the source v_s to the target t, where $v_s = lookup_{SV}(\varphi_s, Z_s)$.

$$\begin{array}{l}[\![\varphi_R]\!]_{Z_R}^{M_S} = [\![\varphi_s]\!]_{Z_s}^{M_S} = \\ [\![\varphi_t]\!]_{v \mapsto t}^{M_S} = 1\end{array} \Leftrightarrow \begin{array}{c}\langle \varphi_A, \varphi_s, \varphi_t, Z_R, Z_s, Z_t, \mathrm{A}, v_s, t \rangle \in T \\ v_s = lookup_{SV}(\varphi_s, Z_s)\end{array} \Leftrightarrow [\![\mathrm{A}(x,t)]\!]_{x \mapsto v_s}^{M_V} = 1$$

As a result, we obtain a declarative formalism for defining view models with execution semantics compliant with incremental and live graph transformations [Rát+08]: when a new activation of a forward rule is detected, the corresponding view elements are created and when a previously existing activation of a forward rule disappears the related view elements are removed. However, this is still a restricted subclass of model transformations since (1) each rule creates exactly one new element (object or reference) in the view model, and (2) the transformation is monotonic in the sense that a view element always depends on the existence of a match of a positive pattern (i.e. we disregard cases when a view element is created when a pattern cannot be matched).

Example 33. Queries used for defining the views of our motivating example are depicted in the top of Figure 8.1. For the *Dataflow* view, src query selects all the MeasurementType to create InformationType instances in the view, while trg is responsible for creating Host instances from the Host objects in the source model. The dataflow pattern has two parameters and selects all the type and host pairs that are connected to each other via a Measurement and a Report objects. The action part of the rule will create an edge between an InformationType and a Host associated with the two parameters upon a match appears for the pattern. Similarly, the *after* pattern is responsible for setting the after edge between view model objects. In case of (1.), the edge will go from an Init object to an Activity, (2.) describes the connection between two Activity, while (3.) activates when an Action (common ancestor of Report and Measurement) is not triggered by any EventFinishedTrigger. The init and finish queries create Init and Finish objects in the view for each PeriodicTrigger objects in the source model. Finally, the action query builds Activity instances from all Action objects.

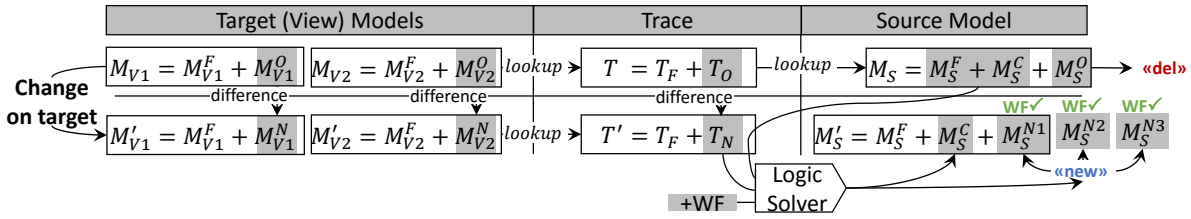


Figure 8.2: Overview of backward change propagation

8.2.3 Characterization of query-based transformation of view models

S. Hidaka et al. [Hid+15] classifies bidirectional transformation approaches based on their features. According to it, our previous work [Deb+14] is a *syntactic approach* for *forward functional* transformation of *MDE artifacts*. View models contain *no complement information*, hence they are regular models without additional annotations. These models are *total targets* as the full view model is specified by the consistency relations. Definition of a view model is *unidirectional*, however the expressiveness of definition is *Turing incomplete*. Forward propagation of the *operation-based* changes are *live*, *incremental* and executed *automatically* that also maintains *explicit traces*. However, that approach has *incomplete change support*, thus only the modification of the source model is supported.

8.3 Backward change propagation by logic solvers

8.3.1 Overview of approach

We present a novel approach to back-propagate view model changes into a consistent source model by using logic solvers. An overview of our approach is depicted in Figure 8.2 where *target (view) models* $M_{V1}, M_{V2}, \ldots$ are derived from a *source model* (M_S) based on the matches of the view definition queries (in the source model), and a *traceability model* T is built and maintained during the forward transformation. Now the engineer makes changes to the view models, which leads to changed view models $M'_{V1}, M'_{V2}, \ldots$. The goal of our approach is to calculate (one or more) source models $M_S{}'$ which corresponds to the change, maintain T', and *satisfy additional constraints of the source model*.

A change in the view model can be separated into two partitions: the fixed model partition M^F_{Vi} denotes a partial model which remains unchanged, while M^O_{Vi} is updated to a new M^N_{Vi} (cross-references are included in M^O_{Vi} and M^N_{Vi}). The change is propagated consequently to the trace model: T_O contains the invalidated trace links, T_N symbolizes the new links to be created and T_F contains the remaining trace links which are not affected by the change. Along the traces, the change can be propagated back to the source model by identifying unchanged, newly activated and deactivated matches of queries in the source model. By analyzing the impact of changing matches in T_N, the source model can be partitioned into three partial models: a fixed part M^F_S contains the elements which cannot be changed, a changing part M^C_S containing the objects which can be modified, and a obsolete part M^O_S containing the objects which can be deleted. The changing trace T_N declaratively specifies structural constraints on the $M^C_S \cup M^O_S$ model which have to be satisfied in order to ensure the consistency of the forward transformation.

A possible solution M'_S for the changed parts $M^C_S \cup M^{Ni}_S$ has to (1) match the fixed part M^F_{Vi}, (2) the requirements defined by the changed matches defined in T_N, and (3) additional domain-specific *WF* (of the source model). All these constraints are transformed into a first-order logic problem to be solved by a *logic (SAT/SMT) solver* following [C10]. The solver provides several (but not necessarily

all) possible valid solutions for M_S^C and it may create new objects in M_S^{Ni} from which from the model developer may choose the most appropriate one M_S^{Ni}.

As a summary, our approach integrates two novel techniques into an interactive workflow: an inverse impact analysis by *change partitioning* (Section 8.3.2) separates the affected and unaffected parts of the source model, while *partial model generation* creates candidate source models that correspond to the new target views (Section 8.3.3) and source constraints.

8.3.2 Change partitioning

The rationale of change partitioning is as follows: (i) if a view model element does not change, the associated traces cannot be changed; (ii) otherwise, if the change of a source object may induce a valid view model it has to be selected. However (iii) unnecessary source changes should not happen. To ensure these conditions, an impact analysis of the changes has to be conducted to identify the affected part of source model which can be modified.

In general, partitioning aims to select a relatively small fragment of the source model which needs to be changed in order to propagate the change back. However, there is not any ultimate solution for this: larger selections may introduce unnecessary changes in the source model, while smaller changes may result be unfeasible propagation. A detailed discussion of a change partitioning technique is detailed in Section A.5. Here, we just give an example.

Example 34. In our example of Figure 8.1, the affected part for the deletion of dataflow edge from the view model is calculated as follows. The trace model T_F stores that the existence of dataflow edge is related to the `dataflow` pattern where the activation binds pattern parameters `emergency:Host` and `pulse:MeasurementType` to objects in the source model M_S. The affected part of the pattern includes all objects related to the match {*emergency,pulse*}, and the affected part of the constraints includes internal variables {*pulseReport, pulseMeasure*}. However, *pulseMeasure* is also responsible for an after edge in the other view model, thus it can be changed but not allowed to be deleted from the source model. At this stage, the user may manually move objects between these categories (M_S^F,M_S^C,M_S^O) to refine his/her intention on source candidates.

$$M_S^C = \{pulseMeasure\}$$
$$M_S^O = \{emergency, pulse, pulseReport\}$$
$$M_S^F = \{\text{rest of the objects}\}$$

8.3.3 Model generation by logic solvers

Logic solver based model generation for a domain specific language is an actively researched area. Instance models can be created to provide models that satisfies required properties, test cases or to create counterexamples for false language properties [J2], and incremental model generation techniques [SFC12][C10] are able to take advantage nearly finished partial instance models.

8.3.3.1 Logic representation of view models

In general, solver-based model generation takes the logic representation of the metamodel *MM* and well-formedness constraints *WF* as a theorem over a signature $\langle \Sigma, a \rangle$ to synthesize conforming instance models $M = \langle O_M, \mathcal{I}_M \rangle$ with $M \models MM$. A model query can be represented as a relation

$Q_i(v_1, \ldots, v_n)$ over objects of the model which is evaluated to true only if some objects satisfy the translated query specification $\varphi(v_1, \ldots, v_n)$:

$$\forall v_1, \ldots, v_n : Q_i(v_1, \ldots, v_n) \Leftrightarrow \varphi(v_1, \ldots, v_n).$$

A match m_i is represented as a map $Z : \{v_1, \ldots, v_n\} \rightarrow O_M$. Consistency with the view model is ensured by a formula set *View*, which controls the matches of query predicates. Therefore, the generation of a valid and consistent view model is specified as $M \models Meta \wedge WF \wedge View$.

Example 35. In the logic equivalent of our running example the type, what and where references are modeled by relations. The specification of $dataflow(t, h)$ pattern can be represented by the following predicate:

$$dataflow(t, h) := \exists i_1, i_2 : \mathsf{type}(i_1, t) \wedge \mathsf{what}(i_2, i_1) \wedge \mathsf{where}(i_2, h).$$

Here t and h has to be connected by a specific path of relations. In our example the changed dataflow model has two matches: $Z_1 = t \mapsto c_1^1, h \mapsto c^2$ and $Z_2 = t \mapsto c_2^1, h \mapsto c^2$, as both types are forwarded to the general practitioner. With the following axioms added to the logic problem it can be ensured that there are exactly two matches of the pattern (as defined by the dataflow view), and each match is unique:

$$\forall h, t : P_{DF}(t, h) \Leftrightarrow (t = c_1^1 \wedge h = c_1^2) \vee (t = c_2^1 \wedge h = c_2^2)$$

8.3.3.2 Incremental transformation of view models

In case of view models, the affected part $M_S^C \cup M_S^O$ typically remains proportional to the change, thus M_S^F explicitly defines most of the generated models. Incremental model generation techniques like [C10] are able to take advantage of fully specified model fragments, and encode the graph generation problem in a way that the problem is proportional to the newly created fragment, as in Chapter 6. In the following, we give a brief description of the mapping technique.

- **Objects**: the object set is partitioned into three subsets: M_S^F is mapped to the fixed objects O_F, M_S^C stands for the changing objects O_C and finally O_N replaces the objects which are removed M_S^O. In general, predicates dealing with M_S^F are interpreted, thus the solver already knows its truth evaluation.

- **Classes**: Each class predicate C is also separated into three subsets: a fully interpreted $\mathsf{C_F}$ defined over O_F, a fully interpreted $\mathsf{C_C}$ defined on the changing objects O_C, and the uninterpreted $\mathsf{C_N}$ over O_N. In summary, the solver has to interpret only relations of $\mathsf{C_N}$.

- **References**: Reference predicates are also separated to multiple smaller relations: $\mathsf{R_{OO}}$ represents the interpreted relation between fixed objects, $\mathsf{R_{CC}}$ the reference between changing objects, and $\mathsf{R_{NN}}$ represents the reference between new objects. Additionally, uninterpreted cross-references have to be added for references connecting these regions: $\mathsf{R_{OC}}$, $\mathsf{R_{ON}}$, $\mathsf{R_{CO}}$, $\mathsf{R_{CN}}$, $\mathsf{R_{NO}}$, $\mathsf{R_{NC}}$. While only $\mathsf{R_{OO}}$ is interpreted from the nine new relations, it contains the most references.

- **Attributes**: Attribute predicates are also separated into three partitions for O_F, O_C and O_N.

- **Model Queries and Matches**: Model queries are separated into multiple queries, each parameter can be bound to O_F, O_C and O_N. This might add several relations to the logic problem, but the unchanged matches are already interpreted. In the construction of the constraint set $View$, the uniqueness of non-interpreted matches needs to be ensured

Compared to solving the model generation problem as a whole, our incremental approach enables the logic solver to handle much fewer variables as a large fragment of the model is already interpreted (prior to calling the solver). The downside is that constraints become more complex as they have to be separated into those groups above. However, we expect that most predicates remain interpreted, which is beneficial for the solver.

8.3.4 Properties of our approach

Our approach has the following properties (based on [Hid+15]):

1. *Full operation support on views:* View models can be edited as regular models, while the technique ensures consistency between source and view models.
2. *Implicit backward consistency:* View models derived from a source model candidate $M'_S = \text{fwd}^i(S')$ are isomorphic to the view models $V'_1, \ldots V'_n$.
3. *Delta-based and offline back-propagation:* After making changes on the target models, our approach generates source model candidates from a stable state of the views and the changes in the traces. Upon a sequence of (possibly concurrent) view changes is applied it leads the view models to a new stable state. Then the difference between the previous and current state of the views can be propagated back even if some changes are contradictory or inconsistent by themselves.
4. *Interactive execution:* There might be several source candidates for a view model change on which the solver can iterate, starting from the smallest solution. The developer or a selection strategy can select the most suitable one from the sequence of valid solutions.
5. *Well-formedness:* S' satisfies the well-formedness constraints of the source domain $S' \models WF$.
6. *Incrementality:* A view change can change only the affected part of the source model. Furthermore, if a view model element is not changed, it is not affected by the back propagation (i.e. there are no elements that are removed then re-added).
7. *Hippocraticness of unaffected partition:* If a view model element is not changed, the associated source model part has to remain unchanged. This implies the standard notion of Hippocraticness: if there is no change in the view models, and the views are consistent with the source, the source model remain unchanged.

Conditions 1-4 are related to usability and they connect the new backward propagation technique to our previous forward transformation approach. Some form of consistency is ensured in several approaches (e.g. [Cic+10]) but we also incorporate WF constraints of the source language. Hippocratic behavior defined in [Ste08] states that a backward or forward transformation must not modify the source or the target model if they are already consistent. In Property 6. and 7. we define a stronger requirement which states that consistent partitions of the source and target models should not be modified. This constraint simultaneously keeps most of the source model untouched and makes the deduction phase more efficient by limiting the task to partial models.

8.4 Experimental evaluation

In order to evaluate the performance of the key step of our backward change propagation technique we have conducted initial measurements on a prototype implementation in the context of the running example as case study (taken from the CONCERTO project). The measurement scenarios and the results are available on GitHub[1]. Our measurements aim to address the following questions:

Q1 What is the influence of the size of the source model, the size of the target change and the scope of the source model (i.e. the number of newly created source objects) on the runtime of the solver?

Q2 What is the difference between incremental model generation and full model generation (like in [Gho+15]) with respect to performance and the quality of results?

Q3 What are the (solver-specific) limitations of our backward change propagation technique?

Our evaluation exclusively focuses on assessing the performance of the model generation step for the source model (detailed in Section 8.3.3), and excludes the performance evaluation of change partitioning (Section 8.3.2). In our initial experiments, we experienced that both the change partitioning time (i.e. the selection of affected source elements) and the forward transformation time is negligible (less than 1 second for the largest problems we measured) compared to the time required to solve the logic problem by Alloy. Thus performance limitations are dominated by the latter.

8.4.1 Change propagation problem generator

In order to measure the performance of our technique we extended the running example visible in Figure 8.1 to a change propagation benchmark, which can be parametrized and scaled by the *size (s)* of the source model and *the number of changes (c)* in the views. We have created valid health care models illustrated in Figure 8.3 in the following way :

1. First, a Sensor is created with a PeriodicTrigger.
2. Two MeasurementTypes are added to the model, which are measured by two respective Measures activated by the periodic trigger. Then two Reports are added to the model, which are triggered by two new *EventFinishedTrigger*s waiting for the measurements of two different types. Then the result is reported to two newly created Hosts.
3. Step 2 is repeated s times, which means that the model has $2s$ MeasurementTypes, $4s$ Measures, $2s$ Reports, $2s$ EventFinishedTriggers and $2s$ Hosts.
4. The view models are derived, which creates in a dataflow model with $2s$ Hosts and $2s$ InformationTypes $4s$ dataflow references, and an event ordering model with 1 Init, 1 Finish, $6s$ Activities and $10s$ after references.
5. Then changes are applied to the view models: c MeasurementType and c Hosts with their dataflow references are randomly removed from the first view model, and a new Action is added to the event ordering view.

As a result, we obtain non-trivial source and view models, while the random changes of the view model remain semantically meaningful.

[1] `https://github.com/FTSRG/publication-pages/wiki/incremental_backward_change_propagation_of_view_models_by_logic_solvers`

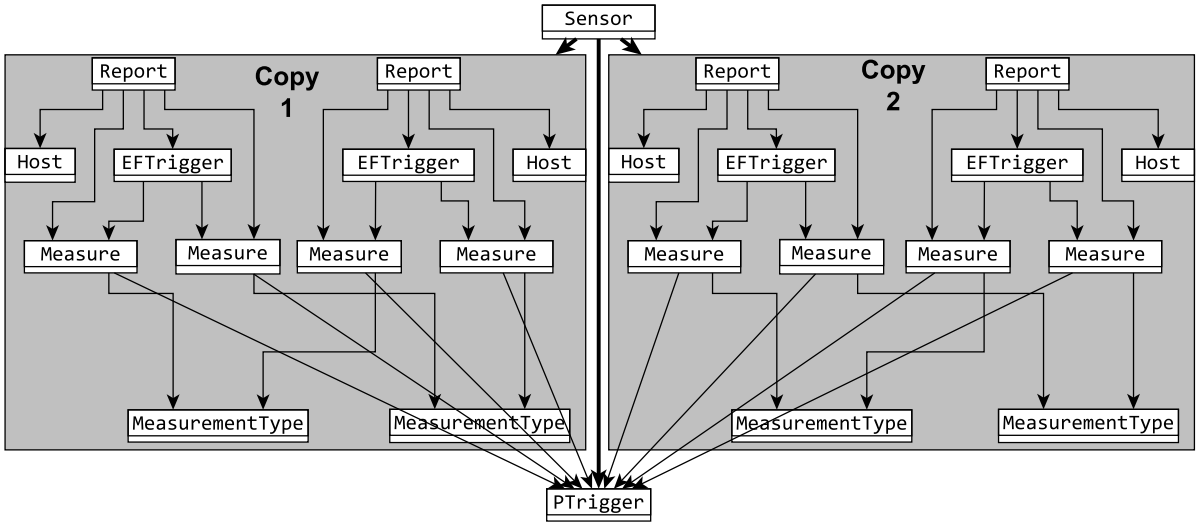


Figure 8.3: Structure of the generated source models

8.4.2 Measurement setup

Each model generation task was executed on the generated healthcare change propagation problems using the Alloy Analyzer (with SAT4j-solver). Each change propagation problem is solved with our incremental solution which generates only the affected part in the source model. As a baseline, we compare it to a solution conceptually similar to [Gho+15] which generates full models for a view model. The full model generation is achieved as a corner case of the incremental generation where the changed view models are interpreted as if they were newly created, and no part of the source model is preserved.

We measured the runtime of Alloy Analyzer, which consists of an initial conversion to a conjunctive normal form and then solving the SAT-problem by the back-end solver. We executed each measurement 5 times, then the average of the execution times was calculated. The measurements were executed with a 120 second timeout on an average personal computer[2]. The memory usage of the solver was always below 2 GB.

8.4.3 Measurement result

The execution times of our measurements (see Figure 8.4– Figure 8.7) are given in seconds.

- $|S|$ denotes the number of objects in the original source model (M_S);
- $|\Delta V|$ denotes the size of the target change, i.e. the sum of removed and newly created matches in the view model (M_V^O and M_V^N);
- Finally, $|N|$ denotes the source scope, i.e. the number of new objects in the changed source model candidates (the number of objects in M_S^N). It is equivalent to the scope of model generation in Alloy when incremental technique is used.

[2]CPU: Intel Core-i5-m310M, MEM: 16GB, OS: Windows 10, Reasoner: Alloy Analyzer 4.2 with sat4j

In case of full model generation where the each model object is newly created, the size of the original model is subtracted from this value, so the two techniques are comparable with respect to the number of objects.

8.4.4 Increasing model size

First, Figure 8.4 displays the runtime results of cases where only one change is performed ($c = 1$, $\Delta V = 8..9$). Eleven different series are measured, where the change propagation is solved with source scope of $|N|$ new elements (ranging from 0 to 10) with increasing the size of source models $|S|$ up to 123 objects. With 0 new elements, the problem was unsatisfiable, otherwise valid solutions were created.

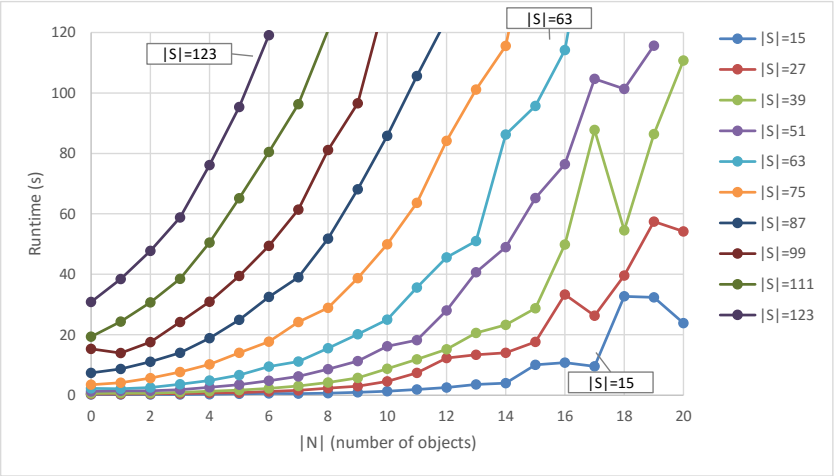


Figure 8.4: Runtime with increasing source model size (vertical axis: size of source model, series: size of source scope, change size: 1)

The results show a polynomial (cubic) increase of run-times in the size of the original source model, and it always harder to solve the problem with increasing source scope size. Additionally, the solver has a similar characteristics regardless a problem is satisfiable or unsatisfiable. Fortunately, smaller solutions can be retrieved more efficiently, which is typically preferred over solutions with unnecessary elements.

We also depicted the run-times in Figure 8.5 while further increasing the size of source scope. This shows that the source scope needs to be decreased when the size of source models increases in order to obtain the same run-time.

8.4.5 Increasing target change size

Figure 8.6 displays the results with larger target change sizes. Here, the vertical axis displays the source scope, i.e the number of new objects added to the source model by the solver, and different series shows the run-times of target change size ranging from 8 up to 43 changes, and the measurement is repeated for two model sizes. For the smallest change one new element is needed to successfully create solutions, two new object is needed for the next change size, and so on, so the largest changes needed at least 5 new objects.

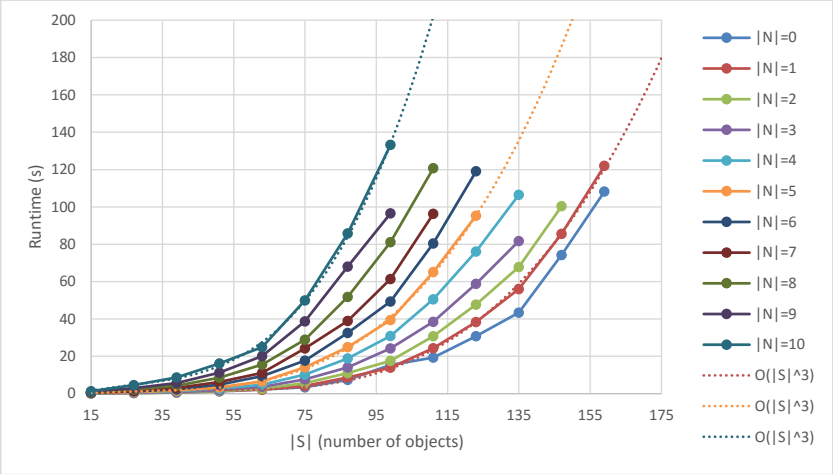


Figure 8.5: Runtime with increasing source scope size (vertical axis: size of source scope, series: size of source model, change size: 1)

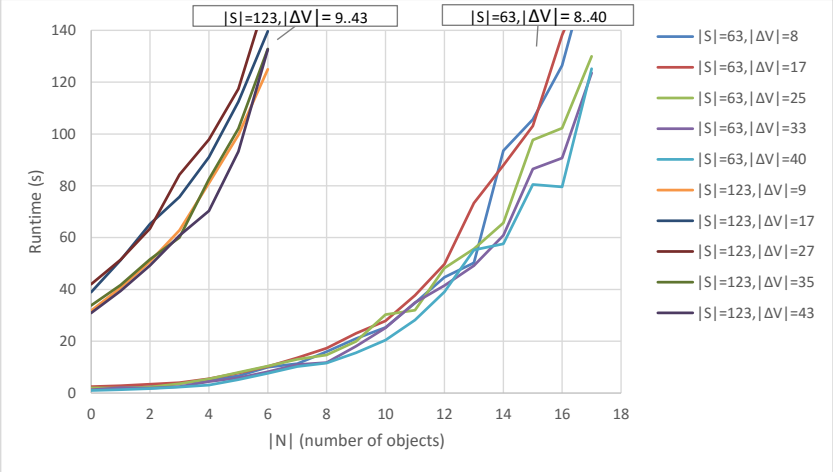


Figure 8.6: Runtime with different change size (vertical axis: number of newly added objects, series: size of model, size of change)

The results shows that the run-time is not directly affected by the size of the change, different change sizes have the same complexity. However, a larger target change size usually implies a larger source scope, which, of course, influences the run-time of the solver. In other terms, if the consequences of a large target changes are attempted to be propagated back to the source model with a small scope size, then the logic solver will conclude that the problem is unsatisfiable. While if we also increase the scope size, then the problem might become satisfiable - but the logic solver may fail to find a solution due to its performance limits.

Therefore, based on our measurements of increasing model and change size, our first question can be addressed as follows:

A1 The runtime of the solver is a polynomial function of the original size of the model and the

source scope size (i.e number of newly added elements). The size of the target change increases the size of the source scope required for a solution.

8.4.6 Incremental vs. full generation

In Figure 8.7 the runtime of the incremental and full model generation is depicted with respect to the size of the source model size and the source scope. Only results of two small source models (15 and 27) are presented as on all larger cases, the full model generation technique was unable to provide a solution.

In comparison, the incremental model generation technique performed much better: it was able to solve some nontrivial problems, and in general, it was orders of magnitude faster than full generation.

From the perspective of model quality, the full model generation approach redirected several relations where the target view model was unchanged (e.g. unchanged dataflows), which might be undesired in change propagation scenarios. On the other hand, the full change propagation may find solutions which requires changes in partitions categorized as unaffected by our change partitioning, which can be only retrieved by manual configuration of the affected part in case of incremental generation.

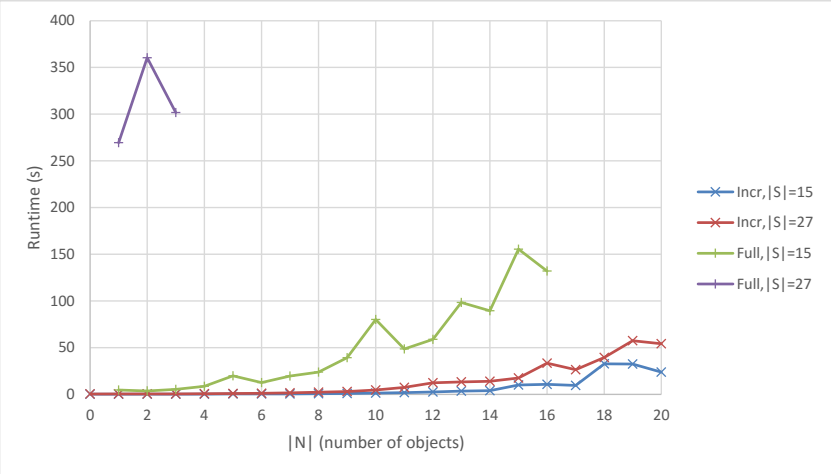


Figure 8.7: Incremental versus full model generation

The difference between incremental and full model generation for backward change propagation purposes can be summarized as follows:

A2 Incremental model generation provides better performance for source models of increasing size for a given scope. Full model generation may be able to retrieve some hidden solutions. Unlike full model generation, incremental generation is able to detect if a change propagation setup is unsatisfiable for a given scope.

8.4.7 Limitations

Our initial experimental results revealed several limitations of our approach most of which were caused by using Alloy as an underlying logic solver. When investigating the runtime for model generation, we found that over 80% of the time is spent for an initial conversion to Conjunctive Normal

Form (CNF) prior to starting the model finding step. In fact, CNF conversion took over 99% of the time for large models with $|S| = 160$ elements. The largest source model our technique was successfully executed had 243 objects, and the run-time for propagating a single target change was over 20 minutes (again, spent dominantly on CNF conversion).

While our incremental query-based forward transformation scales to source models orders of magnitude larger, we can state as a conclusion that such scalability cannot be achieved for incremental backward change propagation when using Alloy as a solver.

However, the fact that model generation time is dominantly spent on CNF conversion in Alloy (and not on the actual SAT/SMT-based model finding), this may trigger future research to replace Alloy with a dedicated incremental model finder for EMF-based models.

Furthermore, when investigating small backward change propagation problems with our approach, one may gain domain-specific insights to identify specific target changes which can be always mapped to a source change (e.g. by some rule-based transformation techniques).

A3 Incremental backward change propagation using model generation by Alloy scales only to small models compared to incremental forward transformations. But an incremental model generation technique scales significantly better than full model generation, which is a hope for dedicated future model generators.

8.4.8 Threats to validity

Finally, let us investigate the most critical threats to validity of our conclusions.

- While our case study is relatively complex (as it originates from the CONCERTO project), our measurements were executed only on a single case study, thus our findings may not be applicable in a more general context. However, our model generator approach has strong roots in [C10] where efficiency of incremental model finding by using existing model solvers were assessed, and the experimental results point to similar limitations. Moreover, our negative results (e.g. poor performance of Alloy) are more likely generalizable for other model generation and transformation scenarios.
- We excluded the time of forward transformation and change partitioning from our measurements - as initial experiments showed that they were less than a second.
- As execution times of Alloy quickly started to increase, we only had 5 measurements for each case, thus we did not carry out a full-fledged statistical analysis of our results. Correspondingly, our findings are softer (more qualitative than quantitative).

8.5 Related work

Most existing view model synchronization techniques use bijective transformations where transformations can be executed in both the forward and reverse directions such in lenses [Fos+07], injective functions [MHT04] or ATL [Xio+07]. Triple Graph Grammars (TTG) [Sch94] are a well-know approach for model synchronization [Her+15] where the forward and reverse transformation rules are derived automatically from a bidirectional rule definition. A special class of TGG is View TGG [Anj+14] which is specialized for efficient update propagation. As a fundamental difference, our approach uses patterns to define the well-formedness constraints and the view instead of generative graph grammars.

Approach	Logic Solver	Traceability	WF const.	Partial Model	Interoperability
ATL[Xio+07]	-	-	-	-	+
TGG[Sch94]	-	+	-	-	+
QVT-R[OMG]	-	+	+	-	+
QVT-R with Alloy[MC13]	SAT Solver	-	+	-	+
JTL[Cic+10]	ASP	-	-	-	+
MTE with MILP[CK13]	MILP	-	-	-	+
EMF Views [Bru+15]	-	+	-	+	+
F-Alloy [GK15]	SAT Solver	-	+	-	+
QueST[Gho+15]	SAT solver	+	+	-	+
Our solution	SAT solver	+	+	+	+

Table 8.1: Comparison of related approaches

Most closely related approaches for view synchronization are listed in Table 8.1. To compare them to our approach, we use several characteristics to guide the structure of this section.

Using Logic Solvers. Using logic solvers for generating possible source and target candidates is common part of several approaches. [Cic+10] uses Answer Set Programming, [CK13] maps the problem to Mixed Integer Linear Programming. Those approaches use solvers to select model elements which may alter the matches related to view model changes (similar to the calculation of the affected part). As a difference, our approach takes the whole DSL specification into the account, to change source model elements which are only implicitly related to the view changes, caused by the interaction of the metamodel, the WF constraints and other view models.

[MC13] uses Alloy to generate change operations on the source model which leads to a modified source model which is (i) well-formed and (i) consistent with the changed target model. As a difference, our solution creates the changed partition (and not the change operations). [Gho+15] and [GK15] converts the transformation to Alloy similarly, but do not handle WF constraints of the source model, and changes the whole source model. By selecting the affected part, our solution likely has better scalability as it has to manage less objects: [GK15] scales up to 20 objects in the source and target models in total, and no other measurement is given. This technique is also suggested in the future work of [MC13].

Traceability Links. For the backward propagation of changes, use of traceability links is a well-accepted approach to define which part of the source model has to be updated upon a change on the target model. In [Sch94], these links are stored as a *correspondence model* where their maintenance is derived from the TGG rules. [OMG] also specifies *trace* classes to facilitate and maintain traceability links. [Gho+15] stores traceability links in Alloy[Jac02] as a bijective mapping. [Bru+15] uses a *weaving* model that stores the traces of references between different models in the view, however all objects in the view model act as proxies to an object in the source model. Our solution builds and maintains a traceability during the forward propagation of changes. Moreover, we reuse the information stored in the traces to improve the affected part calculation for changes in the view.

Well-formedness constraints. To avoid the calculation of *ill-formedness* source models after the backward propagation of a view model change, well-formedness constraints should be taken into account. This property is supported in the specification of [OMG] and by [MC13] and in our approach.

Partial synchronization. [Het10] defines *partial synchronization* to apply the changes of target model only to the relevant part of the source model. This reduces the number of possible source candidates. Our approach identifies the fixed part of the source model, that cannot be changed, and

selects the complement of this part which will be the basis of constraints. While [Het10] defines a formal framework for model synchronization, our solution can be interpreted as an efficient and view-model specific realization of it.

Partial models have certain similarity to *uncertain models*, which offer a rich specification language [FSC12a; SC15] amenable to analysis. Uncertain models provide a more expressive language (called MAVO annotations) compared to partial snapshots (which implements only annotation V and O from MAVO) but without handling additional WF constraints. Such models document semantic variation points generically by annotations on a regular instance model, which are gradually resolved during the generation of concrete models. An uncertain model is more complex (or informative) than a concrete one, thus an a priori upper bound exists for the derivation, which is not an assumption in our case.

Concrete models compliant with an uncertain model can synthesized by the Alloy Analyzer [SFC12], or refined by graph transformation rules [Sal+15]. Each concrete model is derived in a single step, thus their approach is not iterative like ours. Scalability analysis is omitted from the respective papers, but refinement of uncertain models is always decidable.

We believe that our contribution is novel in the context of view model synchronization in the sense that the effects of uni-directional and non-injective forward rules are reversed by mapping models, WF constraints and rules into first-order logic and then using iterative calls to a SAT-solver. Furthermore, the consistency criteria for the derived source models is stricter as it includes WF constraints of the source language (and not only consistency constraints of the transformation). Moreover, a fix partial model is specified upon a change in the target model using traceability links maintained during the forward propagation. Finally, iterative and incremental calls to logic solvers scales better then a full model generation run.

8.6 Conclusion

In this chapter, we presented an incremental backward change propagation approach from view models to source models (in full details compared to [C13]), which (1) provides a change partitioning technique to separate possibly affected and unaffected partitions of the source model, (2) transforms the source model partitions, the queries and WF constraints of the source language to a logic problem and (3) generates well-formed and consistent source model candidates by the Alloy Analyzer. This way, valid source candidates can be deduced in case of multiple view models and when backward transformation rules are not explicitly specified.

An initial experimental evaluation of our approach was carried out in the context of a health care model (taken from the CONCERTO Atremis European project [Con]), which demonstrated that (A) our incremental model generation approach scales much better compared to generating the full source model in a single call to a logic solver, but (B) its scalability is severely limited by the Alloy Analyzer (especially, by an initial internal conversion to a CNF form).

Chapter 9

Validation of Complex Domain-Specific Languages

9.1 Introduction

In case of complex, standardized industrial domains (like ARINC 653 [ARI] for avionics or AUTOSAR [AUT13] in automotive), the sheer complexity of the DSL is a major challenge itself. 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. Moreover, domain metamodels are frequently extended by derived features, which serve as automatically calculated shortcuts for accessing or navigating models. The specification of derived features can also be inconsistent, ambiguous or incomplete.

The mathematical precise validation of DSL specifications themselves have been attempted by only few approaches so far [JLB11; JS06], and even these approaches lack a systematic validation process. 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 as part of a software tool qualification process in order to provide trust in their output. Therefore software tool qualification raises several challenges for building trusted DSL tools for a specific domain.

The main objective of this chapter is to propose an *automated validation framework* to formally check the specification of DSLs. For that purpose we carry out a wide range of validation tasks by automated theorem proving based on this formalization to show *consistency*, *unambiguity* and *completeness*, *subsumption* or *equivalence* of a DSL. To decrease the development time and cost of DSL tools, we aim to detect design flaws in the early phase of DSL development by highlighting validation problems to the developer directly in the DSL tool itself by back-annotating analysis results. As a side effect, our validation framework can also be used for *generating prototypical well-formed instance models* for a DSL, which can be used for synthesizing test cases, for instance.

Language level validation is a very challenging task because the analysis has to cover an infinite range of possible design models, which necessitates symbolic approaches. The language elements are representable by sets and relations which makes first order logic suitable to formalize them. Different constraint languages attached to the modelling languages are semantically close to first order logic, extending them with additional elements, like transitive closure. However, reasoning over a language level problem is undecidable in general. Fortunately logic solvers have become more and more powerful.

- SAT-solvers specialized for graph problems are capable of checking large range of models in order to generate counterexamples of bounded size if the validated property is not satisfied by the target DSL, but they cannot prove the correctness.
- On the other hand, SMT-solvers are able to efficiently handle complex logic problems with unlimited domain by solving them with a combination of background theories. An advanced SMT-solver contains decision procedures for the most common logic fragments, like the effectively propositional [PMB08].

Abstraction is a key element of automatically solving logic problems. First, constraint languages use higher order language elements like transitive closure which cannot be explicitly represented in first order logic. Additionally, a more generic problem can be solved efficiently if it is represented in the target scope of the backend solver. Thus, with suitable approximations, language properties cannot be directly constructed in the target logic formalism.

In the chapter, we make the following contributions:

- We propose an approach for the validation of DSLs which covers the handling of metamodels, well-formedness constraints, derived features and partial snapshots. Our aim is to derive effectively propositional formulae wherever possible, which is an efficiently analyzable fragment of FOL. We also propose approximation techniques to handle complex language constructs which cannot be represented in FOL.
- In order to systematically carry out the validation process for a DSL, we propose a validation workflow, which consequently investigates each language feature to check consistency, completeness and unambiguity (for derived features) and subsumption or equivalence (for well-formedness constraints).
- We provide prototype tool support which takes EMF metamodels with derived features, instance models, constraints captured by graph patterns of the Viatra framework or in OCL as input, and carries out DSL validation using back-end reasoners. Validation results are back-annotated to the source DSL specification and to the initial partial model therefore language engineers may inspect those results as regular instance models.
- We carry out an initial performance evaluation on various DSL validation tasks using a motivating example from the avionics domain using the powerful Z3 SMT solver built on high-level decision procedures and Alloy (based on a SAT-solver) as automated back-end reasoning tools.

This chapter is structured as follows: next Section 9.2 summarizes the Avionics Architecture modeling case study, Section 9.3 defines the proposed validation approaches, which are illustrated on the case study in Section 9.4. Measurement results are illustrated in Section 9.5. Section 9.6 summarizes the related work, and Section 9.7 conclude the chapter. This chapter is based on [C9] and [J2].

9.2 Running example: Avionics modeling environment

9.2.1 Motivating scenario

Trans-IMA [Hor+14] 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 aimed to (i) define a model-driven development process for allocating software

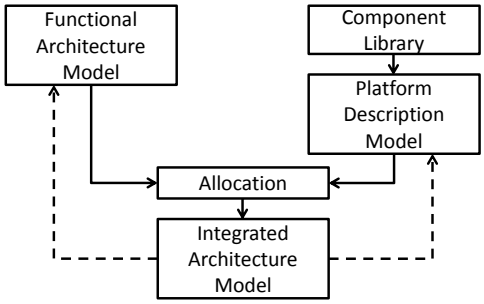


Figure 9.1: High-level overview of the Trans-IMA project

functions captured as Simulink models [Mat] over different hardware architectures and (ii) develop domain-specific languages and tools for supporting the definition of the allocation process.

The high-level overview of the Trans-IMA challenge is illustrated in Figure 9.1. 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.

1. *Functional Architecture Models* can be imported from industrial language and tools such as AADL [SAE] or Matlab Simulink [Mat] to capture the functional description of different systems. In my thesis work, I focuse on the validation of this DSL fragment.

2. Then the system architect specifies the *Platform Description Model* from the elements of the *Component Library* defining the available hardware elements.

3. In the next step the system architect *allocates* all 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.

4. Finally, when the allocation is complete and fulfills all safety and design requirements the *Integrated Architecture Model* is automatically synthesized to enable simulation in Matlab Simulink.

Trans-IMA DSL contains 118 classes, 90 attributes and 170 references, where 56 features were marked as derived (about 20% of total) and each was specified by a corresponding model query. The design rules are defined by 31 well-formedness constraints.

9.2.2 Metamodeling

Metamodels define the main concepts, relations and attributes of the target domain to specify the basic structure of the models.

> **Example 36.** A simplified metamodel for functional architecture is shown in Figure 9.2. The FunctionalArchitectureModel element represents the root of a model, which contains each Function (subtype of the FunctionalElement). Functions have a minimumFrequency attribute, a type attribute and multiple FunctionalInterfaces, where each functional data is either a FunctionalOutput

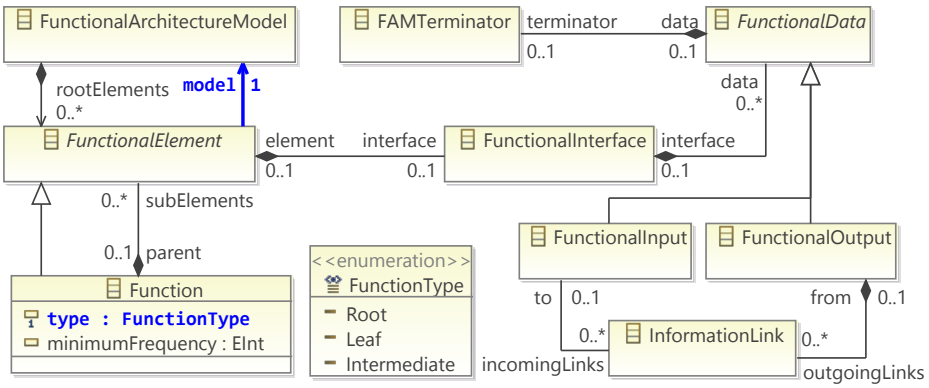


Figure 9.2: Metamodel of the Functional Architecture

(for invoking other functions) or a FunctionalInput (for accepting invocations). An output can be connected to an input through an InformationLink. Finally, if an input or output is not connected to an other Function then it must be terminated in a FAMTerminator.

9.2.3 Derived features

Derived features (DF) are frequent extensions of metamodels to improve navigation by path compression or compute derived attributes. The value of a DF can be computed from other parts of the model as defined by a model query [RHV12; Ocl]. Such queries $\texttt{df}(o, v)$ have two parameters: (i) for derived references o represents the source and v the target object of the reference while (ii) for derived attributes o represents the container object and v is the computed value the attribute.

A derived feature `f` defines the values of the selected feature in the following way: $\forall o, v : \text{feature}(o, v) \Leftrightarrow \texttt{f}(o, v)$. This has to be satisfied for each derived feature in the DSL which is defined by the DF rule, therefore the definition of a valid model is specified as: $M \models MM \wedge DF$. Model query frameworks like Viatra automatically recalculate the value of the derived features in the instance model to satisfy DF [RHV12].

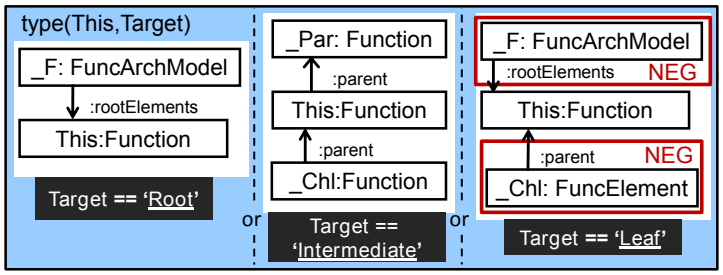
Our sample DSL contains two derived features highlighted in blue in Figure 9.2: a `type` which defines the value of an enum attribute and a `model` which points to the container FAM models.

Example 37. The derived attribute type of Function is defined to take a value from the enumeration literals: Leaf, Root, Intermediate. The pattern defining the type attribute is illustrated on the right side of Figure 9.3. We use both a custom graphical and the textual Viatra notation [Ber+11] to illustrate the queries defined for these derived features. In the graphical notation each rectangle is a variable with a declared type, e.g. the variable `_Par` is a Function, while arrows represent references of the given EReference between the variables, e.g. the function `This` has the function `_Par` as its parent. Negative application conditions (NACs) are illustrated as 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 definitions the `type` query has three `or` pattern bodies each defining the value for the corresponding enum literal of the type attribute:

Root if the container object is directly under the FunctionalArchitectureModel connected by rootElements.

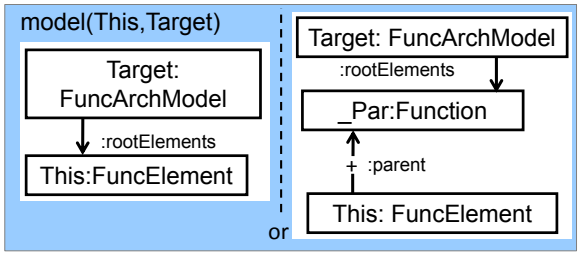
- Leaf if the container object does not have a child along the subElements EReference and it is not a root element (as defined by the corresponding negative application conditions neg).
- Intermediate if the container object has both parent and child functions.




```
@QueryBasedFeature
pattern type(This,Target){
  find rootElement(_Model,This);
  Target==::Root;
} or {
  neg find parent(_Child,This);
  neg find rootElement(_Model,This);
  Target==::Leaf;
} or {
  find parent(This, _Par);
  find parent(_Child, This);
  Target==::Intermediate; }
```

Figure 9.3: Definition of derived attribute type

Example 38. FunctionalElements are also augmented with a derived reference model (highlighted in blue in Figure 9.2) which represents a reference to the container FunctionalArchitctureModel object from any FunctionalElement within the containment hierarchy. The definition of the corresponding graph pattern is visible in Figure 9.4 which calculates the transitive closure of the parent reference between elements This and _Par as denoted by an arrow with a + symbol.




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

Figure 9.4: Definition of derived reference model

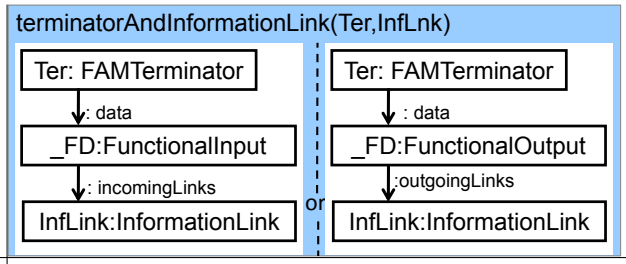
9.2.4 Well-formedness constraints

Structural well-formedness (WF) constraints (aka design rules or consistency rules) complement metamodels with additional restrictions that have to be satisfied by a valid instance model (in our case, functional architecture model). Such constraints can also be defined by query languages such as graph patterns or OCL invariants, in fact, our validation approach supports both of these formalisms. In many practical cases, well-formedness constraints are defined by queries which capture ill-formed model structures that are disallowed to have a match in a valid model. In the presence of a set WF of well-formedness constraints, a model M is called valid if $M \models MM \wedge DF \wedge WF$.

Example 39. In our running example, a WF constraint captures that a FunctionalData object with a FAMterminator cannot be connected to an InformationLink. It is specified by the `terminatorAndInformationLink` query (see Figure 9.5) that has two `or` pattern bodies, one for the FunctionalInputs and one for the FunctionalOutputs with their corresponding incomingLinks and outgoingLinks, respectively.

The same constraint is also captured in OCL, see bottom part of Figure 9.5 for a comparison. Note that graph patterns are normally ill-formedness constraints to capture erroneous situations while OCL invariants capture the valid case, and violations are identified by their context. Another WF constraint specifies that the frequency of a subfunction has to be equal to (or exactly twice or four times as much as) the frequency of its parent function in order to enable communication. This WF constraint is specified in Figure 9.6.

VQL



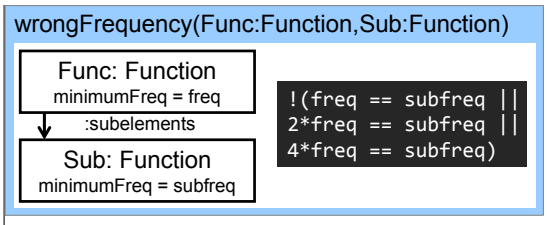

```
@Constraint
pattern terminatorAndInformation
  (Ter,InfLink){
FAMTerminator.data
  .incomingLinks(Ter,InfLink);
} or {
FAMTerminator.data
  .outgoingLinks(Ter,InfLink);}
```

OCL

```
context InformationLink inv terminatorNoLink:
  (self.to <> null implies self.to.terminator = null) and
  (self.from <> null implies self.from.terminator = null)
```

Figure 9.5: Definition of the WF constraints `terminatorAndInformationLink` and `terminatorNoLink`

VQL




```
@Constraint
pattern wrongFrequency(Func,Sub){
  Function.subElements(Func,Sub);
  Function.minimumFrequency(Func,freq);
  Function.minimumFrequency(Sub,subfreq);
  check(!(freq == subfreq ||
        2*freq == subfreq ||
        4*freq == subfreq));
}
```

OCL

```
context Function inv RightFrequency:Function.allInstances()->
  forAll(par,child|(child.parent=par)
  implies ((child.minimumFrequency = par.minimumFrequency) or
           (child.minimumFrequency = par.minimumFrequency * 2) or
           (child.minimumFrequency = par.minimumFrequency * 4)))
```

Figure 9.6: Constraint for the frequency of the subfunctions

9.2.5 Partial snapshots

Multiple PSs will be passed as an input parameter to the validation process, and the solver will try to construct a valid instance model which satisfies each of them.

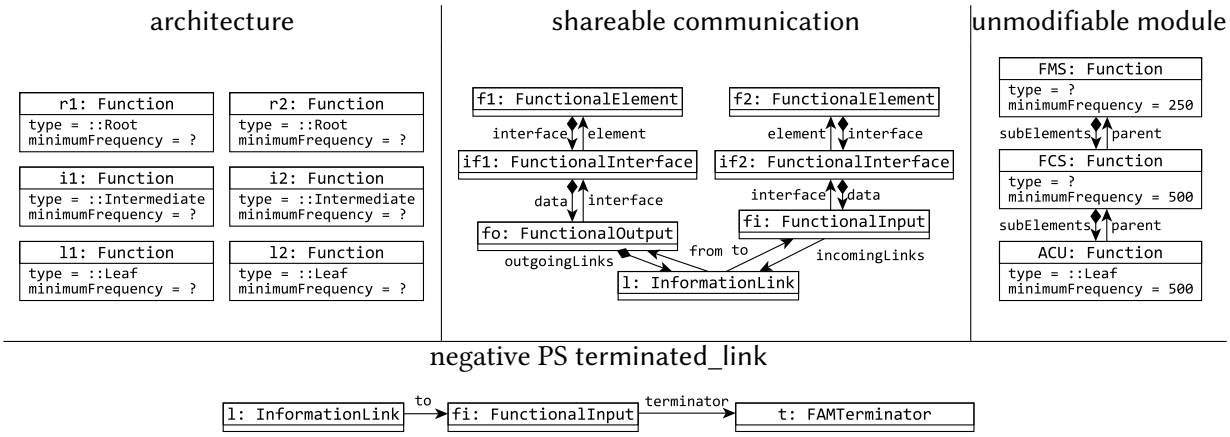


Figure 9.7: Partial snapshots with semantic modifiers

Example 40. Figure 9.7 shows four PSs generalized from instance models by removing certain model elements.

architecture: This PS defines a core structure of an IMA architecture prescribed to contain two of each root, intermediate and leaf elements, but it does not define their exact structure. As type attribute is a derived feature, the instance models that contain this PS must be arranged in an architecture which evaluates to the correct literals.

- communication: This PS contains a communication link from an input FunctionalElement to an output FunctionalElement via FunctionalInterfaces.
- module: This PS defines a module with three functions (FMS: Flight Management System, FCS: Flight Control System, ACU: Avionics Control Unit) arranged into a tree hierarchy via subelement and parent edges.
- terminatedLink: This example shows a FunctionalInput extracted from an invalid model as it contains both a FAMTerminator and an InformationLink.

9.3 Overview of the approach

This section provides a high-level, functional overview of our DSL validation approach using an SMT-solver. It gives the precise definition of the validation challenges for DSLs and describes how these challenges can be addressed by appropriate configuration of the solver.

9.3.1 Functional overview of the approach

The validations are initiated and executed in well-defined *context*, which is treated as a set of axioms for the validation run. This context can be customized during DSL validation by selecting (or de-selecting) certain DSL artifacts from the following list. As a result, the output model M retrieved during DSL validation needs to respect the context.

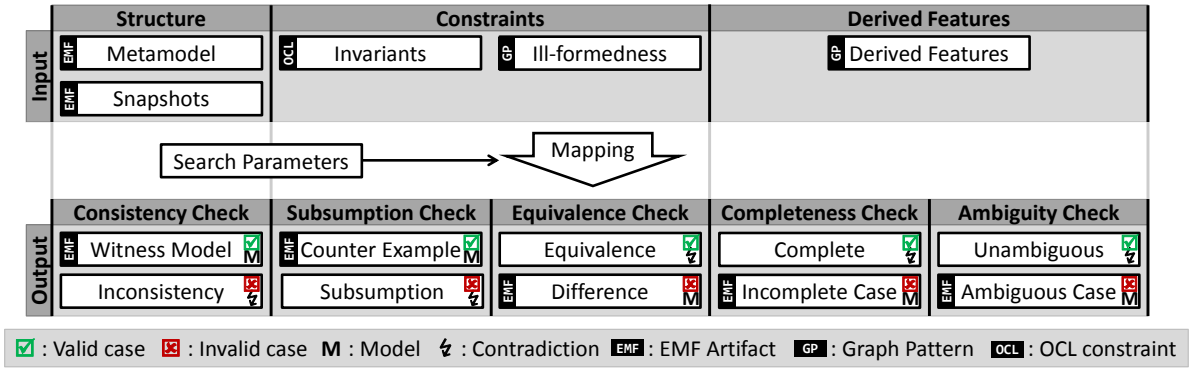


Figure 9.8: Overview of validation tasks

- **Metamodels:** The selected set of domain classes allowed to be instantiated for constructing models: $M \models MM$.
- **Derived Features:** The values of the derived features have to be correctly evaluated with respect to their definition yielding unique and complete results: $M \models DF$.
- **Constraints:** The output model has to satisfy the selected well-formedness constraints: $M \models WF$.
- **Partial Snapshots:** The output model M optionally has to contain partial snapshots: $M \models PS$.
- **Search Parameters:** Additionally, the user may define some reasoning-specific input parameters:
  - **Size:** The number of objects used in the construction of an output model $M = \langle O_M, \mathcal{I}_M \rangle$ can be restricted by a positive integer (defined by $|O_M| = size$). By default, $size = *$, which means that the analysis covers all each possible model regardless of its size.
  - **Approximation level:** Some DSL property (such as the acyclicity of the containment hierarchy) cannot be represented in FOL. The method is customizable with the level of approximation (see Section 3.2.3), which allows to set the limit of approximation level. Higher approximation level will reduce the possibility of false positives.

The constraints serving as the context of DSL validation are summarized as $DSL = MM \wedge DF \wedge WF \wedge PS$, and it defines a possibly infinite (if $size = *$) set of $\mathcal{M}_{DSL} = \{M : M \models DSL, |O_M| = size\}$. If each parameter is set to the default value, the analysis covers the full range of valid instance models (thus the full language is analyzed).

Figure 9.8 shows a more detailed overview of the different DSL validation tasks, their respective input parameters (upper part) and the possible validation outputs (lower part) of our framework.

The input parameters allow to define the DSL validation context (as discussed above), and the selected elements of the DSL context are mapped to FOL in accordance with further reasoning-specific search parameters. We distinguish between five DSL validation tasks (consistency, subsumption, equivalence, completeness and ambiguity checks), which are detailed below. In each case, the validation run terminates with constructing an output model, or revealing a contradiction. The result of the model generation is interpreted for each validation task, the valid and invalid outcomes are highlighted in Figure 9.8. If an output model is retrieved it is presented as a witness model (for consistency and subsumption check), or as a counterexample (e.g. a proof of ambiguity or incompleteness).

9.3.2 Consistency check

Consistency is a property of the whole DSL which means that there is no contradiction in its specification, i.e. there is at least one valid instance model. A consistency check either reveals the conflicting elements (constraints, derived features) of a DSL or proves that the DSL is consistent. Because any statement can be proven from a contradicting set of axioms, an inconsistency invalidates the result of any further DSL checks (like completeness, ambiguity and subsumption and equivalence checks). A consistency check aims to prevent such a situation.

Definition 33 (Consistency) *A DSL context is consistent if it has a valid instance model, i.e.* $\mathcal{M}_{DSL} \neq \emptyset$. *A DSL context is inconsistent if it is not consistent i.e.* $\mathcal{M}_{DSL} = \emptyset$ *(where* $\mathcal{M}_{DSL} = \{M : M \models DSL\}$*).*

Note that there are no further restrictions on the size of M to serve as a proof of consistency, i.e. M might be as simple as a single object. In fact, many SAT-solvers would retrieve such a model by default. Therefore, for practical DSL validation, stricter consistency requirements are necessitated, such as every class and reference in the metamodel have to be instantiated at least once in an output model.

Such consistency criteria can be encoded and checked in our framework using partial snapshots. The use of partial snapshots and flexible model size limit (and further search parameters) make the generation of output model highly customizable. The underlying solver is called with the metamodels, the derived features, the constraints and partial snapshots as input. The output model is interpreted as a witness model of consistency, while a contradiction is a proof of inconsistency. In practice, consistency checks are typically used to (1) identify contradictions in a DSL specification, (2) check if each language element can be instantiated or (3) generate instance models for a given context (e.g. relevant contexts for test cases).

9.3.3 Subsumption check

A complex DSL may contain a large number of independent language properties (well-formedness constraints, derived features, partial snapshots). With a growing number of language properties (i.e. DSL context), a redundant property is difficult to be identified merely by human inspection. While consistency checks reveal a contradicting language specification, it would be advantageous to reveal if a new constraint really imposes further restrictions on valid instances, or it is already covered by the existing DSL specification. Subsumption checks of a language property aims to detect this latter case. A subsumed constraint does not express any additional restriction over the DSL therefore it can be removed without any further consequences.

Definition 34 (Subsumption) *A property* P *is subsumed by a DSL context if* $DSL \models P$. *A* P *property is not subsumed by a DSL context if* $DSL \not\models P$. *A* P *property is independent from a DSL context if* $DSL \not\models P \wedge DSL \not\models \neg P$.

Informally, property P is subsumed by a DSL specification when every model that satisfies the DSL specification will also satisfy this property P (formally, $\forall M : DSL \models M \Rightarrow M \models P$). If property P is not subsumed then there is a valid instance model that satisfies the DSL specification but not the property itself (formally, $\exists M : DSL \models M \wedge M \not\models P$). Finally, the property is independent from a DSL context if it is consistent with it but not subsumed by it.

Given the DSL context as a set of axioms, we carry out traditional theorem proving to decide if property P is derivable from the set of axioms. For this purpose, we aim to prove that adding $\neg P$ as an axiom to *DSL* makes the new specification $\{DSL, \neg P\}$ inconsistent. Therefore, an output model retrieved by the solver is interpreted as a *counterexample* to testify that property P is not subsumed. In order to show the independence of a property P, the axiom set $\{DSL, P\}$ is aimed to be refuted as well, and we require both validation runs to retrieve an output model.

9.3.4 Equivalence check

Language properties can be defined in multiple ways, potentially using different languages and formalisms. A well-formedness constraint or a derived feature can be captured as a graph pattern, as an OCL invariant or as positive or negative partial snapshots. In practical scenarios, a DSL specification in fact uses a mixture of such techniques. Moreover, automated transformations are defined to convert OCL constraints into graph patterns and vice versa. Equivalence checks aim to prove the correctness of conversions between the language properties in a DSL.

Definition 35 (Equivalence) *Properties A and B are equivalent in a context DSL if*

$$DSL \models A \Leftrightarrow DSL \models B.$$

According to the definition, two validation runs are initiated for checking the consistency of sets $\{DSL, \neg A\}$ and $\{DSL, \neg B\}$, respectively. The result of the validation is the proof of the equivalence, or an example to highlight the semantic difference between the two property, which presents an example where one property is satisfied, but the other is not.

9.3.5 Completeness and ambiguity check of DFs

Derived features specified by model queries (defined by graph patterns or OCL constraints) are integral parts of a DSL specification. However, the definition of such DFs is error-prone as the corresponding query has to yield a well-defined result in all situations. Here, we aim to check the completeness and unambiguity of derived features.

Completeness of a derived feature means that the DF satisfies the lower multiplicity constraint of the target structural feature. For example, in case of a reference with 1..? multiplicity, completeness is achieved if the DF evaluates to at least one value for every occurrence of the derived feature. If there is a model where no values can be assigned to an occurrence of the derived feature it is incomplete.

Let $mul^{min}_{MM}(\mathsf{F})$ denote the lower multiplicity and $mul^{max}_{MM}(\mathsf{F})$ denote the upper multiplicity of feature F by appropriate constraints, which is enforced by axioms Mul^{min}_{F} and Mul^{max}_{F} respectively. Furthermore, let $DSL \setminus C$ denote a DSL context where the language constraint C is removed. Completeness is then defined as follows:

Definition 36 (Completeness of Derived Features) *A derived feature* F *is complete in a DSL context if* $DSL \setminus Mul^{min}_{\mathsf{F}} \models Mul^{min}_{\mathsf{F}}$. *Otherwise, DF is incomplete.*

Ambiguity allows the upper limit of the multiplicity to be exceeded. For example, a DF complies with the ?..1 multiplicity if it evaluates to at most one value for every occurrence of the DF. An output model where multiple values can be assigned to some occurrence of the DF means that the DF is ambiguous.

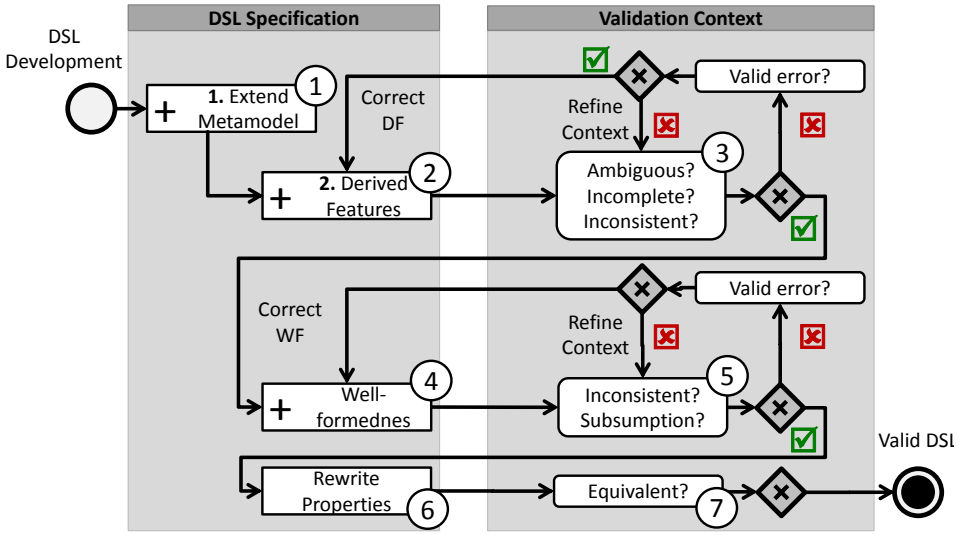


Figure 9.9: DSL validation workflow

Definition 37 (Unambiguity of Derived Features) *A derived feature* F *is unambiguous in a DSL context if* $DSL \setminus Mul_{F}^{max} \models Mul_{F}^{max}$. *Otherwise DF is ambiguous.*

In accordance with the definitions above, the corresponding multiplicity constraints are extracted from the DSL context and their negation is added back and checked for contradiction. The result of the validation task could be either the proof of completeness / unambiguity of the checked DF with respect to the validation context, or a counterexample that, although satisfies the specification of the DF, violates its multiplicity.

9.4 A case study on DSL validation

Complex DSL specifications may contain multiple inconsistencies, and the erroneous language properties are difficult to identify and localize. To assist the developers finding such inconsistencies, we propose an iterative workflow that defines the practical order of addressing and completing the different validation steps. By following this workflow, our framework will reveal the design flaws one by one so with the help of the counterexamples the root cause of the error can be better detected. This iterative approach can be applied in any stage of DSL development (including also on incomplete language specifications) thus specification errors can be detected in an early phase of DSL design. Additionally, the workflow can guide the developer through a complete language validation process.

9.4.1 Overview of DSL validation workflow

The validation workflow is illustrated in Figure 9.9.

1. First, a metamodel is added to the validation context process and checked for consistency.
2. Derived features are iteratively added (extending it with one new DF at a time)
3. The unambiguity and completeness of the DF as well as the consistency of the entire validation context are automatically checked.

4. After checking all DFs, validation continues with the WF constraints, which are added to the validation context iteratively (one-by-one).

5. Our framework inspects whether the current WF constraint causes inconsistency or it is already subsumed by the current validation context.

6. If the validation of the DSL succeeds with all DF and WF constraints included in the context, then the DSL is valid under the assumptions imposed by the search parameters and the partial snapshots. The DSL validation process succeeds.

7. Partial snapshots retrieved in the validation context can be turned into WF constraints of the DSL. In such a case, our framework formally proves that the WF constraint and the PS in the context are formally equivalent, therefore, the corresponding WF is not required to be re-validated.

ER If the validation fails at any step, the language engineer has to correct the DSL artifact based on the counterexample, and continue the validation from the modified element. In case of *false positives* (which are detected by checking the result model whether it truly satisfies the constraints), the parametrization of the search needs to be fine-tuned. If the result is an error but the framework provide a *spurious counterexample*, then the validation context should be extended by the missing constraints.

Starting the validation of derived features prior to WF constraints is based on the observation that each DF eliminates a large set of trivial, non-conforming instance models (which are not valid instances of the DSL). Moreover, adding a single constraint at a time to the validation problem helps identify the location of the problem, because the solver provides only very restricted traceability information. This eases the refinement in case of an erroneous DF or WF is added in the actual step based on the proof provided by the solver.

The rest of this section demonstrates how this DSL validation workflow can be applied on the running example from the avionics domain.

9.4.2 Derived type validation

For illustration purposes, we artificially inject two conceptual flaws into the query defining the derived feature type in the IMA example (depicted in Figure 9.10, and see Figure 9.3 for its correct original definition):

1. The pattern body representing the intermediate case has been removed, which makes the DF incomplete.

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

The validation process is presented in Table 9.1.

- First (**Step 1**) we add the `type` DF to the DSL context and its consistency is successfully validated.

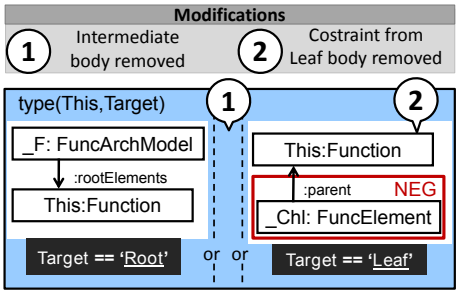


Figure 9.10: Modifications on the `type` pattern

Validation step	Outcome	Action
1) Consistency : `type`	[✓]	
2) Completeness: `type`	[×]→CE1	Acyclicity approximation = 2
3) Completeness: `type`	[×]→CE2	+ body to `type`
4) Completeness: `type`	[✓]	
5) Unambiguity : `type`	[×]→CE3	+ constraint to `type`
6) Unambiguity : `type`	[✓]	+ constraint to `type`
7) Consistency : `model`	[✓]	
8) Completeness: `model`	[×]→CE4	Partial snapshot = PS1
9) Completeness: `model`	[?]	Checked in boundend size
10) Unambiguity : `model`	[✓]	
11) Consistency : `termAndInfLink`	[✓]	
12) Consistency : `InfAndTerm`	[✓]	
13) Subsumability: `InfAndTerm`	[×]	Remove `InfAndTerm`
14) Equivalence : `termAndInf`⇔`termNoLink`	[✓]	
15) Equivalence : `termNoLink`⇔`terminatedLink`	[×]→CE5	

Table 9.1: Example DSL validation run

- Then (**Step 2**), the completeness of the derived feature `type` is checked resulting in a failure illustrated by the Counterexample 1 showing three functions without type to form a cycle in the containment hierarchy. This counter example is visualized in Figure 9.11 where the invalid elements are highlighted in red, and the containment references are represented with black diamonds. Note that almost all properties of the instance model are correct, only the containment hierarchy is violated (along the $n1$-$n3$-$n4$ cycle). This is a false positive case since the acyclicity of the containment hierarchy can only be approximated in first order logic. In our framework, this problem can be easily solved by simply raising the level of approximation for transitive acyclicity .

- In **Step 3**, our tool shows a real counterexample (middle part of Figure 9.11) where the intermediate function $n4$ does not have type attribute. This is fixed by adding (back) the second pattern body with the Intermediate definition to the `type` pattern.

- After correcting it, the validation is successfully executed in **Step 4**.

- Then the ambiguity of attribute type is checked (**Step 5**), which fails again with a single function that is both a Leaf and a Root. This counterexample is also visible in the right part of Figure 9.11.

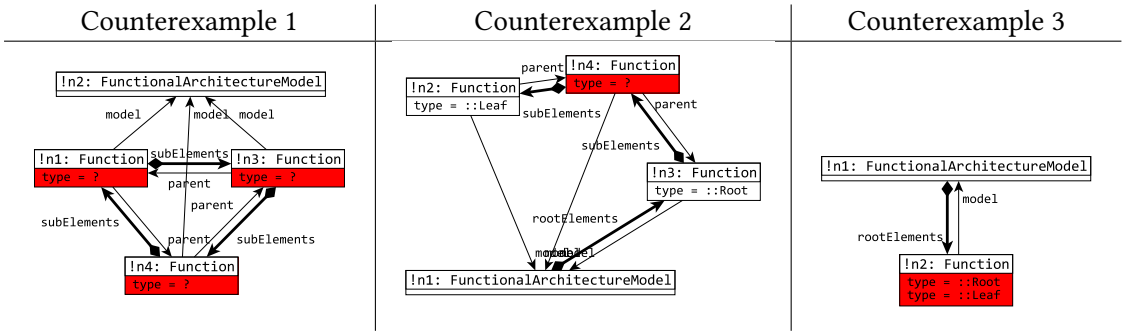


Figure 9.11: Counterexamples of the `type` validation

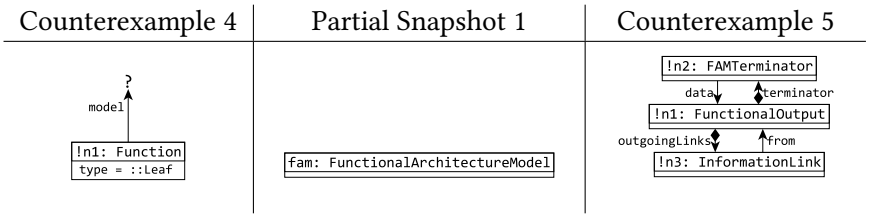


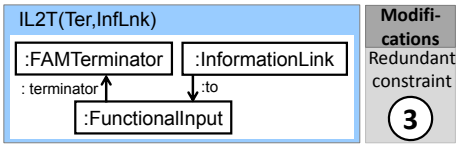
Figure 9.12: Counterexample and partial snapshot for validating `model` DF, and counterexample 5 for failed equivalence check

- This issue is fixed by adding the missing NAC condition on the rootElements to the third pattern body of `type` in **Step 6**.

9.4.3 Derived reference validation

Now the validation process (see Table 9.1) of DF `model` is presented that defines a reference to the container FunctionalArchitctureModel from a FunctionalElement.

- **Step 7** adds the `model` DF to the specification, and the consistency check is executed successfully.
- Then in **Step 8**, completeness validation fails as pointed out in Counterexample 4 in Figure 9.12 since a model with a single Function element does not have anything to refer to with the model link. This result represents a spurious counterexample, because Functions are only used in the context of a FunctionalArchitectureModel. For this purpose, a partial snapshot is defined with a FunctionalArchitectureModel object to prune the search space and avoid such counterexamples (Figure 9.12).
- 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.
- Finally in **Step 10**, the unambiguity of the `model` DF is validated without a problem.




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

Figure 9.13: Definition of the redundant `InformationAndTermintator` pattern

9.4.4 Validation of well-formedness constraints

To demonstrate the subsumption check, another WF constraint is added to the DSL specification expressed by the `InformationAndTermintator` query (Figure 9.13, 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 (see Figure 9.5) in that it uses the inverse edges, thus it is redundant.

The validation process of WF constraints is illustrated in Table 9.1.

- At first, the consistency validation of the WF constraint `terminatorAndInformationLink` (**Step 11**) is executed with a success.
- Then the redundant `InformationAndTermintator` is added which remains consistent (**Step 12**).
- Finally, the last constraint is checked for subsumption (**Step 13**) and found positive. Thus it is already covered by the DSL specification, therefore it can be deleted from the set of WF constraints.

9.4.5 Equivalence check

We selected two use cases to demonstrate the equivalence check in DSL validation in Table 9.1.

- First (**Step 14**), we used our framework to show the equivalence of the WF constraint defined by the `terminatorAndInformation` graph pattern and `terminatorNoLink` OCL invariant (see Figure 9.5). There was no valid instance model found, that violates only one of these two constraints, so the equivalence of the two different representations is successfully proved, thus they can replace each other in the DSL.
- Then in **Step 15**, we tried to check the equivalence of the `terminatorNoLink` (see Figure 9.5) OCL invariant and terminated_link negative PS (see Figure 9.7).

  Our framework returned with the Counterexample 5 (see in Figure 9.12), which highlights the semantic difference between the two constraints. The PS cannot be matched on this counterexample and since it is a negative PS, the constraint is not violated. On the other hand, the second part of the OCL constraint is violated, because there is a FAMTerminator connected to an InformationLink through a FunctionalOutput. This counterexample proves the inequality of this DSL property.

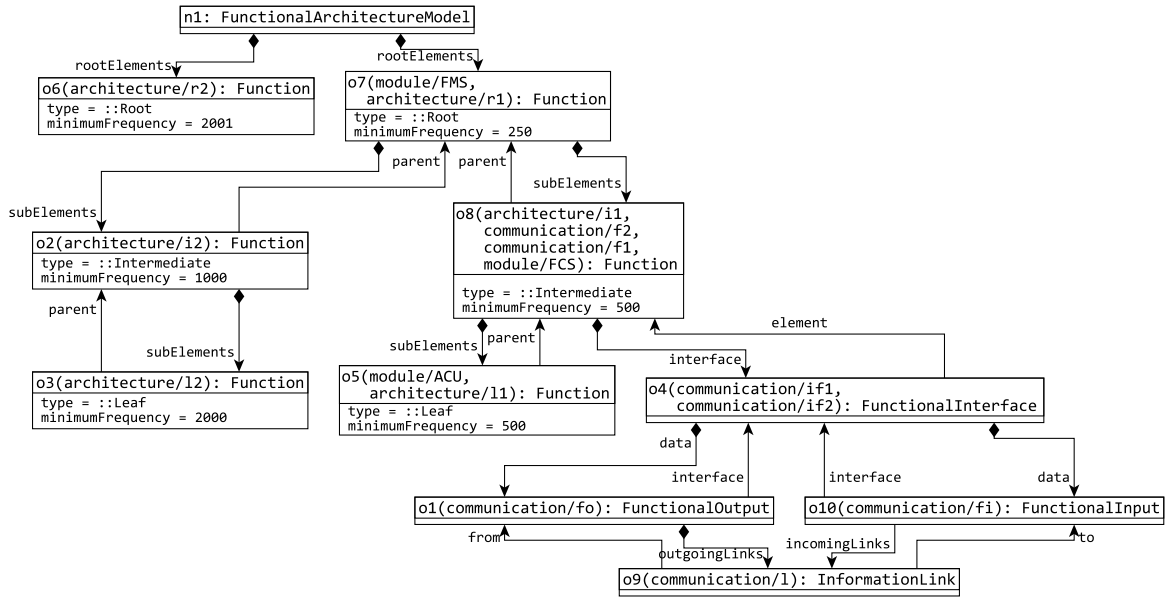


Figure 9.14: Screenshot of a valid model satisfying all partial snapshots of Figure 9.7

9.4.6 Model generation for partial snapshots

Our framework can also be used for generating instance models satisfying a DSL specification under certain assumptions (partial snapshots). This is, in fact, very useful in test generation [Mic+12; Abd+18] or quick fix generation purposes. Below we only briefly demonstrate how to derive a valid model of minimal size (see Figure 9.14) which contains all PSs from Figure 9.7.

- Partial snapshot architecture is satisfied in the output model by objects $o2$, $o3$, $o5$, $o6$, $o7$, $o8$.
- Partial snapshot shareable communication is satisfied along objects $o1$, $o4$, $o8$, $o9$, $o10$, (and their corresponding links). Note that Function o8 and FunctionalInterface o4 is shared for the source and target end of the InformationLink.
- Partial snapshot unmodifiable module can be transformed to objects $o5$, $o7$, $o8$.

Our example demonstrates that a single object in the output model can satisfy multiple roles in different PSs. Furthermore, the same object can be shared when matching shareable PSs.

9.5 Runtime measurements

In order to assess the performance of our approach, we carried out initial experimental evaluation.

9.5.1 Measurement environment

The execution of our validation framework consists of four phases:

1. the *transformation of the validation task* (DSL2FOL), which is proportional to the size of models and the number of constraints;

2. the *execution of the reasoner tool* (FOL2FOL), which can be complex and time consuming
3. the *resolution of the output model* (FOL2PS), which is proportional to the size of the output FOL model
4. the *visualization of the output model* (PS2DSL), which is proportional to the size of output partial snapshot

We concluded that the runtime of Steps 1, 3 and 4 is predictable and negligible compared to the execution of the reasoning step, which is, in fact, difficult to predict. Therefore, below we restrict our runtime measurements to assess the performance of the reasoning step for various validation tasks on the avionics DSL presented in the paper (see Table 9.2 for an overview of properties used for our experiments).

Abbr.	Property	Defined in
Freq-GP	wrongFrequency	Figure 9.6
Freq-OCL	RightFrequency	Figure 9.6
T&I-GP	terminatorAndInformation	Figure 9.5
T&I-OCL	terminatorNoLink	Figure 9.5
Half-GP	InformationandTermintator	Figure 9.13
Half-PS	negative terminated_link	Figure 9.7

Table 9.2: Properties in experimental validation

The tasks serving as test cases are marked with the expected output, which can be positive or negative. The analysis can prove the selected property, check the absence of violations within a bounded context, or result in a timeout. The marking used in this section are illustrated in Table 9.3.

[✓]	Positive result is expected
[×]	Negative result is expected
[?]	Checked in a bounded context, but not proven
-	Timeout

Table 9.3: Measurement outcomes

Each validation task was executed on the DSL three times for both the Z3 SMT solver and the Alloy Analyzer (with SAT4j-solver), then the median of the execution times was calculated. The measures are executed with a 5 minute timeout on an average personal computer[1]. Execution times are presented in seconds.

9.5.2 Evaluation of language-level validation

Consistency Analysis. First, consistency analysis of the full DSL (without the use of partial snapshots) is executed for arbitrary model size ($|O| = *$), then it is repeated for exactly 10 and 100 number of objects. The results are presented in Table 9.4. As the results show, consistency analysis is easily solved with each reasoning approach for small models. The Alloy Analyzer is unable to solve large models (with $\approx$ 100 elements), while the SMT-solver easily solves consistency check for larger size by generating highly symmetric models.

[1]CPU: Intel Core-i5-m310M, MEM: 16GB, OS: Windows 8.1 Pro, Reasoners: Alloy Analyzer 4.2 and Z3 4.3.0.

	$\lvert M\rvert$ =		
	* [✓]	10 [✓]	100 [✓]
Z3, int=$\mathbb{Z}$	0.02	1.85	0.1
Alloy, int=undef	0.06	0.26	-

Table 9.4: Consistency check measurements

	`type`		`type` w. error		`model`	
	Comp [✓]	Unamb [✓]	Comp [×]	Unamb [×]	Comp [×]	Unamb [✓]
Z3, Complete	0.27	0.03	0.70	0.08	0.02	0.01
Alloy, $\lvert M\rvert \leq 10$, int=$[-64; 63]$	20.65 [?]	15.23 [?]	24.75	23.54	29.06	30.13 [?]

Table 9.5: Derived feature validation measurements

	DSL $\models$ Freq		DSL $\models$ T&I		DSL $\models$ Half	
	GP [×]	OCL [×]	GP [×]	OCL [×]	GP [✓]	PS [✓]
Z3, Complete	0.46	1.50	0.85	1.50	0.01	0.02
Alloy, $\lvert M\rvert \leq 10$, int=$[-64; 63]$	0.27	0.18	33.26	32.85	22.58 [?]	24.16 [?]

Table 9.6: Subsumption check measurements of DSL constraints

These measurements indicate that consistency analysis on the language level (i.e. without PSs) is an easy DSL validation task as the output models retrieved as consistency proofs are trivial in most practical cases (e.g. consisting of a single model element).

Derived Features. Next, we checked completeness and unambiguity for derived features `type` and `model`, and the measurement results are summarized in Table 9.5. The complete validation problem was given to the SMT solver, while we restricted the analysis to at most 10 model elements and a [-64; 63] interval for integers in case of Alloy. This is an underapproximation due to the limited expressive power of SAT problems, therefore the UNSAT result from the Alloy SAT-solver can not be used as a proof.

Since completeness and unambiguity still leads to a real theorem proving problem, the SMT solver excels in these cases (both for proving correctness without assumptions and retrieving counterexamples). Furthermore, the lack of counter-examples in case of SAT solvers is still not a general proof due to the bounded context.

Subsumption and equivalence checks

We also carried out subsumption checks to decide if a certain constraint (captured in GP, OCL or PS formalism in the different cases) is already covered by the DSL specification. Measurement results are summarized in Table 9.6. The SMT solver is particularly good for proving subsumption (3rd case) but it also has predictable runtime for the negative cases (1st and 2nd). It is interesting to note that FOL formulae derived from OCL constraints required more time, which might indicate some inefficiencies in our OCL transformation.

We also aimed to prove equivalence for certain constraints captured in different formalisms (graph patterns vs. OCL invariants; partial snapshots vs. graph patterns or OCL invariants). Measurement results are listed in Table 9.8. In this validation setup, Alloy also performed well - but the SMT solver

	Freq [✓] GP ⇔ OCL	T&I [✓] GP ⇔ OCL	Half [✓] GP ⇔ PS	T&I - OCL ⇔ Half - PS [×]
Z3, Complete	0.04	0.03	0.02	1.14
Alloy, $\|M\| \leq 10$, int=$[-64; 63]$	-	1.40 [?]	1.11 [?]	0.40

Table 9.7: Equivalence check measurements of DSL constraints

still had a predictable runtime.

Our experiments to carry out various language-level validation tasks clearly indicates that the Z3 SMT solver outperforms the SAT-based Alloy reasoner tools, which coincides with our a priori expectations.

9.5.3 Model generation evaluation

In DSL validation properties are decided by detecting contradicting requirements or providing small examples. Valid instance models of increasing size are generated to measure its efficiency.

By customizing the validation context, our approach generates various instance models with designated properties. To avoid empty and symmetric models, the generation processes are executed with three PSs as input: architecture, shareable communication and unmodifiable module which are defined in Figure 9.7, so the results are similar to the model in Figure 9.14. Models are generated with 10 to 20 objects, where the smallest model (with 10 objects) is too small to contain each PS, but with 11 or more objects the problem is satisfiable. The results are presented in Table 9.8.

In the first series of measurements (with arbitrary integers), the Z3 solver generated models up to 16 elements, with increasing execution times, while the Alloy Analyzer was unable to initialize. The reason of the Alloy failure is that the unmodifiable module contains large integers (around 500), which is difficult for SAT-solvers.

In the second series of measurements, the minimalFrequency values are reduced to 2 and 4 (from 250 and 500), and the range of the integers is reduced to the $[-64; 63]$ interval for both solvers. With this integer range, the two solvers produced models after about the same runtime, but the Alloy Analyzer ran out of memory for models with over 14 elements. Note that using a integer limit decreased the efficiency of model generation for the Z3 solver.

When the interval of the integers is further reduced, the Alloy Analyzer clearly outperforms the Z3 solver. A third series of measurement were executed, where the integers are removed from the DSL, only objects and enums are present. In this case the Alloy Analyzer has close to zero runtime, and even models with 80 objects can be generated within 145 seconds. The Z3 solver also generates models without integers with higher efficiency.

Our measurements indicate that SMT-solvers are strong in proving language-level properties and handling integer attributes, while SAT-solvers can generate larger instance models as witness or counter-example. Part of our future work will be directed to combine the strengths of the two approaches.

9.6 Related Work

In Model-Driven Engineering language analysis and validation has become a very hot topic, especially in the safety critical design and development domain (e.g., DO-178C [Do1] for the civil avionics

	$\|M\|$ =										
	10[×]	11[✓]	12[✓]	13[✓]	14[✓]	15[✓]	16[✓]	17[✓]	18[✓]	19[✓]	20[✓]
Z3, int=$\mathbb{Z}$	8.29	11.16	22.13	26.31	194.93	236.15	363.39	-	-	-	-
Alloy, int≈500	-	-	-	-	-	-	-	-	-	-	-
Z3, int=$[-64; 63]$	24.31	41.81	31.03	83.24	125.32	235.29	416.71	357.33	-	-	-
Alloy, int=$[-64; 63]$	29.85	33.46	38.00	68.54	-	-	-	-	-	-	-
Z3, int=$\emptyset$	20.80	8.81	11.87	18.2	23.43	38.24	41.71	51.02	57.01	77.29	94.91
Alloy, int=$\emptyset$	0.33	0.22	0.24	0.21	0.20	0.20	0.21	0.46	0.32	0.31	0.32

Table 9.8: Model generation with increasing size

domain) that accepts formal verification as certification artifacts. In the current section we provide an insight to similar approaches in a broader research scope.

Metamodeling framework validation- Formula [JLB11] is a tool for validating DSLs, which takes a metamodel, a partial instance model and a set of constraints and rewriting rules as input, and it aims to extend the partial instance model so that the dedicated state can be reached from it by applying the rewriting rules. As a technological difference, our tool is compliant with standard Eclipse based technologies, while Formula uses its own modeling language. Most validation tasks identified in the paper are not yet supported in Formula, which is specialized in reachability and consistency checks. The Formula tool also uses the Z3 SMT-solver as underlying engine.

Clafer [BCW10] is a lightweight structural modeling language used for feature modeling with minimalistic syntax and rich semantics equivalent to first-order relational logic. The specification language supports structural modeling, constraints (well-formedness constraints are written in their own language, which is said to be equivalent to FOL) and also partial configurations. Partial configurations are like partial snapshots in our approach: instance models with undefined attributes and features that can be the basis of model completion. DSL specification given in Clafer are validated using the Clafer Tools [Ant+13] that supports various tasks for domain engineering, like consistency checking and instance model generation based on backend reasoners like Alloy or Choco [Cho; Lia12]. The provided solution for model completion (ClaferIG - Instance Generator) for structural requirements and another solution for model optimization (ClaferMOO - Multi-Objective Optimizer) [Ola+12] for attributed models to find a set of Pareto-optimal model instances based on given a set of optimization objectives.

The main difference between the Clafer and our approach is that we support EMF as our metamodeling language compared to the Clafer specification language, which is only supported by their own framework. However, one interesting feature of the Clafer tooling is that it uses two different tools for the structural and attribute rule validation therefore it might scale better in case of complex DSLs and thus is one of our future goal to adapt such approach.

Validation of OCL enriched metamodels. There are several approaches and tools aiming to validate models enriched with OCL constraints [GBR05] relying upon different logic formalisms such as constraint logic programming [CCR07; CCR08; BC12], SAT-based model finders (like Alloy) [SAB09; Ana+10; Büt+12; KHG11; Soe+10], first-order logic [BKS02], constructive query containment [Que+12], higher-order logic [BW07; GRR09], or rewriting logics [CE08]. Some of these approaches (like e.g. [CCR08; Büt+12; KHG11; SAB09]) offer bounded validation (where the search space needs to be restricted explicitly) in order to execute the validation and thus results can only be considered within the given scope, others (like [BW07; BKS02]) allow unbounded verification (which normally

results in increased level of interaction and decidability issues).

One of the most relevant mapping from a subset of OCL into first order logic (OCL2FOL) is presented in [CED09], 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. In [DC13] the authors present the extension of their previous mapping, called OCL2FOL$^+$, which deals with a four-valued logic defined in the OCL standard that is not yet supported by our approach. These works support consistency checking between a set of OCL invariants, while our approach aims to deal with the whole specification and is able to detect inconsistencies between the different DSL artefacts.

In [CGR13] the transformation is done in a reverse direction, but includes similar approach for mapping the Alloy language elements to UML and OCL. The main difference between our work and this solution is that the engineer should use Alloy to formalize the model and do the V&V tasks, while we allow the usage of pure EMF and OCL. As a key difference is that our work covers (1) multiple inheritance in metamodels and (2) handling of float arithmetics while the rest of OCL coverage is similar. Their work can better exploit some higher order features in Alloy to capture OCL constructs like size(), min() - where our approach can only provide an approximation.

In [KG12a] a mapping from UML and OCL to Relational Logic Formulae is presented. As a difference our paper covers (1) multiple inheritance in metamodels and (2) the new transitive closure construct in OCL by approximation (3) handling of float/double arithmetics. On the other hand, their approach covers equality between strings and other collections like Bags. Some technical details of their mapping relies upon advanced language features available in Alloy, which we do not use as being independent of target back-end solvers.

The [CCR14] presents a mapping from UML models enriched with OCL formulae to CSP, but the main goal is the consistency check and provides formal verification for the models. Additionally their solution provides similar consistency checks and formal verifications (like subsumption, equivalence and advanced consistency checks). As a difference, our approach handles multiple inheritance, and approximate transitive closure, but they support higher order OCL constructs like `size()`, `min()` or `max()`.

Additionally, we proposed a translation [Ber14] of a subset of OCL to graph patterns to provide a effective model validation on the instance level as opposed to the current work, which aims DSL specification validation. There are also mappings from programming languages extended with OCL constraints to reasoners such as Testera [MK01] (from Java to Alloy) and Pex [Res] (from C# to Z3).

Analysis of model and graph transformations. SMT solvers have also been used to verify declarative ATL transformations [BEC12] allowing the use of an efficiently analyzable fragment of OCL [CED09]. The main advantage of using SMT solvers is that it is refutationally complete for quantified formulae 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 counterexamples (model finding) as well as for proving theorems. In case of using approximations for rich query features, our approach converges to bounded verification techniques.

Graph constraints captured as a subset of graph transformation rules are used in [Win+08] 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 [Cab+10] 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 [Cab+10; LBA10; BEC12], as they built upon the semantics of source and target languages

to prove or refute properties of the model transformation. However, the validation tasks identified in the paper are different from the verification challenges of model transformations.

Model extensions using partial models. The idea of using *partial models*, which are extended to valid models during verification also appears in [Sen+12; JLB11; KG12b]. These initial hints are provided manually to the verification process, while in our approach, these models are assembled from a previous (failed) verification run by adding partial snapshots of the spurious counterexamples or increase the level of approximation. [KHG11] presents an approach for the completion of partial snapshots where OCL constraints have to be satisfied. Instead of creating new PS notation the structure is defined by a concrete model and the relaxed properties are specified by dedicated invariants and queries.

Partial models also share certain similarity with uncertain models, which offer a rich specification language [FSC12a] amenable to analysis by the Alloy Analyzer [SFC12]. Uncertain models provide a richer language for partial snapshots for a different purpose: to document semantic variation points generically for instance models. Different potential system models are then synthesized by Alloy accordingly as design decisions. However, their formalism does not support the instantiation of abstract classes, while semantic modifiers are defined individually for model elements (and not for snapshots).

However, any approximations are only used in [JSB12] 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. The PSs are constructed from fully specified instance models in a similar way. However, we additionally propose semantic modifiers which simplifies the specification of complex partial snapshots. On the technological level, our approach handles standard EMF models.

9.7 Conclusion

In this chapter, we presented a validation technique for domain-specific language specifications by a transforming to a first-order logic formulae. The main added value of our approach is to cover rich DSL constructs such as derived features and well-formedness constraints captured in declarative languages such as graph patterns and OCL invariants. We identified several validation tasks (such as consistency, completeness, unambiguity, subsumption and equivalence checks) which are relevant in a DSL validation context. We also proposed a workflow to systematically address these validation tasks for a DSL. We also enhanced this context with partial snapshots to capture further (instance-level) assumptions on valid models. Our mapping tries to transform language features into a decidable fragment of first-order logic (called effectively propositional logic), and to handle language features which cannot be represented in FOL, we proposed powerful approximations.

Chapter 10

Summary of the Research Results

In this chapter I summarize the results of the thesis by formally stating my novel scientific contributions. A previous version of this contribution structure was presented in [C4].

10.1 A graph solver for model generation

My first group of contributions deals with the background logic solvers that required for model generators to address **Research Question 4**.

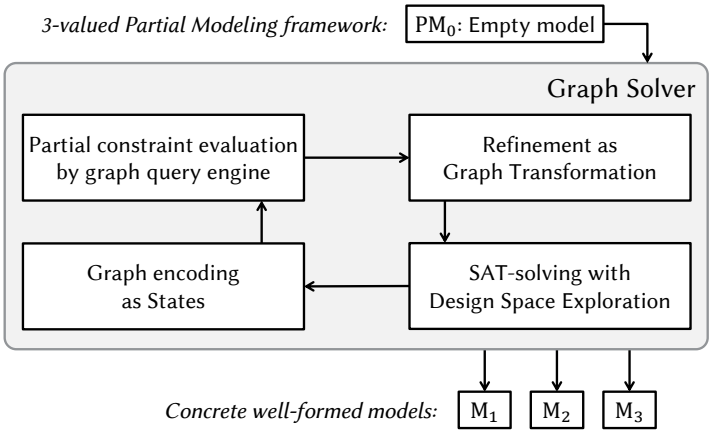


Figure 10.1: Model refinement strategy

Approach As a generalized formalism for all model generation tasks I used *first order relational logic*, which is commonly used as a background theorem for model queries [VB07] and other model generation approaches [TJ07; KJS11]. As an extension, I introduced an extended relational logic with *3-valued logic*, so it is able to represent abstract and partial (unfinished) models as graphs. Based on the framework of partial models I constructed a *novel automated solver technique* (outlined in Figure 10.1) to efficiently generate models by exploiting and innovatively combining a multitude of advanced graph-based techniques:

1. Incremental graph query evaluation of the VIATRA engine [Ujh+15] is used to efficiently *evaluate violations of constraints over partial models* during the model generation process.
2. We formulate model generation as a *refinement of partial models* where models are gradually refined and concretized during exploration with *partial model refinement rules* formalized as *graph transformation rules*.
3. We exploit rule-based design space exploration [HHV15] to *drive the generation process directly over graph shapes* as *decision* and *unit propagation* steps by following core SAT-solving techniques [DLL62].
4. We integrate *shape analysis as state encoding* [RD06; Ren04; RSW04] for graphs to efficiently detect if two partial models should be treated as equivalent during exploration.

Contribution group 1. I proposed a novel logic solver that operates directly on graph models. I integrated existing SAT [Jac02; TJ07; LBP10; ES03] and SMT [DMB08] solvers to the framework.

1.1. *Partial Modeling with 3-Valued Logic.*
I introduced a partial modeling formalism that uses 3-valued logic with interpreted equivalence and existence symbols. I showed that the new formalism is able to represent other popular partial modeling techniques (MAVO and TVLA) [C5][C7].
1.2. *Evaluation of Graph Patterns on Partial Models.*
I presented a graph predicate rewriting technique to enable the evaluation of under- and over-approximation of predicates directly over the representation of a partial model [C5] by graph query engines. I applied the technique to derive additional validation constraints to highlight unrepairable inconsistencies in modeling environments. [J1].
1.3. *Graph Solver approach.*
I created an efficient logic solver that generates consistent graph models for domain-specific languages. The approach uses (i) partial (graph) models as states and (ii) partial model refinement decision and unit propagation rules. The generation is is controlled by (iii) a design-space exploration engine which (iv) continuously evaluates the approximations of well-formedness constraints on partial solutions [C6].
1.4. *Implementation: VIATRA Solver framework.*
I developed an open-source implementation of the graph solver [Vs], which also integrates existing logic solvers like Z3 or Alloy. The tool is available both as an integrated modeling tool and as a standalone application [C8].
1.5. *Experimental Evaluation.*
I carried out experimental scalability evaluation of the graph solver in three case studies of industrial DSLs.

Added value The proposed approach is implemented in a framework [Vs][C8] that is able to solve model generation problems with three underlying solvers, including the novel graph generation technique which operates directly on graph models. The proposed partial modeling technique introduced a sophisticated encoding technique, which enabled the efficient evaluation of constraints on abstract, partial solutions (**Ch1: Encoding** and **Ch2: Abstraction**). In [C6] we showed that the graph solver is able to synthesize consistent graph models with over 500-6000 objects with similar consistency guarantees as other solvers (thus significantly outperforming them in **Ch3: Scalability**). The scalability

of our solver is 1-2 orders of magnitude better than existing mapping based approaches using Alloy [TJ07] with state-of-the-art SAT-solver in the background (which scaled only up to models with 80 objects). In [C11] we showed that the generated models are also more diverse (**Ch4: Diversity**).

Additional applications and related contributions Our research line on model generators is preceded by the development of several performance benchmarks for modeling and model management tools [VSV05; Ber+08] culminating in major open benchmarks like the Train Benchmark [Szá+17][C18] and the CPS Benchmark [Cps] developed by current and former members of the research group. Those benchmarks required the automated creation of a set of graph models with increasing size in order to compare the performance of modeling and query tools. However, those models are constructed with manually defined generation rules, which (i) makes them domain-specific, and (ii) and provide no completeness or coverage guarantees needed for testing.

Design-space exploration (DSE) is a related technique for creating model candidates which are optimal with respect to given objective functions using a set of transformation rules. As the main difference, VIATRA Solver uses generic logic reasoning rules instead of custom domain-specific operations, thus it could generate the complete set of valid model (within the given scope). VIATRA-DSE is a rule-based design space exploration technique for (graph) models [HHV15; Abd+14], which is the contribution of Ákos Horváth, Ábel Hegedüs and András Szabolcs Nagy. In the proposed graph solver technique [C6] I use VIATRA-DSE as an efficient execution engine for reasoning rules, where András Szabolcs Nagy developed custom exploration strategies for reasoning [C6]. Finally, in her master thesis under my supervision Alexandra Sólyom used VIATRA-DSE using the prototype of decision rules [Sol16].

In [C7], we extended 3-valued partial models [C5] as a background theory for representing inconsistent and ambiguous views models. The original theoretical background is my contribution, and the view modeling approach is a separate contribution from Kristóf Marussy.

10.2 Language-level validation for domain-specific languages

The second group of contributions focuses on the language level validation of domain-specific languages to address **Research Question 1**.

Approach I created a novel approach to analyze the language specification of *modeling tools* by mapping them into first order logic (FOL) formulae that can be processed by advanced *reasoners* such as *SMT solvers* (Z3) or *SAT solvers* (Alloy, see Figure 10.2). The outcome of a reasoning problem is either satisfiable or unsatisfiable. If the problem is satisfiable, the solver constructs an output (or completed) model (which is interpreted as a witness or a counterexample depending on the validation task), while an unsatisfiable result means a contradiction. Because certain validation tasks are undecidable in FOL it is also possible that validation terminates with an unknown answer or a timeout.

We carried out a wide range of validation tasks by automated theorem proving based on this formalization to prove different properties of a DSL such as consistency, subsumability, completeness and unambiguity. To highlight such design flaws directly in modeling environments, analysis results of all validation problems are back-annotated in the form of regular instance models. Linking the independent reasoning tool to the modeling tool allows the DSL developer to make mathematically precise deductions over the developed languages.

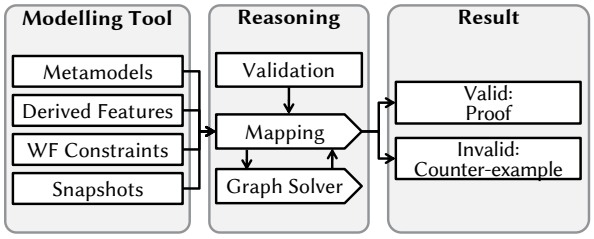


Figure 10.2: Functional overview of the approach

Contribution group 2. I proposed formal analysis techniques for the language-level validation of domain-specific languages by mapping them to formal logic specifications.

2.1. *Mapping of graph patterns to effectively propositional logic.*
I introduced a technique to transform WF and DF rules captured by graph patterns to a decidable fragment of FOL [PMB08] by using over- and underapproximation techniques [C9].
2.2. *Identification of context dependent DSL validation criteria.*
I defined completeness and unambiguity properties of derived features, subsumption and equivalence relations of well-formedness constraints, derived features and instance models. These properties can be checked on the full DSL, or on a specific fragment of it [J2].
2.3. *Uniform validation of DSL specification.*
I introduced a technique that uniformly translates DLS elements to FOL to analyze the consistency of the whole DSL specification, which includes metamodels, instance models, well-formedness (OCL or graph pattern) and derived features. I proposed a validation process for the DSL which systematically checks the language properties, and highlights inconsistencies by deriving representative counterexamples which violate the target properties [C9][J2].
2.4. *Application: Validation of an avionics DSL.*
I carried out the validation of a functional architecture modeling language of avionics systems developed in Trans-IMA project [Hor+14].

Added value The main added value of the approach is to cover rich DSL constructs such as derived features and well-formedness constraints captured in declarative languages such as graph patterns and OCL invariants (**Ch1: Encoding**). While other approaches use bounded verification or simply ignore unsupported features, I proposed approximations to transform language features into a decidable fragment of first-order logic (called effectively propositional logic [PMB08; GM09]), and to handle language features which cannot be represented in FOL. Therefore, the correctness of a DSL can be proved using our method, while others only can detect errors (**Ch2: Abstraction**).

Our approach is supported by a prototype tool integrated into Eclipse, which takes EMF metamodels, instance models and VIATRA graph patterns input to carry out DSL validation. As a technological difference, our tool is compliant with standard Eclipse-based technologies, while Formula and Alloy use their own modeling language. When an output model is derived as a witness or counterexample, this model is back-annotated to the DSL tool itself so that language engineers could observe the source of the problem in their custom language without the need for theorem proving skills.

Related contributions The case study for language level validation was originally developed in the Trans-IMA project [Hor+14] where Ákos Horváth was the technical lead. The mapping technique

of OCL constraint to first order logic [J2] is developed by Ágnes Barta (included to Section A.1 for the sake of completeness). Additional underlying theorem provers [RV99; KV13] are being integrated by Aren Babikian as part of his research.

10.3 Iterative model generation techniques for modeling tools

In my third group of contributions, I developed automated test generation (to address **Research Question 2**) and view synchronization (to address **Research Question 3**) techniques using underlying model generators.

Approach Test generation and view synchronization necessitate the synthesis of nontrivial instance models for complex DSL specifications, which is rarely feasible with a single direct call to the underlying solver. In my thesis, I proposed an iterative process for generating valid instance models by calling existing model generators *in multiple steps* (as seen in Figure 10.3), and using various abstractions and approximations of the previous solution to improve the overall scalability the approach or quality of the models. In particular, I used three iterative techniques:

- **Positive feedback:** instance models can be incrementally generated in multiple steps as a sequence of extending partial models $M_1, \ldots, M_n$, where each step is an independent call to the underlying solver. The main idea behind this approach is that the solver can be guided by smaller logic problems, where only the newly created elements have to be added (marked by Δ), thus increasing their scalability and performance.
- **Change feedback:** in view model synchronization, a sequence of historical source model states $M_1, \ldots, M_n$ are kept consistent with a sequence of view model states $V_1, \ldots, V_n$. As each change Δ_V from V_i to V_{i+1} affects only a small fragment of the view model, most M_i remain unaffected in M_{i+1}. Therefore, using an iterative generation technique, only the changing part of the source model (Δ) needs to be reconstructed by the solver, which results in significantly smaller logic problems.
- **Shape feedback:** best practices of testing (such as equivalence partitioning [Rei97]) recommend the synthesis of a diverse set of graph models $M_1, \ldots, M_n$, where any pairs of models are structurally different from each other to achieve high coverage or diverse solution space. By extracting the shape [Ren04; RD06] of previous models, an iterative model generation technique is obtained to automatically synthesize a diverse set of models (large Δ between models) by enforcing different graph shape for each new model.

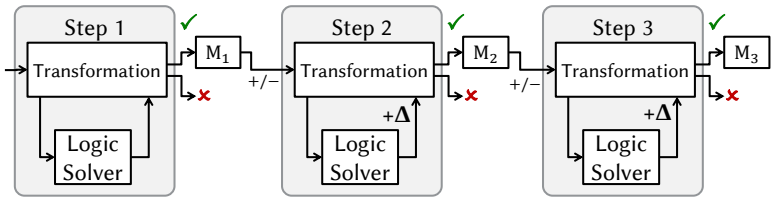


Figure 10.3: Overview of iterative model generation

Contribution group 3. I proposed iterative techniques for generating a diverse set of input models with increasing model size and source model candidates using the output of logic solvers.

3.1. *Iterative and Incremental Model Generation.*
I elaborated a decomposition technique for instance models in order to specify a partial solutions for model generation using partial modeling, and metamodel pruning. I proposed an iterative workflow to incrementally generate instance models of increasing size [C10].
3.2. *Diversity and Distance Metrics.*
I proposed a distance metric based on graph shapes [Ren04; RD06] for measuring the diversity of a single model and set of models. I showed correlation between diversity and mutation score in using such models for mutation testing. I proposed an iterative technique for the generation of a diverse set of models using this distance metric [C11][J3].
3.3. *Incremental Synthesis for Bidirectional Transformations of View Models.*
I transformed query-based view specification into logic formulae to automatically synthesize possible source model changes consistent to a view model change [C12][C13][C7].
3.4. *Applications of Results.*
I demonstrated the applicability of the view model maintenance approach in the context of healthcare models developed in the Concerto ARTEMIS project [Con]. I successfully derived a diverse set of statechart models for the industrial Yakindu Statechart Modeling tool [Yak].

Added value Incremental generation simultaneously improves the quality and size of models created by logic solvers. First, significantly larger model instances can be generated with the same solvers using iterative model generation technique (**Ch3: Scalability**). Furthermore, the diversity metrics based on neighborhood shapes [RD06] generalizes existing metrics (such as metamodel coverage and graph isomorphism used in many research papers). Moreover, our model generation technique derives a structurally diverse set of models by calculating the shape of the previous solutions and avoiding those shapes in the next generated model (**Ch4: Diversity**). Finally, incremental model generation also enables to deduce valid source candidates in case of multiple view models with favorable scalability (**Ch3: Scalability**).

Additional applications and related contributions In R3-COP Artemis project [R3c], we carried out test environment generation for autonomous laser guided forklift robots. In collaboration with Ágnes Barta, Zoltán Szatmári and István Majzik we developed a test track generation technique that combines two solvers (Alloy [TJ07] with Sat4j [LBP10] and Z3 [MB08] as backend solvers) in multiple iterative steps to create complex test rooms [Sza16; HMM13].

The proposed diverse model generation approach is applied for testing model-based access control policies in the MONDO collaboration framework [Deb+17]. In collaboration with Gábor Bergmann and Rebeka Farkas, we developed additional distance metrics (e.g. cosine similarity) and proposed a test selection policy. With Mária Bekő we developed test Cypher query [Fra+18] generation framework for graph databases like Neo4j [Web12] using diverse model generation. The approach is presented in the bachelor thesis of Mária Bekő [Bek18] under my supervision.

The forward transformation approach [Deb+14; Gho+15] used in the proposed bidirectional synchronization is developed by Csaba Debreceni and Zoltán Ujhelyi. Publications [C12][C13] introducing backward change propagation are shared contributions with Csaba Debreceni. My contribution was the mapping of view models to logic using a selected part of the model that needs to change. The impact analysis responsible for selecting the changing part is the contribution of Csaba Debreceni.

(For the sake of completeness, a simplified version of impact analysis is included to Section A.5.) [C7] presents another view modeling technique using 3-valued partial model as background framework.

Finally, a multi-phase homework assignment generation is used in the automation of System Modeling course [Rem], where we generated personalized statechart modeling tasks in [Yak].

10.4 Future work

Recent ongoing research of Kristóf Marussy aims to develop an *incremental stochastic analysis* framework for industrial modeling environments using view models [MM18][C7]. In the proposed framework, view models are used to automatically and continuously derive reliability and fault models (like stochastic Petri nets [AMCB84]), that can be incrementally analyzed by suitable numerical solvers. Therefore, the framework is able to give immediate analysis feedback to the developer, or it can prepare optimization processes (like [Heg+11]) for optimizing extra-functional properties (like reliability or mean time to failure).

Our long-term research goal is to *unify and develop automated graph model generation techniques*, which are currently used in several independent research areas. Object-oriented data structures requires diverse generation of valid models [Mil+07; MK01]. Verification techniques like [Ren04; RSW04] are required graph consistency analysis to check (concurrent) data structures. Autogenerated graphs help the testing and benchmarking of graph databases [Bag+17] and modeling environments [FSB04b; Ara+15; Ali+13; Szá+17]. Benchmarking graph databases and modeling applications requires a realistic set of graphs with increasing sizes. Automated generation can help synthesize such a benchmark suite since obtaining real graphs from business use cases is often difficult to due to the protection of intellectual property rights. Automated synthesis of prototypical test contexts [Mic+12; Abd+18] aims to systematically derive previously unanticipated contexts for smart cyber-physical systems (CPS). To unify those research lines in [B14] we outlined the properties of an ideal model generator family which is abbreviated as CoRe-DiSc:

- **Consistent**: A model generator is consistent, if it derives well-formed models (soundness), and able to derive all well-formed models for a given scope (completeness). In this classification, the main focus of this thesis was consistent model generation [C6][J2].
- **Realistic**: A model generator is realistic if it is able to create models that are close to real ones with respect to some metrics. While several graph metrics have been proposed [Ber+13; BNL14; NL15; Izs+13], the characterization of realistic models is a major challenge [Szá+16].
- **Diverse**: A model generator is diverse if it is able to guarantee given a difference between models with respect to some designated distance metrics. As a secondary objective of my work, I proposed neighborhood-based distance metrics in [C11].
- **Scalable**: A model generator is scalable if it is able to create large models in proportional time. Existing benchmarks are typically [Szá+17; Cps] scalable.

Currently, existing model generation approaches developed in different research areas usually support one (or rarely at most two) of these properties. The grand challenge of CoRe-DiSc is to develop an automated model generator which simultaneously satisfies multiple (ideally, all four) properties.

Publications

Number of publications:	17
Number of peer-reviewed journal papers (written in English):	2
Number of articles in journals indexed by WoS or Scopus:	2
Number of publications (in English) with at least 50% contribution:	5
Number of peer-reviewed publications:	17
Number of independent citations (26 March 2019):	53

Publications linked to the theses

	Journal papers	International conference and workshop papers	Book chapters	Extended abstracts
Thesis 1	[J1]	[C4]·[C5]·[C6]·[C7]·[C8]	[B14]	[A15]
Thesis 2	[J2]	[C4]·[C9]	—	—
Thesis 3	[J3]*	[C4]·[C10]·[C11]·[C12]·[C13]	[B14]	—

* These publications are currently under revision.
This classification follows the faculty's Ph.D. publication score system.

Journal Papers

[J1] **Oszkár Semeráth** and Dániel Varró. Evaluating Well-Formedness Constraints on Incomplete Models. *Acta Cybernetica* 23(2), 2017, pp. 687–713. DOI: 10.14232/actacyb.23.2.2017.15

[J2] **Oszkár Semeráth**, Ágnes Barta, Ákos Horváth, Zoltán Szatmári, Dániel Varró. Formal Validation of Domain-Specific Languages with Derived Features and Well-Formedness Constraints. *Software and System Modeling* 16(2), 2017, pp. 357–392. DOI: 10.1007/s10270-015-0485-x

[J3] **Oszkár Semeráth**, Rebeka Farkas, Gábor Bergmann, Dániel Varró. Diversity of Graph Models and Graph Generators in Mutation Testing. *International Journal on Software Tools for Technology Transfer (STTT)*, 2019
▹ *Under revision*

International Conference and Workshop Papers

[C4] **Oszkár Semeráth**. Formal Validation and Model Synthesis for Domain-specific Languages by Logic Solvers. In: *Proceedings of the ACM Student Research Competition at MODELS 2016 co-located with the 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), St. Malo, France, October 3-4, 2016.* 2016

[C5] **Oszkár Semeráth** and Dániel Varró. Graph Constraint Evaluation over Partial Models by Constraint Rewriting. In: *Theory and Practice of Model Transformation - 10th International Conference, ICMT 2017, Held as Part of STAF 2017, Marburg, Germany, July 17-18, 2017, Proceedings*, pp. 138–154. 2017. DOI: 10.1007/978-3-319-61473-1_10

[C6] **Oszkár Semeráth**, András Szabolcs Nagy, and Dániel Varró. A Graph Solver for the Automated Generation of Consistent Domain-Specific Models. In: *40th International Conference on Software Engineering (ICSE 2018)*, Gothenburg, Sweden: ACM, 2018

[C7] Kristóf Marussy, **Oszkár Semeráth**, and Dániel Varró. Incremental View Model Synchronization using Partial Models. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, pp. 323–333. 2018. DOI: 10.1145/3239372.3239412

[C8] **Oszkár Semeráth**, Aren A. Babikian, Sebastian Pilarski, Dániel Varró. VIATRA Solver: A Framework for the Automated Generation of Consistent Domain-Specific Models. In: *Proceedings of the 41th International Conference on Software Engineering: Demonstrations*, pp. 43–46. IEEE / ACM, 2019

[C9] **Oszkár Semeráth**, Ákos Horváth, and Dániel Varró. Validation of Derived Features and Well-Formedness Constraints in DSLs - by Mapping Graph Queries to an SMT-Solver. In: *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, pp. 538–554. 2013. DOI: 10.1007/978-3-642-41533-3_33
▹ *IEEE/ACM Best Paper Award*

[C10] **Oszkár Semeráth**, András Vörös, and Dániel Varró. Iterative and Incremental Model Generation by Logic Solvers. In: *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pp. 87–103. 2016. DOI: 10.1007/978-3-662-49665-7_6

[C11] **Oszkár Semeráth** and Dániel Varró. Iterative Generation of Diverse Models for Testing Specifications of DSL Tools. In: *21st International Conference on Fundamental Approaches to Software Engineering (FASE)*, Thessaloniki, Greece: Springer, 2018

[C12] **Oszkár Semeráth**, Csaba Debreceni, Ákos Horváth, Dániel Varró. Change Propagation of View Models by Logic Synthesis using SAT solvers. In: *Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 8, 2016.* Pp. 40–44. 2016

[C13] **Oszkár Semeráth**, Csaba Debreceni, Ákos Horváth, Dániel Varró. Incremental backward change propagation of view models by logic solvers. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*, pp. 306–316. 2016

Book Chapters

[B14] Dániel Varró, **Oszkár Semeráth**, Gábor Szárnyas, Ákos Horváth. Towards the Automated Generation of Consistent, Diverse, Scalable and Realistic Graph Models. In: *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig*, pp. 285–312. 2018. DOI: 10.1007/978-3-319-75396-6_16

Extended Abstracts

[A15] **Oszkár Semeráth** and Dániel Varró. Validation of Well-formedness Constraints on Uncertain Models. In: *THE 10TH JUBILEE CONFERENCE OF PHD STUDENTS IN COMPUTER SCIENCE*, Szeged, Hungary, 2016

Additional publications (not linked to theses)

International Conference and Workshop Papers

[C16] Gábor Szárnyas, **Oszkár Semeráth**, Benedek Izsó, Csaba Debreceni, Ábel Hegedüs, Zoltán Ujhelyi, Gábor Bergmann. Movie Database Case: An EMF-IncQuery Solution. In: *Proceedings of the 7th Transformation Tool Contest part of the Software Technologies: Applications and Foundations (STAF 2014) federation of conferences, York, United Kingdom, July 25, 2014.* Pp. 103–115. 2014

[C17] Zoltán Micskei, Raimund-Andreas Konnerth, Benedek Horváth, **Oszkár Semeráth**, András Vörös, Dániel Varró. On Open Source Tools for Behavioral Modeling and Analysis with fUML and Alf. In: *Proceedings of the 1st Workshop on Open Source Software for Model Driven Engineering co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems, OSS4MDE@MoDELS 2014, Valencia, Spain, September 28, 2014.* Pp. 31–41. 2014

[C18] Gábor Szárnyas, **Oszkár Semeráth**, István Ráth, Dániel Varró. The TTC 2015 Train Benchmark Case for Incremental Model Validation. In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 24, 2015.* Pp. 129–141. 2015

Bibliography

[Abd+14] Hani Abdeen, Dániel Varró, Houari A. Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedüs, and Ákos Horváth. Multi-objective optimization in rule-based design space exploration. In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pp. 289–300. 2014. DOI: 10.1145/2642937.2643005.

[Abd+18] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pp. 1016–1026. 2018. DOI: 10.1145/3180155.3180160.

[ADW16] Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, and Andrzej Wasowski. Symbolic execution of high-level transformations. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pp. 207–220. 2016.

[AF12] AV Arkhangel'Skii and VV Fedorchuk. *General topology I: basic concepts and constructions dimension theory*. Vol. 17. Springer Science & Business Media, 2012.

[Ali+13] Shaukat Ali, Muhammad Zohaib Z Iqbal, Andrea Arcuri, and Lionel C Briand. Generating test data from OCL constraints with search techniques. *IEEE Trans. Software Eng.* 39(10), 2013, pp. 1376–1402.

[Ali+16] Shaukat Ali, Muhammad Zohaib Iqbal, Maham Khalid, and Andrea Arcuri. Improving the performance of OCL constraint solving with novel heuristics for logical operations: a search-based approach. *Empirical Software Engineering* 21(6), 2016, pp. 2459–2502. DOI: 10.1007/s10664-015-9392-6.

[All] Alloy online tutorial. 2017.

[AMCB84] Marco Ajmone Marsan, Gianni Conte, and Gianfranco Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems (TOCS)* 2(2), 1984, pp. 93–122.

[Ana+10] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling* 9(1), 2010, pp. 69–86.

[Anj+14] Anthony Anjorin, Sebastian Rose, Frederik Deckwerth, and Andy Schürr. Efficient model synchronization with view triple graph grammars. In: *Modelling Foundations and Applications*, pp. 1–17. Springer, 2014.

[Ant+13] Michał Antkiewicz, Kacper Bak, Alexandr Murashkin, Rafael Olaechea, Jia Liang, and Krzysztof Czarnecki. Clafer tools for product line engineering. In: *SPLC*, 2013.

[Ara+15] Vincent Aranega, Jean-Marie Mottu, Anne Etien, Thomas Degueule, Benoit Baudry, and Jean-Luc Dekeyser. Towards an automation of the mutation analysis dedicated to model transformation. *Softw. Test., Verif. Reliab.* 25(5-7), 2015, pp. 653–683.

[ARI] ARINC - Aeronautical Radio, Incorporated. A653 - Avionics Application Software Standard Interface. `http://www.aviation-ia.com/standards`.

[AUT13] AUTOSAR Consortium. *The AUTOSAR Standard.* `http://www.autosar.org/`. 2013.

[BA05] Behzad Bordbar and Kyriakos Anastasakis. Uml2alloy: a tool for lightweight modelling of discrete event systems. In: *IADIS AC*, pp. 209–216. 2005.

[Bag+17] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gMark: schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering* 29(4), 2017, pp. 856–869.

[Bak+16] Kacper Bak, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. Clafer: unifying class and feature modeling. *Software & Systems Modeling* 15, 3 2016, pp. 811–845.

[Bau+06] Benoit Baudry, Trung Dinh-Trong, Jean-Marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, and Yves Le Traon. Model transformation testing challenges. In: *Integration of Model Driven Development and Model Driven Testing*, 2006.

[Bau+14] Benoit Baudry, Martin Monperrus, Cendrine Mony, Franck Chauvel, Franck Fleurey, and Siobhán Clarke. Diversify: ecology-inspired software evolution for diversity emergence. In: *Software Maintenance, Reengineering and Reverse Engineeringn*, pp. 395–398. 2014.

[BC12] Fabian Büttner and Jordi Cabot. Lightweight string reasoning for OCL. In: Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitrios S. Kolovos (eds.), *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Lyngby, Denmark, July 2-5, 2012. Proceedings*, LNCS, vol. 7349, pp. 244–258. Springer, 2012.

[BCE15] Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, eds. *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*. IEEE Computer Society, 2015.

[BCW10] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. Feature and meta-models in clafer: mixed, specialized, and coupled. In: *3rd International Conference on Software Language Engineering*, 2010. DOI: `10.1007/978-3-642-19440-5_7`.

[BEC12] 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 MODELS*, LNCS, vol. 7590, 2012.

[Bek18] Mária Bekő. Functional Testing of Graph Query Engines by Automated Graph Generators. Bachelor thesis. Budapest University of Technology and Economics, 2018.

[Ber+08] Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In: *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, pp. 396–410. 2008. DOI: 10.1007/978-3-540-87405-8_27.

[Ber+10] 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*, LNCS, vol. 6395, Springer, 2010.

[Ber+11] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for emf models. In: Jordi Cabot and Eelco Visser (eds.), *Fourth International Conference on Theory and Practice of Model Transformations*, LNCS, vol. 6707, pp. 167–182. Springer, 2011.

[Ber+12] Gábor Bergmann, István Ráth, Gergely Varró, and Dániel Varró. Change-driven model transformations. *Software & Systems Modeling* 11(3), 2012, pp. 431–461. DOI: 10.1007/s10270-011-0197-9.

[Ber+13] Michele Berlingerio et al. Multidimensional networks: foundations of structural analysis. *World Wide Web* 16(5-6), 2013, pp. 567–593. DOI: 10.1007/s11280-012-0190-4.

[Ber14] Gábor Bergmann. Translating OCL to Graph Patterns. In: *ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems, MODELS 2014*, Springer, 2014.

[BKS02] 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.

[BNL14] F. Battiston, V. Nicosia, and V. Latora. Structural measures for multiplex networks. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics* 89(3), 2014.

[Bro+06] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In: *17th International Symposium on Software Reliability Engineering, 2006. ISSRE '06.* Pp. 85–94. 2006.

[Bru+15] Hugo Bruneliere, Jokin Garcia Perez, Manuel Wimmer, and Jordi Cabot. Emf views: a view mechanism for integrating heterogeneous models. In: *Conceptual Modeling*, pp. 317–325. Springer, 2015.

[BS16] Edouard Batot and Houari Sahraoui. A generic framework for model-set selection for the unification of testing and learning MDE tasks. In: *MODELS*, pp. 374–384. 2016. DOI: 10.1145/2976767.2976785.

[Búr+15] Márton Búr, Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró. Local search-based pattern matching features in EMF-IncQuery. In: *8th International Conference on Graph Transformation*, 2015.

[Büt+12] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. Verification of ATL transformations using transformation models and model finders. In: *14th International Conf. on Formal Engineering Methods,ICFEM'12*, pp. 198–213. LNCS 7635, Springer, 2012.

[BW07] A. D. Brucker and B. Wolff. The HOL-OCL tool. http://www.brucker.ch/. 2007.

[Cab+10] 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), 2010, pp. 335–357.

[CCR07] 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)*, pp. 547–548. ACM, 2007.

[CCR08] J. Cabot, R. Clariso, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In: *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conf. on*, pp. 73–80. 2008.

[CCR14] Jordi Cabot, Robert Clarisó, and Daniel Riera. On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software* 93, 2014, pp. 1–23.

[CE08] M. Clavel and M. Egea. The ITP/OCL tool. `http://maude.sip.ucm.es/itp/ocl/`. 2008.

[CED09] Manuel Clavel, Marina Egea, and Miguel Angel García de Dios. Checking unsatisfiability for OCL constraints. *ECEASST* 24, 2009.

[CGR13] Alcino Cunha, Ana Garis, and Daniel Riesco. Translating between alloy specifications and uml class diagrams annotated with ocl. *Software & Systems Modeling*, 2013, pp. 1–21.

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. Syntax and semantics of datalog. In: *Logic Programming and Databases*. Springer Berlin Heidelberg, 1990, pp. 77–93. DOI: `10.1007/978-3-642-83952-8_6`.

[Cho] Choco. `http://www.emn.fr/z-info/choco-solverp`.

[Cic+10] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. JTL: a bidirectional and change propagating transformation language. In: *Software Language Engineering*, pp. 183–202. Springer, 2010.

[CK13] Glenn Callow and Roy Kalawsky. A satisficing bi-directional model transformation engine using mixed integer linear programming. *Journal of Object Technology* 12(1), 2013, 1: 1–43.

[Con] CONCERTO ARTEMIS project. `concerto-project.org/`.

[Cps] *CPS Benchmark*. `https://github.com/viatra/viatra-cps-benchmark`. The Eclipse Project. 2017.

[Cra+96] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. *KR* 96, 1996, pp. 148–159.

[DC13] Carolina Dania and Manuel Clavel. OCL2FOL+: coping with undefinedness. In: Jordi Cabot, Martin Gogolla, István Ráth, and Edward D. Willink (eds.), *OCL@MoDELS*, CEUR Workshop Proceedings, vol. 1092, pp. 53–62. CEUR-WS.org, 2013.

[Deb+14] Csaba Debreceni, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth, and Dániel Varró. Query-driven incremental synchronization of view models. In: *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, p. 31. 2014.

[Deb+17] Csaba Debreceni, Gábor Bergmann, István Ráth, and Dániel Varró. Enforcing fine-grained access control for secure collaborative modelling using bidirectional transformations. *Software & Systems Modeling*, 2017. DOI: `10.1007/s10270-017-0631-8`.

[Dif] *EMF DiffMerge*. `wiki.eclipse.org/EMF_DiffMerge`. The Eclipse Project.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM* 5(7), 1962, pp. 394–397.

[DMB08] 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, pp. 337–340. Springer-Verlag, 2008.

[Do1] DO-178C, Software Considerations in Airborne Systems and Equipment Certification. Special Committee 205 of RTCA, 2011.

[Do3] DO-330, Software Tool Qualification and Considerations, Radio Technical Commission for Aeronautics., 2011.

[DPV06] Andrea Darabos, András Pataricza, and Dániel Varró. Towards testing the implementation of graph transformations. In: *GTVMT*, ENTCS, Elsevier, 2006.

[Ehr+06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. DOI: 10.1007/3-540-31188-2.

[EKT09] Karsten Ehrig, Jochen Malte Küster, and Gabriele Taentzer. Generating instance models from meta models. *Softw. Syst. Model* 8(4), 2009, pp. 479–500. DOI: 10.1007/s10270-008-0095-y.

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

[ERK99] Hartmut Ehrig, Grzegorz Rozenberg, and Hans-J rg Kreowski. *Handbook of graph grammars and computing by graph transformation*. Vol. 3. world Scientific, 1999.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In: *International conference on theory and applications of satisfiability testing*, pp. 502–518. 2003.

[Fam+13] Michalis Famelis, Rick Salay, Alessio Di Sandro, and Marsha Chechik. Transformation of models containing uncertainty. In: *International Conference on Model Driven Engineering Languages and Systems*, pp. 673–689. 2013.

[Fle+07] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and Yves Le Traon. Towards dependable model transformations: qualifying input test data. *SoSyM*, 2007.

[Fos+07] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems* 29(3), 2007, p. 17.

[Fra+18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: an evolving query language for property graphs. In: *SIGMOD*, pp. 1433–1445. ACM, 2018. DOI: 10.1145/3183713.3190657.

[FSB04a] F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: Testing model transformations. In: *International Workshop on Model, Design and Validation*, pp. 29–40. 2004.

[FSB04b] Franck Fleurey, Jim Steel, and Benoit Baudry. Validation in model-driven engineering: testing model transformations. In: *Model, design and validation, 2004. Proceedings. 2004 first international workshop on*, pp. 29–40. 2004.

[FSC12a] Michalis Famelis, Rick Salay, and Marsha Chechik. Partial models: towards modeling and reasoning with uncertainty. In: *Proceedings of the 34th International Conference on Software Engineering*, pp. 573–583. IEEE Press, 2012.

[FSC12b] Michalis Famelis, Rick Salay, and Marsha Chechik. The semantics of partial model transformations. In: *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, pp. 64–69. 2012.

[GBR05] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and Systems Modeling* 4, 2005, pp. 386–398.

[GC14] Carlos A. Gonzalez and Jordi Cabot. Test data generation for model transformations combining partition and constraint analysis. In: *ICMT*, pp. 25–41. 2014. DOI: 10.1007/978-3-319-08789-4_3.

[Gho+15] Hamid Gholizadeh, Zinovy Diskin, Sahar Kokaly, and Tom Maibaum. Analysis of source-to-target model transformations in quest. In: *Proceedings of the 4th Workshop on the Analysis of Model Transformations*, pp. 46–55. 2015.

[GK15] Loïc Gammaitoni and Pierre Kelsen. F-alloy: an alloy based model transformation language. In: *Theory and Practice of Model Transformations*, pp. 166–180. Springer, 2015.

[GM09] Yeting Ge and Leonardo Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Ahmed Bouajjani and Oded Maler (eds.), *Computer Aided Verification*, LNCS, vol. 5643, pp. 306–320. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-02658-4_25.

[Gon+12] Carlos A. González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. Emftocsp: a tool for the lightweight verification of EMF models. In: *Proceedings of the First International Workshop on Formal Methods in Software Engineering - Rigorous and Agile Approaches, FormSERA 2012, Zurich, Switzerland, June 2, 2012*, pp. 44–50. 2012. DOI: 10.1109/FormSERA.2012.6229788.

[GRR09] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System model-based definition of modeling language semantics. In: *Formal Techniques for Distributed Systems*, LNCS, vol. 5522, pp. 152–166. Springer, 2009.

[GS15] Esther Guerra and Mathias Soeken. Specification-driven model transformation testing. *Softw. Syst. Model.* 14(2), 2015, pp. 623–644. DOI: 10.1007/s10270-013-0369-x.

[Hay+01] Kelly J Hayhurst, Dan S Veerhusen, John J Chilenski, and Leanna K Rierson. A practical tutorial on modified condition/decision coverage, 2001.

[Heg+11] Ábel Hegedüs, Ákos Horváth, István Ráth, and Dániel Varró. A model-driven framework for guided design space exploration. In: *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, IEEE Computer Society, 2011.

[Heg+16] Ábel Hegedüs, Ákos Horváth, István Ráth, Rodrigo Rizzi Starr, and Dániel Varró. Query-driven soft traceability links for models. *Software & Systems Modeling* 15(3), 2016, pp. 733–756.

[Her+15] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, Yingfei Xiong, Susann Gottmann, and Thomas Engel. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Software & Systems Modeling* 14(1), 2015, pp. 241–269.

[Het10] Thomas Hettel. Model round-trip engineering. PhD thesis. Queensland University of Technology, 2010.

[HHV15] Ábel Hegedüs, Ákos Horváth, and Dániel Varró. A model-driven framework for guided design space exploration. *Automated Software Engineering* 22(3), 2015, pp. 399–436.

[Hid+15] Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. Feature-based classification of bidirectional transformation approaches. *Software & Systems Modeling*, 2015, pp. 1–22. DOI: 10.1007/s10270-014-0450-0.

[HLR06] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental model transformation for the evolution of model-driven systems. In: *Model Driven Engineering Languages and Systems*, pp. 321–335. Springer, 2006.

[HMM13] Gergő Horányi, Zoltán Micskei, and István Majzik. Scenario-based automated evaluation of test traces of autonomous systems. In: Matthieu Roy (ed.), *Proceedings of ERCIM/EWICS Workshop on Dependable Embedded and Cyber-physical Systems (DECS) at SAFECOMP'13*, pp. 181–192. 2013.

[Hor+14] Ákos Horváth, Ábel Hegedüs, Márton Búr, Dániel Varró, Rodrigo Rizzi Starr, and Samoel Mirachi. Hardware-software allocation specification of ima systems for early simulation. In: *Digital Avionics Systems Conference (DASC)*, IEEE, 2014.

[Izs+13] Benedek Izsó, Zoltán Szatmári, Gábor Bergmann, Ákos Horváth, and István Ráth. Towards precise metrics for predicting graph query performance. In: *ASE*, pp. 421–431. 2013. DOI: 10.1109/ASE.2013.6693100.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11(2), 2002, pp. 256–290.

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37(5), 2011, pp. 649–678.

[JLB11] Ethan K Jackson, Tihamer Levendovszky, and Daniel Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In: *Model Driven Engineering Languages and Systems*, pp. 653–667. Springer, 2011.

[JS06] Ethan K. Jackson and Janos Sztipanovits. Towards a formal foundation for domain specific modeling languages. In: *Proceedings of the 6th ACM / IEEE Int. Conf. on Embedded Software*, EMSOFT '06, pp. 53–62. ACM, 2006.

[JS07] Ethan K Jackson and Janos Sztipanovits. Constructive techniques for meta-and model-level reasoning. In: *Model Driven Engineering Languages and Systems*, pp. 405–419. Springer, 2007.

[JSB12] Ethan K. Jackson, Wolfram Schulte, and Nikolaj Bjørner. Detecting specification errors in declarative languages with constraints. In: *Proc. of the 15th Int. Conf. on MODELS*, LNCS, vol. 7590, pp. 399–414. 2012.

[JSS13] Ethan K Jackson, Gabor Simko, and Janos Sztipanovits. Diversely enumerating system-level architectures. In: *Proceedings of the 11th ACM Int. Conf. on Embedded Software*, p. 11. 2013.

[KG12a] Mirco Kuhlmann and Martin Gogolla. From UML and OCL to relational logic and back. In: *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, pp. 415–431. 2012. DOI: 10.1007/978-3-642-33666-9_27.

[KG12b] 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*, LNCS, vol. 7349, pp. 32–48. 2012.

[KHG11] 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*, LNCS, vol. 6705, pp. 290–306. 2011.

[KJS11] Eunsuk Kang, Ethan Jackson, and Wolfram Schulte. An approach for effective design space exploration. In: Radu Calinescu and Ethan Jackson (eds.), *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, LNCS, vol. 6662, pp. 33–54. Springer Berlin Heidelberg, 2011.

[Kle+52] Stephen Cole Kleene, NG De Bruijn, J de Groot, and Adriaan Cornelis Zaanen. *Introduction to metamathematics*. Vol. 483. van Nostrand New York, 1952.

[KPP09] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. On the evolution of ocl for capturing structural constraints in modelling languages. In: *Rigorous Methods for Software Construction and Analysis*, pp. 204–218. 2009.

[KV09] Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In: Renate A. Schmidt (ed.), *Automated Deduction – CADE-22*, LNCS, vol. 5663, pp. 199–213. Springer Berlin Heidelberg, 2009.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pp. 1–35. 2013. DOI: 10.1007/978-3-642-39799-8_1.

[LBA10] Levi Lucio, Bruno Barroca, and Vasco Amaral. A technique for automatic validation of model transformations. In: *Proc. of the 13th Int. Conf. on MODELS*, LNCS, vol. 6394, pp. 136–150. 2010.

[LBP10] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation* 7, 2010, pp. 59–64.

[Lia12] Jia Liang. Solving clafer models with choco. (GSDLab-TR 2012-12-30), 2012.

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

[MB08] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS 2008)*, LNCS, vol. 4963, pp. 337–340. Springer, 2008.

[MBT06] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Mutation analysis testing for model transformations. In: *ECMDA-FA*, LNCS, vol. 4066, pp. 376–390. Springer, 2006. DOI: 10.1007/11787044_28.

[MC13] Nuno Macedo and Alcino Cunha. Implementing QVT-R bidirectional model transformations using Alloy. In: *Fundamental Approaches to Software Engineering*, pp. 297–311. 2013.

[MHT04] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In: *Mathematics of Program Construction*, pp. 289–313. 2004.

[Mic+12] Zoltán Micskei, Zoltán Szatmári, János Oláh, and István Majzik. A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In: *KES-AMSTA*, LNCS, vol. 7327, pp. 504–513. Springer, 2012. DOI: 10.1007/978-3-642-30947-2_55.

[Mil+07] Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. Korat: A tool for generating structurally complex test inputs. In: *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pp. 771–774. 2007. DOI: 10.1109/ICSE.2007.48.

[Mil+15] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. Alloy*: A general-purpose higher-order relational constraint solver. In: *37th IEEE/ACM Int. Conf. on Software Engineering, ICSE*, pp. 609–619. 2015.

[MK01] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In: *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*, p. 22. 2001. DOI: 10.1109/ASE.2001.989787.

[MM16] Ferenczi Miklós and Szőts Miklós. Mathematical Logic for Applications. In: Varasdi Károly (ed.), Typotex, 2016.

[MM18] Kristóf Marussy and István Majzik. Constructing dependability analysis models of reconfigurable production systems. In: *14th IEEE International Conference on Automation Science and Engineering, CASE 2018, Munich, Germany, August 20-24, 2018*, pp. 1158–1163. 2018. DOI: 10.1109/COASE.2018.8560551.

[Mot+15] Jean-Marie Mottu, Sagar Sen Simula, Juan Cadavid, and Benoit Baudry. Discovering model transformation pre-conditions using automatically generated test models. In: *ISSRE*, pp. 88–99. IEEE, 2015. DOI: 10.1109/ISSRE.2015.7381802.

[Mou+09] Alix Mougenot, Alexis Darrasse, Xavier Blanc, and Michèle Soria. Uniform random generation of huge metamodel instances. In: *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pp. 130–145. Springer-Verlag, 2009.

[NL15] Vincenzo Nicosia and Vito Latora. Measuring and modeling correlations in multiplex networks. *Phys. Rev. E* 92, 3 2015, p. 032805. DOI: 10.1103/PhysRevE.92.032805.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The fujaba environment. In: *Proceedings of the 22nd international conference on Software engineering*, pp. 742–745. 2000.

[Ocl] *Object Constraint Language, v2.4*. The Object Management Group. 2014.

[Ola+12] Rafael Olaechea, Steven Stewart, Krzysztof Czarnecki, and Derek Rayside. Modeling and multi-objective optimization of quality attributes in variability-rich software. In: *International Workshop on Non- functional System Properties in Domain Specific Modeling Languages*, 2012.

[OMG] OMG. MOF 2.0 Query/View/Transformation specification (QVT), version 1.1. http://www.omg.org/spec/QVT/1.2/.

[Pen08] Karl-Heinz Pennemann. Resolution-like theorem proving for high-level conditions. In: *International Conference on Graph Transformation*, pp. 289–304. 2008.

[PMB08] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjorner. Deciding Effectively Propositional Logic with Equality. Microsoft Research, MSR-TR-2008-181 Technical Report. 2008.

[Que+12] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. OCL-Lite: finite reasoning on UML/OCL conceptual schemas. *Data Knowl. Eng.* 73, 2012, pp. 1–22.

[R3c] R3-COP (Resilient Reasoning Robotic Co-operative Systems). ARTEMIS project n° 100233, `http://http://www.r3-cop.eu/`.

[Rad+15] Hendrik Radke, Thorsten Arendt, Jan Steffen Becker, Annegret Habel, and Gabriele Taentzer. Translating essential ocl invariants to nested graph constraints focusing on set operations. In: *International Conference on Graph Transformation*, pp. 155–170. 2015.

[Rát+08] István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. Live model transformations driven by incremental pattern matching. In: *Proc. First International Conference on the Theory and Practice of Model Transformations (ICMT 2008)*, LNCS, vol. 5063, pp. 107–121. Springer Berlin-Heidelberg, 2008. DOI: `10.1007/978-3-540-69927-9_8`.

[RD06] Arend Rensink and Dino Distefano. Abstract graph transformation. *Electronic Notes in Theoretical Computer Science* 157(1), 2006, pp. 39–59.

[Rei97] Stuart C Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In: *Software Metrics Symposium*, pp. 64–73. 1997.

[Rem] *System Modeling*. `https://portal.vik.bme.hu/kepzes/targyak/VIMIAA00/en/`. Budapest University of Technology and Economics.

[Ren04] Arend Rensink. Canonical graph shapes. In: *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004*, pp. 401–415. 2004. DOI: `10.1007/978-3-540-24725-8_28`.

[Ren06] Arend Rensink. Isomorphism checking in GROOVE. *ECEASST* 1, 2006.

[Res] Microsoft Research. Pex. `http://research.microsoft.com/projects/pex/`.

[RHV12] István Ráth, Ábel Hegedüs, and Dániel Varró. Derived features for EMF by integrating advanced model queries. In: *Modelling Foundations and Applications*, LNCS, pp. 102–117. Springer Berlin / Heidelberg, 2012.

[RSW04] Thomas W Reps, Mooly Sagiv, and Reinhard Wilhelm. Static program analysis via 3-valued logic. In: *International Conference on Computer Aided Verification*, pp. 15–30. 2004.

[RV16] Daniel Ratiu and Markus Voelter. Automated testing of DSL implementations: experiences from building mbeddr. In: *AST@ICSE 2016*, pp. 15–21. 2016. DOI: `10.1145/2896921.2896922`.

[RV99] Alexandre Riazanov and Andrei Voronkov. Vampire. In: *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, pp. 292–296. 1999. DOI: `10.1007/3-540-48660-7\_26`.

[RZ12] Arend Rensink and Eduardo Zambon. Pattern-based graph abstraction. In: *Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings*, pp. 66–80. 2012. DOI: `10.1007/978-3-642-33654-6_5`.

[SAB09] Seyyed M. A. Shah, Kyriakos Anastasakis, and Behzad Bordbar. From UML to Alloy and back again. In: *MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pp. 1–10. ACM, 2009.

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

[Sal+15] Rick Salay, Marsha Chechik, Michalis Famelis, and Jan Gorzny. A methodology for verifying refinements of partial models. *Journal of Object Technology* 14(3), 2015, 3:1–31.

[SBM09] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic model generation strategies for model transformation testing. In: *ICMT*, pp. 148–164. 2009. DOI: 10.1007/978-3-642-02408-5_11.

[SC15] Rick Salay and Marsha Chechik. A generalized formal framework for partial modeling. In: Alexander Egyed and Ina Schaefer (eds.), *Fundamental Approaches to Software Engineering*, LNCS, vol. 9033, pp. 133–148. Springer Berlin Heidelberg, 2015.

[SCG12] Rick Salay, Marsha Chechik, and Jan Gorzny. Towards a methodology for verifying partial model refinements. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 938–945. 2012.

[Sch+13] J. Schonbock, G. Kappel, M. Wimmer, A. Kusel, W. Retschitzegger, and W. Schwinger. TETRABox - a generic white-box testing framework for model transformations. In: *APSEC*, pp. 75–82. IEEE, 2013. DOI: 10.1109/APSEC.2013.21.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In: *Graph-Theoretic Concepts in Computer Science*, pp. 151–163. 1994.

[Sen+09] Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel. Meta-model Pruning. In: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2009.

[Sen+12] 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*, LNCS, vol. 7307, pp. 24–39. 2012.

[SFC12] Rick Salay, Michalis Famelis, and Marsha Chechik. Language independent refinement using partial modeling. In: Juan de Lara and Andrea Zisman (eds.), *Fundamental Approaches to Software Engineering*, LNCS, vol. 7212, pp. 224–239. Springer Berlin Heidelberg, 2012.

[Sir] *Sirius*. http://www.eclipse.org/sirius. The Eclipse Project. 2017.

[SLO17] Sven Schneider, Leen Lambers, and Fernando Orejas. Symbolic model generation for graph properties. In: *International Conference on Fundamental Approaches to Software Engineering*, pp. 226–243. 2017.

[Soe+10] 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)*, pp. 1341–1344. IEEE, 2010.

[Sol16] Alexandra Anna Solyom. Tesztkörnyezetek előállítása automatikus szabályalapú modellgenerálás segítségével. Masters thesis. Budapest University of Technology and Economics, 2016.

[SSB17] Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C. Briand. Synthetic data generation for statistical testing. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pp. 872–882. 2017. DOI: 10.1109/ASE.2017.8115698.

[Ste08] Perdita Stevens. Bidirectional model transformations in qvt: semantic issues and open questions. *Software & Systems Modeling* 9(1), 2008, pp. 7–20. DOI: 10.1007/s10270-008-0109-9.

[Szá+16] G. Szárnyas, Z. Kővári, Á. Salánki, and D. Varró. Towards the characterization of realistic models: evaluation of multidisciplinary graph metrics. In: *MODELS*, 2016. DOI: 10.1145/2976767.2976786.

[Sza16] Zoltán Szatmári. Metamodel-based model generation and validation techniques with applications. PhD dissertation. Budapest University of Technology and Economics, 2016.

[Szá+17] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. The Train Benchmark: cross-technology performance evaluation of continuous model queries. *Softw. Syst. Model.* 2017. DOI: 10.1007/s10270-016-0571-8.

[Szá19] Gábor Szárnyas. Query, Analysis, and Benchmarking Techniques for Evolving Property Graphs of Software Systems. PhD thesis. Budapest University of Technology and Economics, 2019.

[THR10] Paolo Torrini, Reiko Heckel, and István Ráth. Stochastic simulation of graph transformation systems. In: *FASE*, LNCS, vol. 6013, pp. 154–157. Springer, 2010. DOI: 10.1007/978-3-642-12029-9_11.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: a relational model finder. In: *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 632–647. Springer, 2007.

[Ujh+15] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.* 98, 2015, pp. 80–99.

[Val+12] Antonio Vallecillo, Martin Gogolla, Loli Burgueño, Manuel Wimmer, and Lars Hamann. Formal specification and testing of model transformations. In: *SFM*, pp. 399–437. 2012. DOI: 10.1007/978-3-642-30982-3_11.

[Var+16] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software and System Modeling* 15(3), 2016, pp. 609–629. DOI: 10.1007/s10270-016-0530-4.

[VB07] Dániel Varró and András Balogh. The Model Transformation Language of the VIATRA2 Framework. *Science of Computer Programming* 68(3), 2007, pp. 214–234.

[Vs] *Viatra Solver*. 2018.

[VSV05] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for graph transformation. In: *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), 21-24 September 2005, Dallas, TX, USA*, pp. 79–88. 2005. DOI: 10.1109/VLHCC.2005.23.

[Web12] Jim Webber. A programmatic introduction to Neo4j. In: *SPLASH*, pp. 217–218. 2012. DOI: 10.1145/2384716.2384777.

[WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software* 31(3), 2014, pp. 79–85.

[Wie+12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming* 12(1-2), 2012, pp. 67–96.

[Wil12] E. D. Willink. An extensible OCL virtual machine and code generator. In: *Proc. of the 12th Workshop on OCL and Textual Modelling*, pp. 13–18. ACM, 2012.

[Win+08] 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), 2008. Proc. of the 5th Int. Workshop on Graph Transformation and Visual Modeling Techniques, pp. 159 –170. DOI: 10.1016/j.entcs.2008.04.038.

[WKC06] Junhua Wang, S-K Kim, and David Carrington. Verifying metamodel coverage of model transformations. In: *Software Engineering Conference*, 10–pp. 2006.

[Xio+07] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In: *Proceedings of the 22nd IEEE/ACM Int. Conf. on Automated Software Engineering*, pp. 164–173. 2007.

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

[Yak] *Yakindu*. http://statecharts.org/. Yakindu Statechart Tools. 2017.

Appendix A

Appendix

A.1 Transforming OCL invariants to first order logic

OCL constraints (invariants) are widely used means to express *well-formedness rules* of DSLs which has to be satisfied by all valid instance models. Here we present a transformation from a subset of OCL invariants to FOL. As a result, DSL validation tasks (e.g. consistency check, subsumption check) can be executed even when certain WF constraints are defined by graph patterns while others are captured by OCL invariants. This is a very practical setup to allow DSL engineers to mix specification languages.

While there are existing OCL-to-FOL transformations (e.g. [CED09; DC13]) which cover a larger portion of the OCL language (and its semantic cornercases), our technique allows to detect inconsistencies between WF constraints with different representations (OCL vs GP) and allows to reason about subsumption or equivalence of such constraints. Furthermore, the combined use of our rich partial snapshot language and OCL invariants also allows to gain additional insight into DSL specifications. Finally, we also introduce approximations for certain OCL language elements.

A.1.1 An overview of OCL transformation

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

`context` ⟨C⟩ `inv` ⟨*name*⟩ : ⟨*expression*⟩

Our transformation takes an OCL invariant as input and synthesizes FOL formulae as output. Certain OCL language elements are too expressive to be represented in FOL, but they can be approximated by appropriate FOL formulae. Each supported language element is presented in this subsection.

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

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

The mapping algorithm traverses the *abstract syntax tree* (AST) of the OCL invariant recursively and applies the corresponding rules to each subexpression element. However, there are some special cases which require pre- or post-processing the result, e.g. for comparison functions, null references

and collection operators. The mapping also handles a restricted set of higher-order structures (like sets) which structures are unfolded and represented by FOL predicates.

A.1.1.1 Basic expressions

The mapping rules of the basic expressions (primitive-type, simple arithmetic and bool expression and variable) of OCL are depicted in Table A.1. In OCL, we cover the primitive types Integer, Boolean and Real, while the String types is not yet supported. The logic and arithmetic operators are directly transformed to FOL, since they have their equivalent counterpart in the FOL if they are interpreted on Integers, Reals and Booleans.

OCL	`var`	`a+b`	`a-b`	`a*b`	`a/b`	`a=b`
FOL	var	$a + b$	$a - b$	$a \cdot b$	a/b	$a = b$

OCL	`a and b`	`a or b`	`a implies b`	`not a`
FOL	$a \wedge b$	$a \vee b$	$a \Rightarrow b$	$\neg a$

Table A.1: Mapping of basic OCL expressions

The OCL function `oclIsKindOf(Class)` is translated to a type predicate of *EClass*, which is satisfied if and only if the object has the same type as the argument of the function.

OCL	v.`oclIsKindOf(`$\langle C \rangle$`)`
FOL	$C(v)$

Objects, as complex structures require special transformation rules. *Single-valued attributes* of objects are translated to functions.

OCL	$v_1 . \langle F \rangle = v_2$,$F \in Ref_{MM}$ or $F \in Attr_{MM}$
FOL	$F(v_1, v_2)$

The *equal* operator of OCL (==) is transformed similarly to other mathematical operators unless objects need to be compared. The transformation of such equality expressions are divided into two cases: (1) if a variable is placed on the right hand side, then an existential quantifier is used to avoid undefined values, (2) if the comparison is to value `null`, then the expression is transformed using a negated existential qualifier in FOL.

OCL	$v_1 . \langle F \rangle$=v_2	$v_1 . \langle F \rangle$=`null`
FOL	$\exists v_2 : F(v_1, v_2)$	$\neg \exists v_2 : F(v_1, v_2)$

The transformation of the *not equal* operator of OCL (<>) uses the dual counterparts of equal operation by adding a negation to the corresponding FOL expressions.

A.1.2 Collections

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

In our approach, every collection C is represented by a characteristic predicate $P_C(x)$ captured in FOL, where the predicate evaluates to *true* on a model element only if it is the member of the collection:

$$P_C(x) \Leftrightarrow x \in C.$$

If the collection can be implied by the context, the index of the collection is omitted: $P(x)$.

The OCL operation `allInstances` refers to all instances of a certain type, that way, the corresponding FOL formula collects the objects with the referred type. We also can refer to the set of elements with a certain type as defined by the context of the invariant using `self` keyword, which is handled exactly as the `allInstances` construct.

OCL	`context`⟨C⟩`inv:...self...`, $C \in Cls_{MM}$
OCL	⟨C⟩`.allInstances()`, $C \in Cls_{MM}$
FOL	$P(x) := C(x)$

A set of instances can be also referred by a reference with more than one multiplicity. The predicate of the reference and the predicate of the variable is included in the FOL expression during the mapping.

OCL	$v\,.\langle F\rangle$, $F \in Ref_F$ or $F \in Attr_F$
FOL	$P(x) := F(v, x)$

Navigation along references (see next example) with more than one multiplicity is also alloved in OCL, but it is a shorthand of the `collect` operator and is transformed to this operator directly by the OCL parser (see below). The equivalent FOL expression contains the predicates of the references included in the path and a temporary variables with universal quantifier for the intermediate points. The predicates of the intermediate- and endpoints are not needed, since the predicates of the references include them.

OCL	$v\,.\langle F_1\rangle.....\langle F_n\rangle$, for $0 \le i \le n$: $F_i \in Ref_F$ or $F_i \in Attr_F$
FOL	$P(x) := \exists v_1, \ldots, v_{n-1} : F_1(v, v_1) \wedge$ $\bigwedge_{2 \le i \le n-2} F_i(v_{i-1}, v_i) \wedge$ $F_n(v_{n-1}, x)$

The `collect` operator derives a collection from another by applying an $exp(v)$ OCL expression on the elements of the collection v, which is defined by other mapping rules.

OCL	C `->` `collect(`$v \mid exp(v)$`)`
FOL	$P(x) := \exists v : P_C(v) \wedge exp(v) = x$

The `closure` operator derives a collection using an $exp(v)$ expression by iteratively applying on the elements of the collection until it reaches fix point. The transitive closure in ocl approximated similarly as in the case of graph patterns. The following transformation presents the overapproximation of the `closure` operator by 3 steps:

OCL	c -> `closure`$(v \mid exp(v))$
FOL	$P_{O=3}(x) := P_c(x) \vee \exists v : P_c(v) \wedge$ $((x = exp(v) \wedge distinct(v, x)) \vee$ $(x = exp^2(v) \wedge distinct(v, exp(v), x)) \vee$ $(x = exp^3(v) \wedge distinct(v, exp(v), exp^2(v), x)) \vee$ $(true \wedge distinct(v, exp(v), exp^2(v), exp^3(v), x)))$

The `select` and `reject` operators are used to define a special subset of the collection by an expression $exp(v)$. The `select` constructs a condition, which elements should be included, while the `reject` defines the elements that should be excluded from the collection. The transformation of these operators appends the transformation of the expression to end of the predicate of the collection.

OCL	C->`select`$(v \mid exp(v))$	C->`reject`$(v \mid exp(v))$
FOL	$P(x) := P_C(x) \wedge exp(x)$	$P(x) := P_C(x) \wedge \neg exp(x)$

A.1.2.1 Collection operators

We now overview the transformation of different OCL operators which are applicable to collections.

The OCL operation `includes` is evaluated to *true* if the collection contains at least one element satisfying the argument expression of the function. This function is translated to a predicate which checks containment. The `excludes` OCL function is the dual of `includes`, so it is transformed to the negated FOL expression of `includes`. This expression is satisfied if the elements of the collection do not satisfy the condition.

OCL	C->`includes`(v)	C->`excludes`(v)
FOL	$P_C(v)$	$\neg P_C(v)$

OCL operation `forAll` is an iterator over a collection to state that certain conditions hold for each member of the collection. We restrict our transformation to set semantics, and then the FOL equivalent is the universal quantifier. The OCL operation `exists` implements an iterator and the FOL equivalent is the existential quantifier.

OCL	C->`forAll`$(v \mid exp(v))$	C->`exists`$(v \mid exp(v))$
FOL	$\forall v : P_C(v) \Rightarrow exp(v)$	$\exists v : P_C(v) \wedge exp(v)$

The OCL function `notEmpty` is applied on collections and is satisfied if the collection is not empty. This operation is transformed to an existentially quantified predicate which means that there is at least one object in the given set or collection. The OCL function `isEmpty` handled with an additional negation.

OCL	C->`notEmpty`()	C->`isEmpty`()
FOL	$\exists v : P_C(v)$	$\neg\exists v : P_C(v)$

The OCL function `size` returns the size of the collection. In FOL the size of a collection cannot be formulated in general, but if the complarison of the size of the collection to an integer is translated similarly as in case of handling multiplicities in metamodels (see Section 3.3.2.2).

We implemented transformations using approximation for every comparison operators. The transformation of the less than (=<) and greater than (=>) operators provides the basis for the mapping of the equality and inequality operators.

The translation of the greater than (=>) operator is carried out by introducing temporary variables (as much as needed), adding the predicates of the reference for each variable and finally declare pairwise inequality.

OCL	C->`size()`>=n, $n \in \mathbb{Z}^+$
FOL	$\exists v_1, \ldots, v_n : distinct(v_1, \ldots, v_n) \wedge \bigwedge_{1 \leq i \leq n} P_c(v_i)$

The translation of the less than (=<) operator is similar, but the equality is declared at least for one pair of the temporary variables.

OCL	C->`size()`<=n,$n \in \mathbb{Z}^+$
FOL	$\neg\exists v_1, \ldots, v_n : distinct(v_1, \ldots, v_n) \wedge \bigwedge_{1 \leq i \leq n} P_c(v_i)$

Finally, the equality operator is divided into two parts before the mapping:

OCL	C->`size()`=n, $n \in \mathbb{Z}^+$
OCL	C->`size()`<=n `and` C->`size()`>=n

The handling of equality and inequality operators has specific rules if two collections are passed as input. Two collections are equal if and only if neither of them contains an element which is not in the other set.

OCL	C_1=C_2
FOL	$\neg\exists v : (P_{C_1}(v) \wedge \neg P_{C_2}(v)) \vee (\neg P_{C_1}(v) \wedge P_{C_2}(v))$

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

OCL	C_1<>C_2
FOL	$\exists v : (P_c(v) \wedge \neg P_d(v)) \vee (\neg P_c(v) \wedge P_d(v))$

A.1.2.2 Restrictions and expressiveness

The expressiveness of OCL is higher than first-order logic, so some language constructs are obviously not covered by our transformation. Due to the undecidable nature of the full OCL language, it cannot be expected to come up with an automated unsatisfiability checker for all the OCL expressions. Still, we believe that covering a subclass of OCL expressions and support language-level validation on them is a practical solution.

In our approach, the OCL constructs like OrderSet, Bag and Sequence and operations like max() and min() are not handled. We only focus on OCL invariants and do not support general OCL queries or operation constraints captured by pre- and postconditions. The list of the supported language elements is overviewed in Table A.2.

Features of the OCL	FOL	EPR	BA
Logic operators	+	+	+
Arithmetic operators	+	−	A
oclIsTypeOf	+	+	+
Attributes	+	+	+
References	+	+	+
Collections (Sets)	+	+	+
Collections (Bag, Sequence)	−	−	−
allInstances, self	+	+	+
Iterator expressions (e.g. exists, forAll)	+	−	+
notEmpty	+	−	+
isEmpty	+	+	+
Transitive closure	+	A	+
Aggregated expressions	−	−	−

+: Expressible, −: Inexpressible, A: Approximable

Table A.2: Expressing OCL features in FOL

A.2 Implicit equivalence check rewriting

First, we present a rewriting technique to eliminate implicit equivalence checks. For the rewriting variables in a predicate, we use the following construction:

Definition 38 (Variable rewriting) *Let $\varphi(v_1, \ldots, v_n)$ denote a first order logic predicate, $o_1, \ldots, o_m$ be variable occurrences in $\varphi(v_1, \ldots, v_n)$, and $v_1, \ldots, v_m$ be variable symbols. A variable rewriting of $\varphi(v_1, \ldots, v_n)$ from $o_1, \ldots, o_m$ to $v_1, \ldots, v_m$ is a predicate (denoted by $\varphi_{o_1 \mapsto v_1, \ldots, o_m \mapsto v_m}(v_1, \ldots, v_n)$), where each occurrence o_i $(1 \leq i \leq m)$ of is replaced with a new occurrence of variable symbol v_i.*

Example 41. Let $\varphi(v) := \mathsf{Transition}(\underbrace{e}_{o_1}) \wedge \mathsf{source}(\underbrace{e}_{o_2}, v)$ denote a predicate with two variable occurences o_1 and o_2 of variable symbol e, and let n_1 and n_2 denote new variable symbols. Then

$$\varphi_{o_1 \mapsto n_1, o_2 \mapsto n_1}(e) = \mathsf{Transition}(n_1) \wedge \mathsf{source}(n_2, v)$$

The input of a the rewriting is a first order logic predicate $\varphi(v_1, \ldots, v_n)$, and the output is an *unfolded predicate* $unf[\varphi(v_1, \ldots, v_n)]$, which is semantically equivalent $\varphi(v_1, \ldots, v_n) \leftrightarrow unf[\varphi(v_1, \ldots, v_n)]$, but it does not contain any implicit equivalence checks. The unfolding is constructed in two steps: $unf_{\forall\exists}[\cdot]$ rewrites all quantified variables, and $unf_F[\cdot]$ rewrites the free variables. Thus, the predicate is rewritten to

$$unf[\varphi(v_1, \ldots, v_n)] := unf_F[unf_{\forall\exists}[\varphi(v_1, \ldots, v_n)]].$$

First, the quantified variables are unfolded with $unf_{\forall\exists}[\cdot]$ recursively:

- If $\varphi(v_1, \ldots, v_n) := R(\cdots)$ or $R^+(\cdot, \cdot)$ is an atomic expression, then it is not changed:

$$unf_{\forall\exists}[\varphi(v_1, \ldots, v_n)] := \varphi(v_1, \ldots, v_n)$$

- If $\varphi(v_1, \ldots, v_n) := \mathsf{X}v : \varphi'(v'_1, \ldots, v'_m)$ is a quantified expression (where X denote either $\forall$ or $\exists$), then let $o_1, \ldots, o_l$ denote the occurrences of symbol v in φ' bound by φ, and $u_1, \ldots, u_l$ denote a set of new variable symbols.

$$unf_{\forall\exists}[\mathsf{X}v : \varphi'(v'_1, \ldots, v'_m)] := \mathsf{X}v : \exists u_1, \ldots, u_l : unf_{\forall\exists}[\varphi'_{o_1 \mapsto u_1, \ldots, o_l \mapsto u_l}(v'_1, \ldots, v'_m)] \wedge \bigwedge_{1 \leq i \leq l} \{u_i \sim v\}$$

- Otherwise, $unf_{\forall\exists}[\varphi(v_1, \ldots, v_n)]$ rewrites all subexpressions recursively.

Next, free variables are unfolded with $unf_F[\cdot]$. In predicate $\varphi(v_1, \ldots, v_n)$ let $O = \{o_1, \ldots, o_m\}$ denote all variable occurrences of free variable symbols $V = \{v_1, \ldots, v_n\}$, and let function $ref : O \to V$ denote the referred variable of an occurrence. Then, all variable occurrences O are renamed to a new set unused variable symbols $u_1, \ldots, u_m$. Then, the unfolded predicate of φ is created as follows:

$$unf_F[\varphi(v_1, \ldots, v_n)] := \exists u_1, \ldots, u_l : \varphi_{o_1 \mapsto u_1, \ldots, o_m \mapsto u_1}(u_1, \ldots, u_m) \wedge \bigwedge_{1 \leq i \leq m} \{u_i \sim ref(u_i)\}$$

Example 42. Let $\varphi(v) := \exists e : \mathsf{Transition}(e) \wedge \mathsf{source}(e, v)$ denote a predicate. The unfolding of this predicate is the follows:

$$unf[\varphi(v)] = \exists e : \exists u_1, u_2 : \mathsf{Transition}(u_1) \wedge \mathsf{source}(u_2, v) \wedge e \sim u_1 \wedge e \sim u_2$$

Next, lets rewrite the free variables:

$$unf_F[unf[\varphi(v)]] = \exists u_3 : \exists e : \exists u_1, u_2 : \mathsf{Transition}(u_1) \wedge \mathsf{source}(u_2, u_3) \wedge e \sim u_1 \wedge e \sim u_2 \wedge v \sim u_3$$

A.3 Partial models

First, we show that the information ordering relation ($\sqsubseteq$) ensures the under- and over-approximation rules for any 3-valued truth value.

Lemma 1 (Information order vs Under- and over-approximation) *If X and Y are 3-valued truth values with $X \sqsubseteq Y$, then $(X = 1) \Rightarrow (Y = 1)$ and $(Y = 1) \Rightarrow (X \geq ½)$.*

Proof First, if $X = 1$ then according to the definition of information ordering, $(1 = ½) \vee (Y = 1)$ thus $Y = 1$.

Now if $Y = 1$ then similarly $(X = ½) \vee (X = 1)$ thus $X \geq ½$. ■

Selected mathematical operations respect the information ordering:

Lemma 2 (Information order vs Mathematical operations) *If $X_1 \sqsubseteq Y_1, \ldots, X_n \sqsubseteq Y_n$ then*

[1] $1 - X_1 \sqsubseteq 1 - Y_1$

[2] $min\{X_1, \ldots, X_n\} \sqsubseteq min\{Y_1, \ldots, Y_n\}$

[3] $max\{X_1, \ldots, X_n\} \sqsubseteq max\{Y_1, \ldots, Y_n\}$

Proof

[1] Since $X_1 \sqsubseteq Y_1$ then either $X_1 = Y_1$ or $X_1 = ½$. If $X_1 = Y_1$, then $1 - X_1 = 1 - Y_1$ and therefore $1 - X_1 \sqsubseteq 1 - Y_1$ is true. Otherwise, if $X_1 = ½$, then $1 - X_1 = ½$ and $½ \sqsubseteq Y_1$ holds for any Y_1.

[2] If some $X_i = 0$ then $Y_i = 0$. Thus $min\{X_1, \ldots, X_n\} = 0$ and $min\{Y_1, \ldots, Y_n\} = 0$, and $0 \sqsubseteq 0$ holds. Otherwise, if all $X_i = 1$ then all $Y_i = 1$, therefore $min\{X_1, \ldots, X_n\} = min\{Y_1, \ldots, Y_n\} = 1$, and $1 \sqsubseteq 1$ is satisfied. Finally, if there is no X_i with $X_i = 0$ but some $X_j = ½$ then $min\{X_1, \ldots, X_n\} = ½$, and $½ \sqsubseteq min\{Y_1, \ldots, Y_n\}$ holds for any $Y_1, \ldots, Y_n$ values.

[3] If there is an $X_i = 1$, then $Y_i = 1$. Thus $max\{X_1, \ldots, X_n\} = 1$ and $max\{Y_1, \ldots, Y_n\} = 1$, and $1 \sqsubseteq 1$ holds. Otherwise, if all $X_i = 0$ then all $Y_i = 0$, therefore $max\{X_1, \ldots, X_n\} = max\{Y_1, \ldots, Y_n\} = 0$, and $0 \sqsubseteq 0$ is satisfied. Finally, if there is no X_i with $X_i = 1$, but some $X_j = ½$ then $max\{X_1, \ldots, X_n\} = ½$, and $½ \sqsubseteq max\{Y_1, \ldots, Y_n\}$ holds for any $Y_1, \ldots, Y_n$ values.

■

Our the refinement relation respects information ordering for each formula φ.

Theorem 13 (Approximation) *Let P, Q be partial models with $P \sqsubseteq Q$ and φ be a graph pattern.*

- *If $[\![\varphi]\!]^P = 1$ then $[\![\varphi]\!]^Q = 1$; if $[\![\varphi]\!]^P = 0$ then $[\![\varphi]\!]^Q = 0$ (called **under-approximation**).*
- *If $[\![\varphi]\!]^Q = 0$ then $[\![\varphi]\!]^P \leq ½$; if $[\![\varphi]\!]^Q = 1$ then $[\![\varphi]\!]^P \geq ½$ (called **over-approximation**).*

Proof Correctness of under- and over-approximation Let φ be a graph pattern formula, and let P and Q be two partial models where $P \sqsubseteq Q$ with a refinement function $ref : O_P \to 2^{O_Q}$.

First, based on the definition of refinement, for each $p_1, p_2 \in O_P$ and $q_1 \in ref(p_1)$, $q_2 \in ref(p_2)$, the following statements hold for atomic predicates:

- $[\![\mathsf{C}(v)]\!]^P_{v \mapsto p_1} \sqsubseteq [\![\mathsf{C}(v)]\!]^Q_{v \mapsto q_1}$

- $[\![\mathsf{R}(v_1, v_2)]\!]^P_{v_1 \mapsto p_1, v_2 \mapsto p_2} \sqsubseteq [\![\mathsf{R}(v_1, v_2)]\!]^Q_{v_1 \mapsto q_1, v_2 \mapsto q_2}$
- $[\![v_1 \sim v_2]\!]^P_{v_1 \mapsto p_1, v_2 \mapsto p_2} \sqsubseteq [\![v_1 \sim v_2]\!]^Q_{v_1 \mapsto q_1, v_2 \mapsto q_2}$

Next, let φ_1 and φ_2 be two formulae, and let Z_1^P, Z_2^P, Z_1^Q and Z_2^Q be variable bindings with:

- $[\![\varphi_1]\!]^P_{Z_1^P} \sqsubseteq [\![\varphi_1]\!]^Q_{Z_1^Q}$ and $[\![\varphi_2]\!]^P_{Z_2^P} \sqsubseteq [\![\varphi_2]\!]^Q_{Z_2^Q}$,
- Z_1^P and Z_1^Q maps each variables of φ_1 to $\mathcal{O}_P$ and $\mathcal{O}_Q$
- Z_2^P and Z_2^Q maps each variables of φ_2 to $\mathcal{O}_P$ and $\mathcal{O}_Q$
- for all variables v in φ_1: $Z_1^Q(v) \in \mathit{ref}(Z_1^P(v))$
- for all variables v in φ_2: $Z_2^Q(v) \in \mathit{ref}(Z_2^P(v))$

Then the following refinements of formulae hold due to Lemma 2:

- $[\![\neg\varphi_1]\!]^P_{Z_1^P} = 1 - [\![\varphi_1]\!]^P_{Z_1^P} \sqsubseteq 1 - [\![\varphi_1]\!]^Q_{Z_1^Q} = [\![\neg\varphi_1]\!]^Q_{Z_1^Q}$
- $[\![\varphi_1 \wedge \varphi_2]\!]^P_{Z_1^P \cup Z_1^P} = \mathit{min}\{[\![\varphi_1]\!]^P_{Z_1^P}, [\![\varphi_2]\!]^P_{Z_2^P}\} \sqsubseteq \mathit{min}\{[\![\varphi_1]\!]^Q_{Z_1^Q}, [\![\varphi_2]\!]^Q_{Z_2^Q}\} = [\![\varphi_1 \wedge \varphi_2]\!]^Q_{Z_1^Q \cup Z_2^Q}$
- $[\![\varphi_1 \vee \varphi_2]\!]^P_{Z_1^P \cup Z_1^P} = \mathit{max}\{[\![\varphi_1]\!]^P_{Z_1^P}, [\![\varphi_2]\!]^P_{Z_2^P}\} \sqsubseteq \mathit{max}\{[\![\varphi_1]\!]^Q_{Z_1^Q}, [\![\varphi_2]\!]^Q_{Z_2^Q}\} = [\![\varphi_1 \vee \varphi_2]\!]^Q_{Z_1^Q \cup Z_2^Q}$
- $[\![\exists v : \varphi_1]\!]^P_{Z_1^P} = \mathit{max}\{[\![\varepsilon(v) \wedge \varphi]\!]^P_{Z_1^P, v \mapsto p} : p \in \mathcal{O}_P\} \sqsubseteq$
  $\sqsubseteq \mathit{max}\{[\![\varepsilon(v) \wedge \varphi]\!]^Q_{Z_1^Q \cup v \mapsto q} : q \in \mathit{ref}(p)\} = [\![\exists v : \varphi_1]\!]^Q_{Z_1^Q}$
- $[\![\forall v : \varphi_1]\!]^P_{Z_1^P} = \mathit{min}\{[\![\varepsilon(v) \vee \varphi]\!]^P_{Z_1^P, v \mapsto p} : p \in \mathcal{O}_P\} \sqsubseteq$
  $\sqsubseteq \mathit{min}\{[\![\varepsilon(v) \vee \varphi]\!]^Q_{Z_1^Q \cup v \mapsto q} : q \in \mathit{ref}(p)\} = [\![\forall v : \varphi_1]\!]^Q_{Z_1^Q}$

Since all these refinement relations hold, the statement of the theorem is now a direct consequence of Lemma 1. ■

A.4 Refinement operations

Theorem 14 (Refinement operations ensure refinement) *Let P be a partial model and op be a refinement operation. If Q is the partial model obtained by executing op on P (formally, $P \xrightarrow{op} Q$) then $P \sqsubseteq Q$.*

Proof We split the proof cases along the refinement operations. We investigate changes in the truth evaluation of different predicates implied by executing these operations, since each partial model is a refinement of itself if no changes occur.

- In case of *concretize*(p, val):
  - For each class predicate $p = \mathsf{C}_i(o)$, only operation *concretize*(p, val) can potentially change its value to 1 (or 0) if $[\![\mathsf{C}_i(o)]\!]^P = ½$. But then $[\![\mathsf{C}(o)]\!]^P = ½ \sqsubseteq [\![\mathsf{C}(o)]\!]^Q = 1$ (or $[\![\mathsf{C}(o)]\!]^Q = 0$), which satisfies the refinement relation.
  - Reasoning is identical for each reference predicate $\mathsf{R}(o_1, o_2)$.
  - An equivalence predicate $o_1 \sim o_2$ can be manipulated by operation *concretize*(p, val) to set an ½ value to 1 (for self-loop equivalence predicates) or to either 1 or 0 (for non-self loops). In this case, the refinement conditions are trivally satisfied.
- When *splitAndConnect*$(o, mode)$ is applied then two o_1 and o_2 nodes of Q will be derived from a single node o in P.
  - *At-least-two mode*:
    Since $[\![o \sim o]\!]^P = ½$ and both $[\![o_1 \sim o_1]\!]^Q = ½$ and $[\![o_2 \sim o_2]\!]^Q = ½$, but $[\![o_1 \sim o_2]\!]^Q = 0$, the refinement condition is satisfied.
  - *At-most-two mode*:
    Since $[\![o \sim o]\!]^P = ½$ and both $[\![o_1 \sim o_1]\!]^Q = 1$ and $[\![o_2 \sim o_2]\!]^Q = 1$ while $[\![o_1 \sim o_2]\!]^Q = ½$, the refinement condition is satisfied.

■

Corollary 5 (Open derivations does not lead to 1) *Let $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ be an open derivation sequence of refinement operations wrt. φ. Then for each $0 \leq i \leq k$, $[\![\varphi]\!]^{P_i} \leq ½$.*

Proof This is a direct consequence of Theorem 13. If we indirectly assume that $[\![\varphi]\!]^{P_k} \leq ½$ but $[\![\varphi]\!]^{P_i} = 1$ for some P_i along the derivation sequence, then all subsequent partial models P_j derived from P_i $(j > i)$ should be $[\![\varphi]\!]^{P_i} = 1$ which contradicts our assumption for $j = k$. ■

Corollary 6 (Soundness of model generation) *Let $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ be a finite and open derivation sequence of refinement operations wrt. φ. If P_k is a concrete instance model M (i.e. $P_k = M$) then M is consistent (i.e. $[\![\varphi]\!]^M = 0$).*

Proof We require that $[\![\varphi]\!]^{P_i} \leq ½$ for each i which includes the last partial model P_k. Since P_k is a concrete instance model, thus the 2-valued and 3-valued evaluation of φ must be identical. Therefore $[\![\varphi]\!]^M = 1$ or $[\![\varphi]\!]^M = 0$, but only the latter case satisfies our assumption that $[\![\varphi]\!]^{P_k} \leq ½$. ■

Theorem 15 (Finiteness of model generation) *For any finite instance model M, there exists a finite derivation sequence $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ of refinement operations starting from the most generic partial model P_0 leading to $P_k = M$.*

Proof Sketch
An instance model can always be generated:

[1] Assume that M contains exactly n objects. Since P_0 consists of a single object, we need to create $n-1$ new objects as part of the construction.

[2] Execute action *splitAndConnect*$(o, mode)$ in *at-least-two* mode for $n-1$ times, thus n (uncertain) objects will be available.

[3] Concretize all $[\![o \sim o]\!]^{P_{n-1}} = 1$ and $[\![o_1 \sim o_2]\!]^{P_{n-1}} = 0$ (where $o_1 \neq o_2$).

[4] Concretize all class and reference predicates in accordance with M by setting appropriate values in *concretize*(p, val) to 1 or 0. As a result, P_{n-1} is gradually refined into a P_k which no longer contains an ½ value, thus it is an instance model.

Model generation is always finite:

[1] First, note that only *splitAndConnect*$(o, mode)$ actions are able to create new objects, *concretize*(p, val) operations only fix values. Moreover, there are only finite number of uncertain values of p which still needs to be concretized.

[2] The only recursive (thus potentially infinite) computation is carried out when action *splitAndConnect*$(o, mode)$ is executed in *at-least-two* mode.

[3] Assume that in our computation, *splitAndConnect*$(o, mode)$ has been applied in *at-least-two* mode n times, thus P_n contains at least $n+1$ objects, while our instance model has only n objects. We claim that this is a dead end derivation, thus we can cut it off and backtrack.

[4] Due to the specification of the *at-least-two* model, all these objects are non-equivalent to each other, i.e. $[\![o_1 \sim o_2]\!]^{P_n} = 0$ for $o_1 \neq o_2$, thus they can never be merged during concretization. Now any consistent concretization of P_n will contain at least $n+1$ different objects, which contradicts our indirect assumption that M has exactly n objects.

■

Theorem 16 (Completeness of model generation) *For any finite and consistent instance model M with $[\![\varphi]\!]^M = 0$, there exists a finite open derivation sequence $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ of refinement operations wrt. φ starting from the most generic partial model P_0 and leading to $P_k = M$.*

Proof First, M is derivable by a finite derivation sequence due to Corollary 8. Now, for an indirect proof, let us assume that $[\![\varphi]\!]^M = 0$ yet there exist some partial model P_i along the finite derivation sequence $P_0 \xrightarrow{op_1;\ldots;op_i} P_i \xrightarrow{op_{i+1};\ldots;op_k} P_k$ where $[\![\varphi]\!]^{P_i} = 1$. However, the properties of under-approximation (in Theorem 13) imply that for all refinements P_j of P_i, $[\![\varphi]\!]^{P_j} = 1$. But since M is also a refinement of P_j (as each refinement operation ensures refinement, see Theorem 7), $[\![\varphi]\!]^M = 1$, which is a contradiction to our indirect assumption, thus it concludes the proof. ■

Theorem 17 (Decidability of model generation in finite scope) *Given a graph predicate φ and a scope $n \in \mathbb{N}$, it is decidable to check if a concrete instance model M exists with $|O_M| \leq n$ where $[\![\varphi]\!]^M = 0$.*

Proof (Sketch) While Theorem 9 ensures that there exists one finite derivation path, this does not directly guarantee that model generation would terminate along all derivation paths. Fortunately, the designated target scope n for the instance model implies an upper bound (i.e. scope) for the length of operation sequences that derive instance models of size n.

For any model M with n nodes and r edges, one can derive an operation sequences with n *splitAndConnect* operations followed by $r \cdot n^2$ *concretize* operations. Our refinement operations ensure that any derivation longer than $n + r \cdot n^2$ can be terminated as even smallest concrete instance model will exceed the target model scope n.

■

Corollary 7 (Incrementality of model generation) *Let us assume that no consistent models M^n exist for scope n, but there exists a larger consistent model M^m of size m (where $m > n$) with $[\![\varphi]\!]^{M^m} = 0$. Then M^m is derivable by a finite derivation sequence $P_i^n \xrightarrow{op_{i+1};\ldots;op_k} P_k^m$ where $P_k^m = M^m$ starting from a partial model P_i^n of size n.*

Proof As an indirect proof, let us assume that there exists a consistent model M^m of size m while there are no consistent models M^n up to scope n, but no derivation sequence $P_i^n \xrightarrow{op_{i+1};\ldots;op_k} P_k^m$ exists which would yield $M^m = P_k^m$ starting from a partial model P_i^n of size n.

Since M^m is consistent and finite, it is derivable thanks to the completeness theorem (Theorem 9) along some other derivation sequence $P_0 \xrightarrow{op_1;\ldots;op_l} P_k^m$ where $P_k^m = M^m$. Since each refinement operation used in $op_1; \ldots; op_l$ increases the size of P_i with at least one, the derivation sequence should reach a partial model P_j^n of size n.

With the trivial concretization (of turning all ½ values to 1 for all class and reference predicates and to 0 for equivalence predicates), P_j^n can be turned into an instance model M_j^n which is also exactly of size n. Now if M_j^n is consistent, then our assumption is violated that no consistent models exist for scope n Otherwise, the tail of $P_j^n \xrightarrow{op_j;\ldots;op_l} P_k^m$ is a designated derivation sequence, which is a contradiction to our indirect assumption. ■

Corollary 8 (Completeness of refutation) *If all derivation sequences are closed for a given scope n, but no consistent model M^n exists for scope n for which $[\![\varphi]\!]^{M^n} = 0$, then no consistent models exist at all.*

Proof As an indirect proof, let us assume that a consistent model M^m exists for some scope $m > n$, while all derivation sequences are closed for a given scope n and no consistent models M^n exist for that scope.

Since M^m is consistent and finite, then there shall be a derivation sequence $P_0 \xrightarrow{op_1;\ldots;op_m} P_m$ where $P^m = M^m$. However, all derivation sequences are closed for a given scope n, which holds for the prefix of this derivation sequence as well. Thus there shall be an intermediate partial model P_k along that sequence where (1) either no further refinement operations are executable or (2) φ has a match in P_k i.e. $[\![\varphi]\!]^{P_k} = 1$. In the former case, P^m would not be reachable by refinement operations. In the latter case, all refinements of P_k (including $P^m = M^m$) would have a match of φ due to Theorem 13. This is a contradiction which concludes our proof. ■

A.5 Change partitioning of view models

A.5.1 Affected parts of view model changes

Table A.3 contains the calculation of affected parts in case of modifying a view model M_V. When a view object v is removed, the affected parts are determined by its defining pattern φ and its activation Z that is stored in the trace model T. When a reference ref is removed from M_V, the affected part of the source v_s and the target v_t are returned. When a new view element is created, then there are obviously no activations of forward rules.

A.5.2 Affected parts of pattern activations

A pattern predicate φ with its activation Z marks the union of the bodies defined in Table A.4. Each body consists of several conditions *const* that may introduce additional internal variables $Params_i$. Hence, an activation of a *body* is extended with Z_i all the possible bindings of internal variables. The affected part of a body *body* is the union of the affected parts of each constraints *const*.

A.5.3 Affected parts of source constraints

Affected parts of source constraints are defined in Table A.5. A *class* (class) condition returns the object that is bound to its parameter along activation Z. Similarly, *attribute*(attr) and *reference*(ref) conditions (together feature conditions *feature*(feat) select respective parameters (x and both x, y) from Z. A *path*($\text{feat}_1 \ldots \text{feat}_n$) condition can be split into several *feature*(feat) to calculate its affected part. For *equal*(=) and *not equal*(≠), the bound objects of parameters from both side of the operators are returned.

A pattern φ may call another pattern φ' (*find*$[\varphi']$ or transitively with *find*+$[\varphi']$) by mapping symbolic parameters $Params_{p'}$ of the called pattern to concrete values of the caller. Thus given a binding : $Params_{p'} \rightarrow Params_p$, a match Z' is composed from Z by getting objects from the original activation Z, formally:

$$Z' \circ Z \Rightarrow Z' : Z(\text{binding}(var')) \rightarrow O_{M_S}, var' \in Params'_{p'}$$

However, a *negative application condition* (*neg find*$[p']$) is separated into two cases: if the sub predicate φ' does not introduce any internal variable, the affected part returns the referenced objects from the activation Z. Otherwise, we restrict the affected part to all the objects that has the same type as the introduced internal variables have.

A.5.4 Categorization of affected source model objects

The affected objects of the source model can be categorized into three groups:

- M_S^F : *neither changeable nor removable*: It includes all objects of M_S which are not in the affected part of the change from the view Δ_{view}.

$$M_S^F = M_S - \text{affected}(\Delta_{view})$$

- M_S^C : *changeable but non-removable objects*: It includes all objects in the affected part of the change from the view Δ_{view} which are responsible for the existence of other activations.

$$M_S^C = \text{affected}(\Delta_{view}) - \{o | o \text{ referred by } T_{obj}\}$$

$$\begin{array}{c} -\text{class}(v)\rightarrow\text{affected}(\varphi(Z)) : lookup_{VS}() = \varphi(Z) \\ \hline -\text{ref}(v_1, v_2)\rightarrow\text{affected}(\varphi(Z)) : lookup_{T}(v_1, v_2) = \varphi(Z) \\ \hline -\text{attr}(v, val)\rightarrow\text{affected}(\varphi(Z)) : lookup_{T}(v_1, v_2) = \varphi(Z) \end{array}$$

Table A.3: Affected changes of view model

$$\begin{array}{rl} \varphi(Z) & \rightarrow \bigcup \text{affected}(body_i, Z) \\ \hline body(Z) & \rightarrow \bigcup \text{affected}(cond_i(Z + Z_i)) \end{array}$$

Table A.4: Affected activations

$$\begin{array}{r} \text{class}[obj], (Z)\rightarrow\{o_{obj}|Z(obj) = o_{obj}\} \\ \hline \text{attr}[x, val], (Z)\rightarrow\{o_x|Z(x) = o_x\} \\ \hline \text{ref}[x, y], (Z)\rightarrow\{o_x, o_y|Z(x) = o_x, Z(y) = o_y\} \\ \hline \text{feat}(m)\rightarrow\begin{cases}\text{affected}(\text{attr}(m)) \text{ if feat is attr} \\ \text{affected}(\text{ref}(m)) \text{ if feat is ref}\end{cases} \\ \hline \text{feat}_1 \ldots \text{feat}_n[x, y](m)\rightarrow\bigcup \text{affected}(\text{feat}_i(m)) \\ \hline \begin{array}{r} x = y, (Z) \\ x \neq y, (Z) \end{array}\rightarrow\{o_x, o_y|Z(x) = o_x, Z(y) = o_y\} \\ \hline find[p'](m)\rightarrow\text{affected}(p'(m')), Z' \circ Z \\ \hline find^+[p'](m)\rightarrow\bigcup \text{affected}(p'(m')), Z' \circ Z \\ \hline neg\ find[p'](m)\rightarrow\begin{cases}\{o_x|m(x) = o_x,\}, \text{ if no inner var} \\ \{o_i|o_i.type \in \text{inner types of } p'\}\end{cases} \end{array}$$

Table A.5: Affected changes of source constraints

- M_S^O : *changeable and removable objects*: All objects in the affected part of the change from the view Δ_{view} which are not responsible for the existence of any other activations.

$$M_S^O = \text{affected}(\Delta_{view}) - M_S^C$$