SEARCH-BASED TECHNIQUES IN MODEL-DRIVEN ENGINEERING

PhD Thesis

Ákos Horváth
MSc in Technical Informatics

Supervisor:
Dr. Dániel Varró, PhD
associate professor

Budapest, February 2013
Nyilatkozat önálló munkáról, hivatkozások átvételéről

Alulírott Horváth Ákos 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 forrákból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Declaration of own work and references

I, Ákos Horváth, hereby declare, that this thesis, 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.

Budapest, 2013. 02. 19.

Ákos Horváth
Summary

The growing complexity of system development demands for the widespread use of design automation in order to continuously improve productivity, achieve better quality and lower maintenance and documentation costs.

Model-Driven Engineering (MDE) has recently become a key technique in systems engineering with the goal to provide these automations. MDE facilitates the systematic use of models and implies that models play a central role that encompasses the entire system development lifecycle. These models can be used in design automation for documentation, early model-based validation and generative development. Common in these automation techniques that they are heavily relying on model transformation as it provides their foundations of effective manipulation and querying of models.

Usually in early phases of designing complex systems models are not sufficiently detailed to serve as an input for design automation approaches. Instead, a design space is constituted by multiple models representing different valid design candidates. Design Space Exploration (DSE) aims at searching and constructing such candidates defined in the design space which satisfy all structural and numeric design constraints based on various quality metrics such as performance, cost, power, reliability, etc.

Unfortunately, despite the significant investment of research and development into the application of MDE for the development of complex system there have been very few methods directly aiming to define automated model-based techniques that can solve complex structural constraints over the system design already captured by a set of models. Moreover, as complexity of systems are rising, scaling model transformations capable of handling models ranging in the millions of elements is becoming a crucial problem for supporting such advanced MDE based techniques.

Therefore, this thesis is centered around the concept of providing novel search-based techniques in model-driven engineering for both (i) high-level design automation based on design space exploration for solving complex structural constraints and (ii) scalable model transformation combining different model querying approaches for effective graph pattern matching.

I propose a general framework for uniformly representing arbitrary search operations by expressing them as cost-weighted predicates that allows to uniformly guide the graph pattern matching process. Built on top of this general representation, I propose a hybrid pattern matching approach which enables the transformation designer to combine different graph pattern matching approaches to adapt to memory and performance constraints.

I present a model-driven design space exploration technique defined as an extension to finite-domain constraint satisfaction problems by using graph patterns to define structural (first-order logic) constraints, and graph transformation rules as labeling operations. However, instead of simple variable substitution, the labeling phase applies graph transformation rules to carry out model manipulations on the underlying graph domain.

As a motivating example in the context of the DIANA EU FP6 project for practical application of the contributions of the thesis, I designed and participated in the development of a complete MDE based development framework for systematically designing configuration tables for civil avionics real-time operating systems. My research in the context of this work was focused on three topics: (i) adaptation of MDE techniques for defining a specific development process for avionics configuration design that is in-line with the DO-178B certification requirements (ii) definition and reuse of model-based validation techniques for early error detection and localization and (iii) implementation of end-to-end traceability from high-level models to generated artifacts through the complete process that conforms to DO-178B.
Összefoglaló

A rendszerek komplexitásának növekedésével azonos ütemben egyre széles körben kezdenek elterjedni a különféle tervezésautomatizálási módszerek, melyek célja a termelékenység és a minőség növelésével párhuzamosan a karbantartási és a dokumentációs költségek csökkentése.

A modellvezérelt fejlesztés (MDE) egyik kitűzött célja ezen automatizálási módszerek hatékony megvalósítása. Az MDE szerint minden, a tervezésautomatizálás szempontjából releváns információt absztraktnének kezelünk a modelleket tárolva. E modellek segítségével automatizálható (i) a tervezés alatt álló rendszer struktúrált dokumenáltása, (ii) a valós implementáció megfelelő modell alapú validációja és (iii) a cél platform elemeinek automatikus generálása. Közös ezekben az automatizálási módszerekben, hogy legnagyobb részben modelltranszformációkra támaszkodnak a modellek hatékony lekérdezésére és manipulációjára.

Egy MDE folyamat kezdetén a modell jellemzően még nem elég specifikusak az automatikus kódgeneráláshoz, ezért az automatizálási módszerek hatékony megvalósítása szükséges. Ezért a kifeszített térében megtalálható modell között keresünk egy olyan megoldást (design space exploration DSE), amely teljesíti a rendszerhez szemben támasztott követelményeket és kielégítő megoldást ad az összes minőség-mértékre (teljesítmény, megbízhatóság, stb.).

Azonban annak ellenére, hogy igen széleskörű kutatásokat végeztek komplex rendszerek modellezésével kapcsolatban, kevesen foglalkoztak specifikus módszerekkel realizált komplex struktúrális kényszerek automatikus kielégítésével. Másfelől, az ilyen modellvezérelt technikákra épülő alkalmazások esetén kulcsfontosságú a modelltranszformációk skálázódása a több százezres vagy akár milliósszámú modellméretekre.

Az értekezés fő célkitűzése, hogy olyan újszerű, keresés alapú módszereket dolgozzon ki a modellezéssel fejlesztés szerint kapcsolatban, kevesen foglalkoztak specifikusan modell elektromos definiált komplex struktúrális kényszerek automatikus kielégítésével. Másfelől, az ilyen modellvezérlés technikákra épülő alkalmazások esetén kulcsfontosságú a modelltranszformációk skálázódása a több százezres vagy akár milliósszámú modellméretekre.

Az értekezés eredményeinek gyakorlati alkalmazására az Európai Unió által finanszírozott, DIANA kutatási program keretében megtervezve és megvalósítottam egy komplex modellvezérelt fejlesztési paradigmát Az értékepeket az MDE alapú modellezés és modelltranszformáció relatív hatékony és hatékony módszerek kialakítása miatt hatékony megvalósítására.

Kidolgoztam egy keretrendszer a különböző mintaillesztési műveletek egységes kezelésére, amelyekkel (i) hatékonyan lehet kezelni a tervezési tér felderítési problémákat, (ii) kényelmesen megvalósítható komplex struktúrális kényszerek automatikus kielégítésével. Másfelől, az ilyen modellvezérlés technikákra épülő alkalmazások esetén kulcsfontosságú a modelltranszformációk skálázódása a több százezres vagy akár milliósszámú modellméretekre.

Az értekezés eredményeinek gyakorlati alkalmazására az Európai Unió által finanszírozott, DIANA kutatási program keretében megtervezve és megvalósítottam egy komplex modellvezérelt fejlesztési paradigmát Az értékepeket az MDE alapú modellezés és modelltranszformáció relatív hatékony és hatékony módszerek kialakítása miatt hatékony megvalósítására.

Kidolgoztam egy keretrendszer a különböző mintaillesztési műveletek egységes kezelésére, amelyekkel (i) hatékonyan lehet kezelni a tervezési tér felderítési problémákat, (ii) kényelmesen megvalósítható komplex struktúrális kényszerek automatikus kielégítésével. Másfelől, az ilyen modellvezérlés technikákra épülő alkalmazások esetén kulcsfontosságú a modelltranszformációk skálázódása a több százezres vagy akár milliósszámú modellméretekre.

Kidolgoztam egy keretrendszer a különböző mintaillesztési műveletek egységes kezelésére, amelyekkel (i) hatékonyan lehet kezelni a tervezési tér felderítési problémákat, (ii) kényelmesen megvalósítható komplex struktúrális kényszerek automatikus kielégítésével. Másfelől, az ilyen modellvezérlés technikákra épülő alkalmazások esetén kulcsfontosságú a modelltranszformációk skálázódása a több százezres vagy akár milliósszámú modellméretekre.

Kidolgoztam egy keretrendszer a különböző mintaillesztési műveletek egységes kezelésére, amelyekkel (i) hatékonyan lehet kezelni a tervezési tér felderítési problémákat, (ii) kényelmesen megvalósítható komplex struktúrális kényszerek automatikus kielégítésével. Másfelől, az ilyen modellvezérlés technikákra épülő alkalmazások esetén kulcsfontosságú a modelltranszformációk skálázódása a több százezres vagy akár milliósszámú modellméretekre.
Acknowledgements

I wish to express my gratitude, first and foremost, to my supervisor, Dr. Dániel Varró. Without his continuous support, ideas, enormous patience and valuable insight over the past seven years, I would have never succeeded.

I am also thankful to my colleagues in the Fault Tolerant Systems and Research Group, especially Prof. Dr. András Pataricza and Dr. István Majzik for their trust, friendly advices and support for my research. I am also very grateful to my close colleagues who cooperated in the development of the VIATRA2 and EMF-IncQuery projects, namely Dr. István Ráth, Gábor Bergmann, Ábel Hegedüs, Zoltán Ujhelyi, Zoltán Balogh, András Ökrös and Dr. Gergely Varró for their kind support and productive collaborative research work, and all the fun we have together over these years. Additionally, many thanks goes to research associates Máté Kovács, Dr. Balázs Polgár, Dr. András Balogh, Péter Bokor, Benedek Izzó, András Vörös, Zoltán Micskei, Zoltán Szatmári, András Kövi, Imre Kocsis, László Gönczy, Áron Sisak, István Szombath, János Oláh, Gergely Paljak, Dániel Tóth and my former MSc student Dénes Monostori. Moreover, I wish to thank all the project partners I worked together as they provided me with many good ideas and problems to work on, namely Tobias Schoofs, Eric Jenn, Ivo Viglietti, Klaas Wiegmink, Marc Richard-Foy, Damien Carbonne, Dr. Christian Engel, Zoltán Mátisa Gábor Gregus, Rodrigo Rizzi Starr, Andrea Barp, Márcio Bento Viventini, José Ricardo Parizi Negrão and all the others for their suggestions, ideas, long discussions and hard work that formed my thesis work.

The current work was partially supported by the Hungarian CERTIMOT (ERC_HU-09-01-2010-0003) project, the European Union DIANA (AERO1-030985), SecureChange (ICT-FET-231101) and SENSORIA (IST-3-016004) research projects and a collaborative project with Embraer. Additionally, this work is connected to the scientific program of the "Development of quality-oriented and cooperative R+D+I strategy and functional model at BME" project. Finally, the thesis is also supported by the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

Finally, I get to those people for acknowledgement who did the most to establish a stable background for me, and gave encouragement and support and did not let me to surrender even if it seemed very far to finish my thesis: my parents Zsuzsanna and László, my brother Bálint and finally, Linda.
## Contents

1 Introduction ................................................. 1
   1.1 Model-Driven Engineering ................................. 1
   1.2 Design Space Exploration ................................. 2
   1.3 Model Transformation .................................... 2
   1.4 A Motivating Application Domain ......................... 3
   1.5 Challenges and Contributions ............................. 3
   1.6 The Structure of the Thesis ............................... 7

1 Model-Driven Engineering ..................................... 9

2 Modeling Preliminaries ....................................... 11
   2.1 Running Example: AntWorld ................................. 11
   2.2 Overview of Models and Metamodels ......................... 13
   2.3 Modeling Environments .................................... 14
   2.4 Concepts of Metamodels and Instance Models ............... 17
   2.5 Summary ................................................. 19

3 Model Transformation ...................................... 21
   3.1 Modeling the AntWorld case study ......................... 21
   3.2 Formalizing Model Transformation as Graph Transformation 27
   3.3 Concepts of Graph Patterns and Graph Transformation .......... 28
   3.4 Summary ................................................ 33
# CONTENTS

## II Advances in Graph Pattern Matching

### 4 Introduction

- 4.1 Motivation: Hard Coded Heuristics for Local Search Based Pattern Matching
- 4.2 Motivation: Combining Different Pattern Matching Approaches

### 5 Search Plan Driven Graph Pattern Matching

- 5.1 Search Graphs
- 5.2 Adorned Search Graphs and Search Plans
- 5.3 Search Plan Generation
- 5.4 Realization of an Adorned Search Graph Driven Pattern Matcher in Viatra2
- 5.5 Compile Time Steps
- 5.6 Runtime Steps
- 5.7 Summary

### 6 Hybrid Graph Pattern Matching

- 6.1 Incremental Graph Pattern Matching
- 6.2 Combining Pattern Matching Strategies
- 6.3 Benchmark Evaluation on the AntWorld Case Study
- 6.4 Towards Intelligent Selection of Matching Strategies
- 6.5 Summary

### 7 Summary and Related Work in Graph Pattern Matching

- 7.1 Related Work
- 7.2 Summary

## III Design Space Exploration

### 8 Design Space Exploration

- 8.1 Introduction
- 8.2 Motivation
- 8.3 Constraint Satisfaction Programming
- 8.4 Flexible and Dynamic Constraint Satisfaction Problems over Models
- 8.5 Optimization Strategies and Implementation Details
- 8.6 Evaluation
- 8.7 Related Work
- 8.8 Summary

## IV Model-Driven Engineering for Avionics Systems

### 9 Introduction to Aeronautics

- 9.1 Integrated Modular Avionics
- 9.2 Certification of Airworthiness: Regulatory Bodies and Organizations
- 9.3 Certification of Airworthiness: Processes and Standards in Civil Aeronautics
- 9.4 Summary
Introduction

1.1 Model-Driven Engineering

Models are prime artifacts of engineering. In system development, they have played an important role as a way to capture real world notions as well as abstract constructs. In fact, system architects have been using models and modeling techniques long before model-driven engineering emerged as a trend, e.g. in the form of entity-relationship diagrams, graph-like data structures, abstract syntax trees etc. However, the term Model-Driven Engineering (MDE) [BCW12] implies that models play a central role that encompasses the entire system development lifecycle, starting from requirement analysis to system design, to implementation, to verification, maintenance and even documentation or certification.

Model-driven engineering aims to increase the efficiency and productivity of the system development process by introducing precise engineering practices based on modeling techniques, automated code generation and precise model refinement. It is based on the paradigm that the developer should work from the beginning of the development with high abstraction level models and through well-defined steps the complete process should be automated to the highest possible degree using various MDE techniques.

By this approach, design intelligence is applied to capture all relevant information in the form of abstract models. These models can be used for documentation purposes to store well-structured information about the system-under-design. Models can also be used for early validation, where important properties of the systems (such as performance, robustness, security, complexity) can be evaluated before actual implementation begins. Moreover, models can also serve as source for generative development, where target design artifacts (source code, configuration tables, test cases, textual documentation, etc.) is (semi-)automatically derived by tools. All of these techniques aim at reducing costs and improving modularity and quality.
1.2 Design Space Exploration

In early phases of designing complex systems, models are not sufficiently detailed to serve as an input for automated synthesis tools. Instead, a design space is constituted by multiple models representing different valid design candidates. Design space exploration (DSE) aims at searching and constructing such candidates defined in the design space that satisfy all design constraints.

DSE is a process to quickly obtain feasible, “good enough” solutions which meet all structural and numeric design constraints in order to identify the most suitable design chosen by system architects based on various quality metrics such as performance, cost, power, reliability, etc. Typically, the best solution is flexible in the sense that it provides a trade-off between the optimal solutions with respect to a single quality metrics. Design space exploration is thus a challenging problem in many application areas including the design of critical embedded systems or dynamic reconfiguration of complex IT infrastructures, where MDE techniques have already been quite popular. These problems in an MDE context are frequently addressed as a specific sort of constraint satisfaction problem [Nee01].

Traditionally, most of these constraints and quality attributes were numeric in nature for expressing time, throughput, budget, memory limits, etc. However, the birth of modular software architectures in critical systems (like AUTOSAR [AUT] in the automotive domain or Integrated Modular Avionics (IMA) [RTCd] in the aeronautical domain) introduced complex structural constraints, which express connectivity restrictions for the graph-based model of the system under design. In addition, in many practical scenarios (like IT systems management), design space exploration is further complicated by the continuous evolution of the system, which imposes further constraints and quality metrics.

1.3 Model Transformation

Model transformation is considered as the backbone of MDE. It aims to carry out automated translation within and between modeling languages to propagate information along the complete development process on all abstraction levels. As MDE is becoming a dominant development method in several novel application domains from automotive design to service management, it urges the need for fast execution of general model transformations handling models up to the size of ten millions of elements as only dedicated solutions exit e.g., in the reverse engineering domain [NFB11].

Additionally, as a trend in state-of-the-art model-driven engineering, modeling tools rely on a variety of execution scenarios applied in different application domains such (i) as contract based early validation over models in the safety critical domain [BHCGR09], (ii) as a synchronization technique between modeling languages [ROV10, GW09], (iii) as an abstract monitoring layer for complex systems [SWWM10] and (iv) as a constraint solving engine for complex structural constraints [SDB10] in design space exploration problems for embedded system design.

Over the years a large variety of different tool emerged (e.g., Epsilon [RPKP08], ATL [ATL], etc.) using different concepts and techniques. Among those, one of the most popular approach is the declarative rule-based graph transformation paradigm (e.g., VMTS [VS], ATOM3 [dLV02], MOFLON [AKRS06], AGG [Tae00], GROOVE [Ren04a], FUJABA [NNZ00] or VIATRA2 [Via]).

Graph transformation (GT) provides a declarative language for defining the manipulation of graph models by means of GT rules. A GT rule consists of (i) a left-hand side (LHS), (ii) a right-hand side (RHS) graph. Model manipulations are carried out by replacing a match of the LHS in the model by an image of the RHS. This is performed in two phases. In the performance critical pattern
matching phase, matches of the LHS are searched in the underlying model. In the updating phase, the selected matching parts are modified based on the difference of LHS and RHS.

Tools based on graph transformation already integrate research results of several decades [Zün96, Dör95, MLA10, KKS07] for searching matches of LHSs. In the recent years, the need for fast and effective execution of various model transformations defined by graph transformation rules on models ranging in the millions of elements has become a major challenge due to the widespread use of MDE approaches in different application domains.

1.4 A Motivating Application Domain

MDE for safety critical aeronautical systems As MDE is attracting increasing attention in the aeronautical safety-critical system development [CRH], it needs to be adapted to be in-line with the rigorous DO-178B [RTCe] certification requirements imposed by civil aviation authorities like FAA and EASA. These require (i) V&V activities to be tightly integrated into the development process, (ii) continuous verification activity from early specification to design and development and (iii) end-to-end traceability through the complete development process [Ste]. Moreover, the visited DO-178C[RTCF] certification guidelines has a dedicated subgroup (SWG4) and supplement (DO-333) for Model-Based Development and Verification, which aims to define the specific certification requirements for model-driven engineering approaches based on the already available experience and considerations from DO-178B.

Similar work has already been carried out specifically in the automotive domain and in general for embedded systems:

- The AUTOSAR [AUT] standard explicitly defined model-based development techniques for the semi-automated design of embedded software and hardware [KBFL] for automotive. However, one key difference is that in the automotive domain there is no such strict verification and validation certification process as for the civil avionics systems defined by DO-178B.

- The INDEXYS [IND] project built upon the results of the DECOS [DECB] and GENESYS [GEN] project within the European ARTEMIS framework specified a general architectural principle for embedded system design based on time-triggered architecture. Additionally, it also envisaged an iterative model-based development process that can be adopted to the needs of its application domain.

Although, many of the envisaged techniques within the project are general in principle and can be easily adapted to other model-driven processes, however, due to its general nature it does not take into account the specific requirements of civil avionics system development such as its conservative design methodology and rigid certification principles.

1.5 Challenges and Contributions

My research has been motivated by the practical challenge to adapt model-driven engineering to safety-critical civil avionics systems. I aimed to tackle the development of configuration artifacts for integrated modular avionics systems using a systematic model-driven development process.

Unfortunately, despite the significant investment of research and development into the application of model-driven techniques for the development of embedded software [KSLB03, Bal, CCF] there have been very few methods [KG08] directly aiming configuration development. Lack of techniques in this direction started my research in model-driven design space exploration as certain parts
of the configuration development for civil avionics system required automated techniques that can solve complex structural constraints over the system design already captured by a set of models.

As a prerequisite for applying any advanced MDE based technique for the automated generation of configuration artifacts in the selected avionics domain is the ability to handle huge models in the range of millions of elements. However, at the start of my research in graph pattern matching (2005) model transformation tools were just beginning to scale up to problem sizes of a few ten thousand or hundred thousand model elements, while complex industrial problems were at least an order of magnitude larger. This hindered their use not only for direct, batch model-to-model and model-to-code transformations but also their future application in different execution scenarios such as my model-driven design space exploration approach or model-based validation techniques like on-the-fly design contract evaluation. These applications do not only require fast model transformation executions but also immediate re-evaluation of model queries in case changes occur in their underlying model.

1.5.1 Challenges

Based on these consideration my research challenges are formulated as follows:

- **Challenge 1: How to speed up graph pattern matching to large industrial size problems?**

In order to provide acceptable performance in real-world application scenarios, graph transformation tools apply sophisticated pattern matching algorithms. These are mostly based on two conceptually different approaches: (i) local searches driven by search plans (like FUJABA [NNZ00] or GrGen [JBK10]) and incremental graph pattern matching [JT10, BOR+08] using caching mechanisms to store partial matches.

However, when I investigated existing tools for different implementations [Zün96, GSR05, GBG+06, VVF05, Ren04a, Uni, VVS06a] of both local search and incremental pattern matching, I found that in general, (i) local search based approaches provide a good overall runtime performance on all different execution scenarios [27,21] using relatively low amount of memory compared to the underlying model, while (ii) incremental approaches has shown that in many application scenarios [17] – relevant to my other research directions – they lead to orders-of-magnitude increases in speed, for the price of using increased amount of memory especially when caching large fragments of the underlying model.

Unfortunately, several applications and industrial case studies revealed [16] that available memory can be insufficient (e.g., on restricted virtualized desktop environment) for caching match sets in case of the incremental approach. This problem is especially severe in design space exploration, where the traversed design space also needs to be stored in the memory.

- **Challenge 2: How to support structural constraint solving in evolutionary design space exploration problems?**

Design space exploration problems in an MDE context are mainly tackled as specific sort of constraint satisfaction problem (CSP). However, advanced constraint solvers typically apply certain restrictions for the CSP problem: (i) the domains of variables are required to be (a priori) defined, (ii) the number of variables are also a priori defined and finally, (iii) most approaches disallow the dynamic addition or retraction of constraints [MS00]. Furthermore, mapping graph models obtained in model-driven engineering to variables with finite domain can be a non-trivial task, especially when considering the evolution of models.
1.5. CHALLENGES AND CONTRIBUTIONS

As a summary, existing constraint solvers fail to adequately handle flexible and dynamic structural constraints over graph-like models, which is necessitated for evolutionary design space exploration. Additionally, handling graph models directly in model-driven design space exploration problems necessitate both fast (i) graph pattern matching and (ii) manipulation, which emphasizes challenge 1.

- **Challenge 3: How to support the systematic configuration design of civil avionics systems?**

The ARINC 653 standard [ARIb] has taken a leading role within the aeronautical industry in the development of safety-critical systems based on the Integrated Modular Avionics (IMA) concept. One of the main promises of IMA is saving cost by reducing development, integration and verification effort. In case of ARINC 653 compliant platforms, many deployment and implementation details are defined in configuration tables.

Despite the inherent complexity of avionics systems based on the ARINC 653 platform, current tools supporting configuration design offer very low-level support directly on the XML representation level. Existing tools lack support for (1) capturing the development process for configurations, (2) validating design constraints for configurations on-the-fly and (3) providing traceability between high-level requirements and the configuration tables, which require hand-crafted traceability lists. As a result, finding configuration design flaws early to reduce certification costs is a tedious design task.

1.5.2 Contributions

Figure 1.1 gives an overview on the structure how my different research directions are related to my thesis contributions.

**Contribution 1: A general search plan representation for advanced graph pattern constructs.** I propose a general framework [4,18] for uniformly representing a large variety of search plan operations in graph pattern matching by expressing them as cost-weighted predicates. As an appropriate ordering of these predicates defines an executable search plan, this approach allows to uniformly
guide the pattern matching process for advanced graph patterns regardless of how the actual costs to different search plan operations are assigned.

As a result, the different phases of pattern matching (e.g. cost assignment, generation of search plans, execution of search plans etc.) are fully separated and independent, thus they can be adapted to different graph transformation engines and strategies (metamodel-based vs. model-based search plans [VVF05]). Furthermore, new types of predicates can be introduced easily by assigning appropriate costs without altering the algorithms for search plan generation.

**Contribution 2: Combining local and incremental graph pattern matching algorithms for hybrid strategies.** I propose a hybrid pattern matching approach [16,2], which enables the transformation designer to combine local search based and incremental pattern matching to adapt to memory constraints. My approach is based on the general search plan representation, where incremental pattern matching is introduced as a separate search plan operation. I demonstrate that in certain application scenarios [2] the hybrid approach outperforms the other two approaches and provides a good balance between memory consumption and runtime performance and can support different execution scenarios that are relevant to my other research direction in design space exploration.

However, I notice that selecting the appropriate matching strategy for complex model transformation programs requires a deep understanding of both pattern matching algorithms. Therefore, I examine typical transformation scenarios from the literature [VSV05] [24, 17, 27] and define guidelines [2] for transformation designers to be able to exploit the advantages of the hybrid approach.

**Contribution 3: Defining and implementing a graph transformation based constraint solver for structural constraint problems.** In my research, I aim to tackle [1] evolutionary design space exploration to flexibly identify the most suitable design meeting complex structural constraints and numeric constraints where the underlying constraints may evolve in time, and the evolution of the best design is also restricted by allowed operations and/or quality metrics.

I propose an extension to the definition of constraint satisfaction problems [1, 15] by using graph patterns to define structural (first-order logic) constraints, and graph transformation rules as labeling operations. Informally, all graph pattern constraints need to be satisfied by the underlying model when searching for a specific goal. However, instead of simple variable substitution, the labeling phase applies graph transformation rules to carry out model manipulations on the underlying graph domain. As an analogy, my approach allows to (i) dynamically add/remove constraints from the problem domain, (ii) modify the domain of the variables during search and (iii) define structural constraints in a more natural way.

I demonstrate that model transformation technology can efficiently contribute to formulate and solve certain constraint satisfaction problems with complex structural constraints and dynamic labeling rules. Additionally, apart from the static problem definitions it can also provide a natural way for handling problems in the models@Runtime domain [11]).

**Contribution 4: Model-Driven development of configuration of civil avionics systems.** I designed and participated in the development of the DIANA mapping framework [14] for systematically designing standard ARINC 653 configuration tables in the context of the DIANA EU FP6 project [DIA]. The framework is based on a platform independent architectural modeling language (PIADL) [DECa] that allows integration of industry leading architectural language AADL [SAEb] and system simulation language Matlab Simulink [Mat]. The precise low-level details of a specific configuration for the ARINC 653 platform are captured by a platform specific Integrated Architecture Model (IAM).
1.6. THE STRUCTURE OF THE THESIS

Mapping the PIADL to the IAM is handled by a complex interactive model transformation process that needs to bridge a large abstraction gap where critical design decisions are made by the system architect; thus it cannot be fully automated. Therefore, the mapping process is subdivided into well-defined design steps and precisely defined the contracts, interactions and interfaces of each step. Individual design steps are then organized into complex workflow-driven transformation chains, which are closely aligned with the designated development process followed by the airframer or function provider. Finally, configuration tables for the standard ARINC 653 and VxWorks specific Module descriptions are generated based on the IAM models.

Additionally, to support certification, end-to-end traceability links from the PIALD to the generated configuration files are instantiated using both (i) inter-model traceability based on an integration model and (ii) model-to-configuration traceability with XMI files connecting generated configuration elements to their corresponding model representation.

My research in the context of DIANA was focused on three topics: (i) adaptation of MDE techniques for defining a specific development process for ARINC 653 configuration design that is in-line with DO-178B (ii) definition and reuse of model-based validation techniques for early error detection and localization and (iii) implementation of end-to-end traceability from high-level models to generated artifacts through the complete process that conforms to the certification requirements.

As a summary, the relevance of my contributions to model-driven engineering is twofold:

1. My first three contributions define two search-based techniques for design automation in model-driven engineering that can be applied independently of the application domain.
2. While my fourth contribution proposes and adapts MDE based techniques for safety-critical civil avionics configuration development.

1.6 The Structure of the Thesis

The thesis is structured into five parts containing ten chapters that contain overviews and new results, and four appendices complementing the main contributions of the thesis with additional information.

- Part 1: Model-Driven Engineering
  - Chapter 2 overviews the foundation of metamodeling and describes our AntWorld running example.
  - Chapter 3 introduces the basics of graph transformation based querying and modifications of models.

- Part 2: Advances in Graph Pattern Matching
  - Chapter 4 gives an overview on trends and techniques used for graph pattern matching in the model transformation literature.
  - Chapter 5 presents the concepts of local search based graph pattern matching based on search graphs and search plan.
  - Chapter 6 describes the use of hybrid pattern matching combining local search based and incremental pattern matching techniques through the proposed search graph concept.
• Chapter 7 concludes the work done in graph pattern matching and gives an overview on related work.

• Part 3: Design Space Exploration
  – Chapter 8 proposes a novel constraint satisfaction programming approach directly over graph models, which can be used to solve MDE based design space exploration problems. Additionally, the chapter gives a comparison with related DSE approaches on different case studies.

• Part 4: Model-Driven Development for Avionics Systems
  – Chapter 9 describes preliminaries and concepts to civil aeronautical development and certification with a special focus on integrated modular avionics.
  – Chapter 10 presents a model-driven development workflow as realized in the DIANA project for the semi-automated generation of ARINC 653 configuration artifacts.

• Part 5: Conclusions and Appendix
  – Chapter 11 concludes the main contributions and parts of this thesis and gives an overview on the different applications of the results and finally, outlines future research directions.
  – Appendix A contains the VIATRA2 source code of the complete AntWorld case study.
  – Appendix B describes the computational complexity analysis of the AntWorld case study with regards to the correlation between its parameters and runtime characteristics.
  – Appendix C describes the advantages of the hybrid pattern matching – as implemented in VIATRA2 – approach on the object-to-relational database schema model transformation case study.
  – Appendix D contains listings of the different XML configuration files automatically derived using the tool-chain developed for the DIANA project.

Notational guide
In order to maintain a consistence appearance of the thesis, the following rules are followed:

• The thesis is mainly written in third person singular. In conclusions after each chapter, I emphasize my own contributions by using first person singular or plural.

• Terms in formal definitions are printed in bold letters.

• New concepts, informal definitions are typeset in italics.

• Code extracts always appear as typewritten text in listings with grey background.

• For referring to texts in figures, slanted fonts are used.

• References to own publications appear as numbers (e.g., [10]), while citations from the bibliography are formatted alphanumerically (e.g., [VVF05]).
Part I

Model-Driven Engineering
Modeling Preliminaries

In the current chapter, we introduce the basics of modeling languages specification and follow the definitions defined in [Var08] and [RÍ0]. Throughout this and the following chapters on graph pattern matching we use the AntWorld [RG10] simulation benchmark as our running example. We follow the description of the case study as defined in [RG10].

2.1 Running Example: AntWorld

2.1.1 Motivation of the Case Study

This section will introduce the AntWorld case study, which will serve as the running example for the graph pattern matching related parts of the thesis. This choice was made based on the following considerations:

- It was an official graph-pattern matching performance evaluation case study, proposed at the 2008 Grabats (Graph-Based Tools) conference [Gra08].

- The pattern matching characteristic of the case study [2] is similar to my model-driven design space exploration (see in Chapter 8) and on-the-fly design contract validation (see in Section 10.6.2) thesis contributions.

It can be summarized as follows: (i) the graph pattern matching tasks are part of an in-place transformation (the source and target models are the same), (ii) graph patterns are medium sized consisting of up to fifteen model elements, (iii) a significant portion of model manipulations are concentrated on a restricted subset of the underlying model and finally, (iv) most of the model is a quasi-inmutable structure (either present in the initial model or created in a later phase but never modified during the complete execution process).
Finally, the AntWorld simulation is considered by many as one of the best case study for sheer graph pattern matching performance evaluation as it provides a fine-grain scaling from small to large input models and evades the problem of synthetic input model generation.

2.1.2 Overview of the AntWorld Simulation

The AntWorld case study simulates the life of a simple ant colony. The AntWorld simulation consists of an ant hill sitting in the middle of a large area. The ants are moving around searching for food. If an Ant finds food, it brings the food home to its ant hill in order to grow new ants. On its way home, the ant drops pheromones marking the path to the food reservoir. If an Ant without food leaves the hill or if a searching ant hits a pheromone mark, the ant follows the pheromone path to the food. This behavior already results in the well known ant trails.

As a simulation benchmark for model transformation tools the area in which the ants can move is modelled by a grid of nodes, which is constructed in a spider web shape so that the ants can travel back to the ant hill on a straight path. In the center of the grid there is the ant hill. The first circle around the ant hill consists of 4 exit fields. In the second circle, each exit field has three child fields. In the next circle, the 4 fields on the main axis have three child fields while the normal fields have just one child field. This scheme creates a quite regular grid where each field has a quite straight path to the hill in the center of the grid. An example ant hill with its surrounding is depicted in Figure 2.2.

The rules of the simulation are divided into two parts: area management and ant movement.

2.1.3 Ant Movement

The AntWorld simulation works in rounds. Within each round, each ant does one movement and it depends on the following rules:

- If the ant has no food and it is on a field with food, it takes one piece of food and enters the food carrying mode. It may still move within the current round.

- If the ant carries food, it follows the links towards the inner circle. Thus, the ant moves towards the ant hill by one circle. During its way home, on each visited grid node – including the food node – the ant drops 1024 parts of pheromones. This guides other ants to the food place. Note that if an ant drops new pheromones on an already marked grid node, this new pheromone parts are added to the already existing parts.

- If ant with food is on the hill node, it drops the food and enters the search mode. It may leave the hill within the same round.

- An ant without food is in search mode. In search mode, the ant checks the neighbor node(s) of the next outer circle for pheromones. If there are neighbor nodes on the next outer circle with more than 9 parts of pheromones, the ant chooses one of these fields, randomly.

- If the ant is in search mode and no outer neighbor has sufficient pheromones, the ant moves to any of its neighbor fields based on a fair random choice. However, an ant without food shall not enter the ant hill.

2.1.4 Area Management

- Initially, the area grid shall consist only of the hill and the first two circles. In addition, the hill shall contain 8 ants and there shall be no food on the grid.
• Whenever an ant enters the currently out most circle (i.e. the border of the yet known area), a new circle of nodes shall be created. During the creation of this next circle, every 10th node shall carry 100 parts of food. If a circle has e.g. 28 nodes, node 10 and node 20 of that circle shall have food. Thus, this circle would need just 2 more nodes to create a third food place. Therefore, across circles, every 10th node becomes a food place.

• After each round, the pheromones shall evaporate. This is needed in order to erase an old ant trail once the food has drained. Thus, after each round, on each grid node, the number of pheromones shall be multiplied by 0.95. In case of a fraction, the number shall be rounded to the next smaller natural number. Note that using the factor 0.95 and 1024 start pheromones per drop, the number of pheromones on a grid node drop below 10 roughly within 100 rounds. Note that food places with a distance of more than 50 circles need the collaboration of multiple ants that pick up the ant trail and refresh it, continuously.

• After each round, the hill shall consume the food brought to it and it shall create one new ant per delivered food part. These new ants spread out in the next round.

Goal of the Simulation

The goal of the simulation is to simulate the largest ant colony possible demonstrating the pattern matching capability of the underlying model transformation tool as the size of the model describing the actual state of the simulation is considerably smaller compared to the number of simulation rules needed to be executed.

2.2 Overview of Models and Metamodels

Metamodeling is a fundamental part of model transformation design as it allows the structural definition of modeling languages. The abstract syntax of a modeling language is defined by the metamodel. Metamodels are expressed using a metamodeling language that itself is a modeling language. Nodes of the metamodels are called entities and essentially they represent basic concepts of a modeling domain. Attributes may be defined for entities to represent content information using simple types (e.g., integer, double, string) Inheritance may be defined between entities, which means that the inherited entity has all the attributes and properties of its parent and may contain additional ones. Relations define connections between entities and define a relationship between the entities. They may have multiplicity constraints attached to both end of a relation, which restricts the number of model entities in the instance model.

The instance model is a graph that describes concrete systems defined in a modeling language. Its nodes and edges are called objects and links, respectively. The instance model is a well-formed instance of the metamodel, which means the fulfillment of a set of criteria, described in Def. 3.

Example 1 The metamodel of the AntWorld case study used in our experiments is shown in Figure 2.1.

The entity Field represents a field of AntWorld (grid node in the original specification, but we use the term field to avoid confusion); CornerField entities are fields that are located on the axis, and the AntHill is the central field. Fields are connected by paths. Each circular path formed by circlePath relations connects the set of fields that were created in a single round. Except for the anthill, each field has a single outgoing returnPath relation pointing towards a field in the previous circle; most fields have a single incoming returnPath as well, but corner fields have three, and the
CHAPTER 2. MODELING PRELIMINARIES

anthill has four. The boundary relation is used to ease the search for the actual boundary fields of the grid by pointing them directly from the ant hill. Fields may be associated with an integer number of food items or pheromones associated with them. Finally, a field may contain two kinds of ants: SearcherAnt entities represent ants that do not carry food but are in search of a food bundle, while CarrierAnt entities represent ants that carry a food item, modeled by the FoodBite entity, and are on their way to return to the anthill.

Example 2 Figure 2.2 shows an example area of the ant colony, where the ants have not yet found any food, thus none of the fields has pheromone and all ants are SearcherAnts. The example instance model shows the state of the simulation in the end of a round. However, as ants have reached the boundary of the "known" world the round finishes with the generation of the next circle of the grid according to the defined rules (see in Section 2.1.4).

2.3 Modeling Environments

There is already a large set of metamodeling environments used by both the research community and the industry. In the current section, we give a brief overview on the most widely accepted approaches.

2.3.1 MOF - Meta Object Facility

The Meta Object Facility is a standard of the Object Management Group (OMG). It was designed to be the basis for all UML and related languages (e.g., CWM, SysML) and serves as a platform-independent metamodeling framework and a general mapping platform between various languages. Its current version is called MOF 2 version 2.4.1 [Obj11].

MOF has a core sub-language called Essential MOF (EMOF) that defines the minimal number of concepts required for the building of metamodels. EMOF has been inspired by the Eclipse Modeling Framework an open source project aimed the development of a metamodeling framework for
2.3. MODELING ENVIRONMENTS

the Eclipse tool platform. However, only a subset of the MOF language has been used during the implementation of the final ECore modeling core that proved to be sufficient for practical modeling scenarios (see in Section 2.3.2).

Figure 2.3 illustrates the main concepts of EMOF without many auxiliary elements. The central element is the *Class* that can have *Properties*. Properties can be typed by primitive types defining a simple value slot resulting in an attribute, while inter-class relations are defined by uni-directional *Associations*, with the ability to define multiplicity constraints on their both end and paired using the opposite relation. Additionally, *Operations* can be defined for classes to capture domain specific functions. One of its main drawbacks is it lacks formal semantics and thus different communities interpret certain part of the standard in different way, resulting in lack of compatibility and reusability.

2.3.2 Ecore - Eclipse Modeling Framework

The Ecore metamodeling language is the hearth of the *Eclipse Modeling Framework* [Thea] (EMF). It has been developed to provide a method for metamodel definition that directly supports the implementation of models using conventional programming languages. Introducing the Ecore separately resulted in that the language became the *de facto* metamodeling standard of the industry definition and several other standards [SAEb] and domain specific languages [Joh09] are defined using this formalism. As mentioned earlier in Section 2.3.1 the Ecore language is almost identical to the EMOF language. The core concepts of the Ecore language are depicted in Figure 2.4.

*EClass* models classes (nodes) themselves. Classes are identified by their name and can contain a number of attributes and references. In Ecore multiple inheritance is allowed. *EAttribute* models
attributes that contain the data elements of an EClass. They are identified by their name, and they have a data type. EDataType is used to represent simple data types that are stored as atomic values (their internal structure is not modeled). They are also identified by their names. EReference is used in modeling associations between EClasses. It can be constrained by multiplicity constraints. It is also possible to mark a reference as containment that represents composition relation between elements. Bidirectional association are modeled as two EReference instances that are mutually connected via their opposite references.

These four elements define the kernel of the Ecore language, however, it contains many auxiliary classes for the organization of the models.

Additionally, EMF supports a wide range of functionality for easing the integration of the defined models to any Java program. This includes automated Java representation generation from Ecore, reflective model manipulation, notification mechanism, dynamic model creation, XMI import and many more. All these add-ons and binary compatibility made EMF one of the most widely accepted and used modeling environment by the industry. Its only drawback is again the lack of formal
2.4. CONCEPTS OF METAMODELS AND INSTANCE MODELS

semantics and poor specification, which in many cases results in “specification by implementation”.

2.3.3 VPM - Visual and Precise Metamodeling

Visual and precise meta-modeling framework (VPM) (the built-in metamodeling framework of Via-tra2) is a meta-modeling language, which is capable of defining models and meta-models in a single model space. The VPM framework has a mathematically precise notation, which eases model checking and validation. A VPM model has a graph-based structure facilitating the design and maintenance with visual tools. VPM maintains explicit instanceOf relationship supporting the storage of meta-models and models in the same model space resulting in easy multi-level metamodeling and multi-domain integration. Additionally, it allows dynamic typing of entities.

![Figure 2.5: The VPM language](image)

The VPM language consists of two basic elements: the Entity (a generalization of MOF package, class, or object) and the Relation (a generalization of MOF association end, attribute, link end, slot). Both element have a common abstract supertype called Model that defines the name String attribute that can be used to be the identifier of the model elements. Entities represent basic concepts of a (modeling) domain, while relations represent the relationships between other model elements. Furthermore, entities may also have an associated String value that contains application-specific data. The are three special relationships defined between Models: (i) the supertypeOf relation represents generalization relationship between Models, (ii)the instanceOf relations expresses instantiation between Model elements and explicitly stored allowing direct manipulation, and (iii) the owner relation that arranges the model elements into a strict tree containment hierarchy. Its main advantage is that it helps human readers to understand the model and can be used to reduce the scope of queries.

VPM has two advantage over MOF and Ecore: (i) its precise formal semantics that further analysis over the defined model and (ii) its ability through the direct instanceOf relation of multiple typing and thus multi-level metamodeling. However, as one of its main drawback VPM is far more verbose compared to MOF and Ecore and thus require far more elements to define the same domain.

2.4 Concepts of Metamodels and Instance Models

A description of the most important formal concepts of modeling and metamodeling is presented following the notation of [Var08] and [R10].
Definition 1 A directed attributed graph (denoted by $G = (V_G, E_G, src_G, trg_G, A_G, attr_G, value_G)$) is a 7-tuple, where $V_G$ and $E_G$ denote nodes and edges of the graph, respectively. Functions $src_G : E_G \rightarrow V_G$ and $trg_G : E_G \rightarrow V_G$ map edges to their source and target node, respectively. $A_G$ denotes the attributes of the nodes with a name and type pair, where types are considered simple atomic types (for simplicity in our case: string, integer or double), $attr_G : V_G \rightarrow A_G$ is a function that map nodes to their attributes and finally, the $value$ function denotes the value of an attribute of a given node, formally, $value_G : V_G \times A_G \rightarrow value$.

Although, to ease the understanding this is a simplified representation of attribute graphs it is still suitable for demonstrating the pattern matching results of this thesis.

Definition 2 A metamodel $MM = (M, supertypeOf, multiplicity)$ is a triple, defined by a directed attributed graph $M$, the $supertypeOf : M \times M \rightarrow \text{Boolean}$ predicate denoting supertyping relationship and the $multiplicity : E_{MM} \rightarrow \text{one-to-one, one-to-many, many-to-one, many-to-many}$ function denoting the multiplicity type of the relations, where

- $V_{MM}$ and $E_{MM}$ denote nodes and edges of the metamodel.
- An entity $E$ is a node of the metamodel, formally, $E \in V_{MM}$.
- A relation $R$ is an edge of the metamodel, formally $R \in E_{MM}$.
- Inheritance (generalization) (denoted as in UML by $C_s \leftarrow C_i$) is represented by the $supertypeOf$ predicate, where $E_{super}$ is a supertype of $E_{sub}$ iff $supertypeOf(E_{super}, E_{sub})$ holds and there is no other entity $E_x$ between $E_{super}$ and $E_{sub}$ in the inheritance hierarchy, formally, $\not\exists E_x : supertypeOf(E_{super}, E_x) \land supertypeOf(E_x, E_{sub})$. Note that multiple inheritance is allowed.
- Inheritance is transitive along the $supertypeOf$ relationship, $\forall Sub, S_1, S_2 \in MM : supertypeOf(Sub, S_1) \land supertypeOf(S_1, S_2) \Rightarrow supertypeOf(Sub, S_2)$.
- Relations have multiplicity constraints, which impose a restriction on the instance model. Formally, $\forall R \in E_{MM} : multiplicity(R)$. For detailed definition see in Def. 3
- Attribute values are not allowed in metamodels, formally, $\not\exists E \in V_{MM}, \not\exists Attr \in A_{MM} : attr_{MM}(E) = Attr \land \exists value_{MM}(E, Attr)$.

Definition 3 Given a metamodel $MM$, a (well-formed) instance model $M$ of the metamodel $MM$ is a directed attributed graph together with a direct type predicate $instanceOf : M \times MM \rightarrow \text{boolean}$, which denotes instantiation relationship between $M$ and $MM$, according to the following rules

- All element of the instance model (objects) are mapped to metamodel entities, formally, $\forall object \in V_M, \exists Type \in V_{MM} : instanceOf(object, Type)$.
- All objects can only have attributes that are defined in type metamodel entity, formally, $\forall obj \in V_M, \forall att \in A_M, \exists E \in V_{MM}, \exists ATT \in A_{MM} : attr_{MM}(obj) = att \land attr_{MM}(E, ATT) = att \land instanceOf(obj, E)$.
2.5. SUMMARY

- All model edges (links) are mapped to relations, formally, $\forall link \in E_M, \exists Type \in E_{MM} : instanceOf(link, Type)$. Additionally, for all links it holds that their source end, the object is an instance of Type's source entity, and its target end, the object is an instance of Type's target entity. Formally, $\forall Type, instanceOf(link, Type) : instanceOf(src(link), src(Type)) \land instanceOf(trg(link), trg(Type))$.

- The instanceOf relationship is transitive along supertyping:
  $\forall Inst \in M \land \forall Type, Super \in MM : instanceOf(Inst, Type) \land supertypeOf(Type, Super) \Rightarrow instanceOf(Inst, Super)$.

- Multiplicity criteria
  - one-to-one: Two relations of the same type with the same source have to have the same target, formally, $\forall Type \in E_{MM}, \forall link_1, link_2 \in E_M : instanceOf(link_1, Type) \land instanceOf(link_2, Type) \land src_M(link_1) = src_M(link_2) \iff trg_M(link_1) = trg_M(link_2)$.
  
  - one-to-many: Two relations of the same type having the same target cannot come from different sources, formally, $\forall Type \in E_{MM}, \forall link_1, link_2 \in E_M : instanceOf(link_1, Type) \land instanceOf(link_2, Type) \land trg_M(link_1) = trg_M(link_2) \Rightarrow src_M(link_1) = src_M(link_2)$.
  
  - many-to-one: Two relations of the same type coming from the same source cannot go to different targets, formally, $\forall Type \in E_{MM}, \forall link_1, link_2 \in E_M : instanceOf(link_1, Type) \land instanceOf(link_2, Type) \land src_M(link_1) = src_M(link_2) \Rightarrow trg_M(link_1) = trg_M(link_2)$.
  
  - many-to-many: There is no restrictions on the multiplicity of the relation.

- No parallel edges are allowed, meaning that there cannot be any pair of links of the same type leading between the same pair of objects in a given direction. Formally, $\forall link_1, link_2 \in E_M, Type \in E_{MM} : src_M(link_1) = src_M(link_2) \land trg_M(link_1) = trg_M(link_2) \land instanceOf(link_1, Type) = instanceOf(link_2, Type) \Rightarrow link_1 = link_2$.

2.5 Summary

In the current chapter, the basis of modeling language specification has been overviewed by introducing metamodels for describing modeling domains and instance models for specifying concrete systems. These concepts have been formalized based on the notations of [Var08] and have been exemplified by using the AntWorld simulation case study as our running example. Finally, an overview on three widely accepted modeling environments has been given.
Model Transformation

Model transformation is the backbone of model-driven engineering. It eases the automated manipulation of models to propagate information along the complete development process on all abstraction levels from the high-level design models down to the generated source code.

There are already a large set [CH03] of different dedicated model transformation tools: Epsilon [RPKP08], VMTS [VS], ATOM3 [dLV02], ATL [ATL], AGG [Tae00], GROOVE [Ren04a] or FUJABA [NNZ00] just to name a few examples. All of these tools solve a common problem: the ease specification and fast execution of model transformation through declarative specification and optimized execution architectures. Many of them are based on the formal concept of graph transformation [Roz97].

In the current section, we present the basics of graph transformation a declarative formalism to define model transformation. Its concepts are presented using our AntWorld running example.

3.1 Modeling the AntWorld case study

The current chapter introduces our model transformation based simulation program for the AntWorld case.

As my research on graph pattern matching has been carried out in the context of our Viatra2 model transformation framework [Via], in Section 3.1.1, we will introduce the basics of the Viatra2 transformation language in order to become familiar with its notation and definitions [BV06, Var04]. This eases the understanding of our solution for the AntWorld case study as it was implemented in Viatra2.

3.1.1 The Viatra2 Transformation Language

In order to understand the concepts of Viatra2 graph transformation environment, we give a brief overview on the transformation language of the framework.
The transformation language of VIATRA2 (Viatra Textual Command Language – VTCL [VB07]) consists of several constructs that together form an expressive language for developing both model to model transformations and code generators. Graph patterns (GP) define constraints and conditions on models, graph transformation (GT) [EEKR99] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [BS03] rules can be used for the description of control structures.

Readers familiar with graph patterns and transformation rules may directly check the GT rules and graph patterns defined for the AntWorld case study as depicted in Figure 3.1.

3.1.1.1 Graph patterns

Graph patterns represent conditions (or constraints) that have to be fulfilled by a part of the model space in order to execute some manipulation steps on the model. The basic pattern body contains model element and relationship definitions.

In VTCL, patterns may call other patterns using the find keyword. This feature enables the reuse of existing patterns as a part of a new (more complex) one. The semantics of this reference is similar to that of Prolog clauses: the caller pattern can be fulfilled only if their local constructs can be matched, and if the called (or referenced) pattern is also fulfilled. For complex pattern specification the VTCL language allows to define alternate (OR) pattern bodies for a pattern, with a meaning that the pattern is fulfilled if at least one of its bodies is fulfilled. Finally, the signature of a graph pattern is called pattern head, which consist of its name and its input/output parameters.

A negative application condition (NAC, defined by a negative subpattern following the neg keyword) prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match. Negative conditions can be embedded into each other in an arbitrary depth (e.g. negations of negations), where the expressiveness of such patterns converges to first order logic [Ren04b].

Example 3 The VTCL definition of the anyNeighborButHome graph pattern (see in Figure 3.3) is shown in Listing 3.1.

```plaintext
pattern home(Field2) = 
  { AntHill(Field2); }

pattern anyNeighborButHome(Field1, Field2) = 
  { field(Field1);
    field(Field2);
    field.path(P, Field1, Field2);
    neg find home(Field2);
  } or 
  { field(Field1);
    field(Field2);
    field.path(P, Field2, Field1);
    neg find home(Field2);
  }
```

Listing 3.1: VIATRA2 source code for the anyNeighborButHome pattern

Entities of a pattern body are defined using their type with the pattern node (variable) as their parameter. For example, field(Field1) represents that the Field1 node has to be matched to a field typed object. Relations are defined as a triplet, where the first parameter is the pattern relation, the second defines the source of the relation while the third the target. For example, the field.path(P,
Field1, Field2) relation defines that the P pattern relation has to be matched to a link in the instance model that comes from the object that is matched to the Field1 pattern node and goes to an object that is matched to Field2 and finally, this relation has to have path type.

As already mentioned, the pattern uses alternate pattern bodies to represent moving in the forward or reverse direction of a path relation between Field1 and Field2. It also reuses the home pattern (called by the neg find construct) as its NAC to put the not AntHill type constraint on Field2. The pattern head is the signature of the pattern itself: anyNeighborButHome(Field1, Field2).

3.1.1.2 Graph transformation rules

In VTCL, graph transformation rules may be specified by using a precondition (LHS) pattern determining the applicability of the rule, and a postcondition pattern (RHS) which declaratively specifies the result model after rule application.

Elements that are present only in (the image of) the LHS are deleted, elements that are present only in the RHS are created, and other model elements remain unchanged. Further actions can be initiated by calling any ASM instructions within the action part of a GT rule, e.g. to report debug information or to generate code.

**Example 4** For instance, the GT rule return defines how food carrying ants take a step towards the hill, as shown in Listing 3.2 (graphical representation in Figure 3.2). The mechanism of leaving pheromones is omitted here for the sake of brevity.

The GT rule uses the precondition graph pattern called lhs to define that a Ant of type CarrierAnt is standing on an OuterNeighbor Field element connected through a hasCarrierAnt relation and this OuterNeighbor has a directed RP returnPath relation to the InnerNeighbor Field.

In its postcondition called rhs it defines that the oldHA relation needs to be deleted, thus it is not present in the pattern. On the other hand the newHA relation between the OuterNeighbor and the Ant needs to be created and thus it is present in the rhs pattern.

```java
// Ant returns along returnpath.
grule return(in Ant) = {
  precondition pattern lhs(Ant, InnerNeighbor, OuterNeighbor, Loc) = {
    field(InnerNeighbor);
    field(OuterNeighbor);
    field.returnPath(RP, OuterNeighbor, InnerNeighbor);
    carrierAnt(Ant);
    carrierAnt.hasCarrierAnt(oldHA, Ant, OuterNeighbor)
  }
  // Deletes OldHA and creates NewHA
  postcondition pattern rhs(Ant, InnerNeighbor, OuterNeighbor, newHA) = {
    field(InnerNeighbor);
    field(OuterNeighbor);
    field.returnPath(RP, OuterNeighbor, InnerNeighbor);
    carrierAnt(Ant);
    carrierAnt.hasCarrierAnt(NewLoc, Ant, InnerNeighbor);
  }
}
```

Listing 3.2: VIATRA source code for graph transformation rules
3.1.1.3 Model manipulation

The ASM language of ViATRA2 also includes constructs to directly manipulate models from ASM rules. It is important to point out that in our solution we tried to use declarative GT rules for all simulation operation, however, certain rules could not be defined as GT rules and thus it was implemented as ASM rules.

The example code shown in Listing 3.3 demonstrates how the sequence of creating a new Food is created when the world is expanded using ASM model manipulation constructs. First, it is checked that the actual input Field is the tenth in the creation using the foodCounter ASM function that is store for global variables. If not it simply increases the counter by one using the update construct. If it is the tenth Field in the creation process than the foodCounter is set to 0, followed by the creation (new keyword) of (i) a food typed model element that is reachable through the Food variable and (ii) a food .hasFood relation that goes from the model element of the Field variable to the model element of the Food variable. Finally, the value of the Food model element is set to one hundred.

```
rule newField(in Field) =
  if (foodCounter() < 9) update foodCounter() = foodCounter() + 1;
  else let Food = undef, HF=undef in seq {
    update foodCounter() = 0;
    new (food(Food));
    new (field.hasFood(HF, Field, Food));
    setValue(Food, 100);
  }
```

Listing 3.3: ViATRA2 source code for food grabbing

3.1.1.4 Control structure

To control the execution order and mode of transformations, abstract state machines [BS03] are used. ASMs provide complex model transformations with all the necessary control structures including the sequencing operator (seq), ASM rule invocation (call), variable declarations and updates (let and update constructs), if-then-else structures, non-deterministically selecting (random) constructs, iterative execution (applying a rule as long as possible (ALAP) iterate), the simultaneous rule application at all possible matches (locations) (forall) and single rule application on a single match (choose).

Example 5 The example code shown in Listing 3.4 demonstrates how typical control structure combinations are used in ViATRA2.

The first choose rule tries to find a single match for the Ant variable (defined in its head), which satisfy the precondition of the grab GT rule and apply the GT rule with that variable. If more ants satisfy the precondition, then one is chosen non-deterministically and if there are no such substitutions then the choose rule fails.

Using the iterate rule in the example allows to apply its choose rule as-long-as-possible (ALAP), i.e. as long as a match for the precondition of the grab GT rule can be found.

As for the following forall rule, it finds all substitutions (matches) for the variable defined in its head (Ant), which satisfy the precondition of the deposit GT rule, and then apply the GT rule for each substitution separately. If no variable substitutions satisfy the precondition of the GT rule, then the forall rule is still successful and does not fail. In contrast to the iterate rule, it first collects all available matches and then applies its GT rule for each in a single step.
3.1. MODELING THE ANTWORLD CASE STUDY

3.1.2 Description of the AntWorld Solution

The sequence shown in Listing 3.5 defines how one iteration of the AntWorld case study is managed using graph transformation rules driven by ASM rules. An iteration is divided into seven different phases; four for the ant simulation and three for the world management. All phases are captured by a combination of for all and choose structures using graph transformation rules and graph patterns (see in Figure 3.1). How each phase manages its task is described in the following list:

// main method for an iteration
rule doRound() = seq {
// init actions
iterate choose Ant with apply grab(Ant);
forall Ant with apply deposit(Ant);
forall Ant, FromField with apply return(Ant, FromField)
   do call leavePheromone(FromField);
forall Ant with find searcher(Ant) do call search(Ant); // two kinds of search

// World management
forall Pheromone with find pheromone(Pheromone) do call evaporate(Pheromone); // random
forall Hill with apply consume(Hill);

// only searchers can breach the boundary!
if (find boundaryBreached())
   do call growGrid(); // grow the game map
}

// two type of moves
rule search(in Ant) =
   choose Field1, HA1 with find hasAnt(HA1, Field1, Ant)
   try choose Field2 with
      apply moveTowardsAttractingPheromone(Ant, HA1, Field1, Field2)
      // moves toward pheromone
   else choose Field2 with
      apply moveAnywhereButHome(Ant, HA1, Field1, Field2);
   // simply searches for food

// grows the grid as specified
rule growGrid() = seq {
   update circlesTotal() = circlesTotal() + 1;
  forall V with apply expandVerticalSide(V);
  forall H with apply expandHorizontalSide(H);
  forall C with apply expandCorner(C);
  forall F1, F2 with apply completeCircle(F1, F2);
}

Listing 3.5: Viatra2 source code for an iteration

Ant simulation

- **Grab phase:** First, the food gathering is managed by an ALAP execution of the grab (see in Figure 3.1(a)) GT rule. This way for each ant that stands on food pile the remaining amount of food is calculated by the grab GT rule. If it is positive, the new value of the food bundle is set accordingly, otherwise the food bundle is exhausted and deleted from the model (see in the action part of the GT rule).
CHAPTER 3. MODEL TRANSFORMATION

• **Deposit phase:** The deposit GT rule (depicted in Figure 3.1(b)) in a forall construct identifies all carrier ants that have successfully delivered a food bundle to the hill and leaves their carried FoodBite on the hill.

• **Return phase:** In this phase all carrier ants that did not reach the hill yet, will step one field closer to home along the returnPath relation. This is done in a single forall construct where the return GT rule (see in Figure 3.5) is responsible to select the appropriate carrier ants and move them towards the anthill along the returnPath while the leavePheromone leaves a pheromone on its FromField input parameter. The first checks that the input FromField already has a Pheromone using a try-else control structure combined with a choose invoking the hasPheromone pattern. If it has, then simply add 1024 to its integer value, otherwise the else branch executes attaching a newly created Pheromone with 1024 as its value to the Field. Note that because the deposit phase was already executed in the current iteration there are no carrier ants standing on the anthill for which the hasCarrierAnt pattern could be reused.

For each GT rule application the leavePheromone rule is invoked once.

• **Search phase:** Finally, searcher ants looking for a food source are actuated by the search rule (handling both the unguided and pheromone-guided cases) executed in a forall construct using the searcher (see in Figure 3.1(i)) pattern. The search rule retrieves the Field1 field on which the input Ant is standing. Then using the try-else construct invoked by a choose rule, first it tries to apply the moveTowardsAttractingPheromone GT rule by checking using its lhs if there is any pheromone infested field neighboring Field1. If there is, then it steps to that by executing the GT rule, otherwise the GT rule fails and so the else branch is executed meaning that the ant will step to one of the neighboring fields (except the anthill) using the anyNeighborButHome GT rule (detailed in Sec. 3.1.1.1).

**Area Management**

• **Evaporate Pheromone phase:** To volatilize the Pheromones in the model the simple pheromone pattern (see in Figure 3.1(h)) in a forall construct invoking the evaporate rule is used. In the evaporate rule first the remaining amount of pheromone is calculated. If this is positive, the new value of the pheromone is set accordingly; otherwise the pheromone is exhausted and deleted from the model. The calculation is kept in the integer domain using the JAVA built in rounding mechanism on the division operator.

• **Create Ants:** As the number of food bundles on the anthill is managed by the number of FoodBite entities on the hill itself, the creation of the ants is handled using the simple consume GT rule (see in Figure 3.1(f)) with a forall execution construct. At every invocation of the consume GT rule it deletes a FoodBite from the hill and creates a new SearcherAnt with its hasSearcherAnt relation pointing to the anthill.

• **Boundary Breached phase:** Finally, in order to check that a searcher ant has reached the boundary of the actual world an if construct is used to check that the growGrid rule (see in Figure 3.1(c)) pattern matches to the actual model. If it matches the growGrid rule is invoked to handle the expansion of the world. The algorithm used is a circular based traversal of the boundary fields along the circularPath starting from a randomly selected border field. During the traversal for each boundary field a new outer neighbor is created connected to its
3.2. FORMALIZING MODEL TRANSFORMATION AS GRAPH TRANSFORMATION

Graph transformation (GT) [Roz97] provides a declarative, rule and pattern-based language for specifying both inter-language and intra-language model queries and manipulations for model analysis, refactoring, simulation etc.

The complete source code of the case study is available in Appendix A.

3.2 Formalizing Model Transformation as Graph Transformation

The following graph transformation rules and patterns are used in the doRound rule:

(a) grab GT rule

(b) deposit GT rule

(c) boundaryBreached GT pattern

(d) moveTowardsAttractingPheronome GT rule

(e) moveAnywhereButHome GT rule

(f) consume GT rule

(g) hasAnt GT rule

(h) pheromone pattern

(i) searcher pattern

neighboring fields along with the update of the boundary relation from the hill. The only exception from this rule are the CornerFields where three new (a CornerField and two simple Field fields along with their relations between them and the boundary relation to the hill) fields are generated to create the new corner of the actually constructed boundary. This way the distribution of the newly created food bundles are also arranged during the traversal and on every tenth newly created boundary Field an additional Food bundle is added using the already mentioned foodCounter ASM function.

The complete source code of the case study is available in Appendix A.
GT rules can be specified by using graph patterns. Graph patterns are frequently considered as the atomic units of model transformations [VB07]. They represent conditions that have to be fulfilled by a part of the underlying instance model.

A graph transformation rule consist of a left-hand side – LHS (or precondition) graph pattern determining the applicability of the rule, and a right-hand side – RHS (postcondition) graph pattern, which declaratively specifies the result model after rule application. Elements that are present only in (the image of) the LHS are deleted, elements that are present only in the RHS are created, and other model elements remain unchanged.

Example 6 The GT rule return defines how the ants that are carrying food take a step towards the hill, as shown in Figure 3.2. The rule simply deletes the OldHA hasCarrierAnt relation going from the OuterNeighbor Field entity to the Ant entity and and creates the NewHA relation between the InnerNeighbor and Ant entities.

3.3 Concepts of Graph Patterns and Graph Transformation

A description of the most important formal concepts of graph patterns and transformation rules is presented, following its introduction as described in [Var08].

3.3.1 Graph Pattern

Definition 4 Given a metamodel $MM$, a pattern body $PB = (SC, AC, \forall j \in J NAC_j)$ is a tuple consisting of

- structural constraints $SC$ prescribing the existence of nodes and edges described by a graph that is conform to the metamodel $MM$.

- attribute constraints ($AC$) prescribe boolean conditions over the attributes of the matched elements, formally, $attrcondition_{i,j} : G \rightarrow \text{Boolean}$

- negative application conditions $NAC = (GP_{NAC}, p_{NAC})$, defined by a graph pattern $GP_{NAC}$ that prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match. The nodes and edges shared between the negative application condition and the container pattern body are defined by the injective partial morphisms $p_{NAC} : SC_{PB} \mapsto GP_{NAC}$.
3.3. CONCEPTS OF GRAPH PATTERNS AND GRAPH TRANSFORMATION

Definition 5 A graph pattern $GP = \bigvee_{i \in I} PB_i$ is a disjunction of pattern bodies. It means that a graph pattern may be represented by more than one pattern bodies, where the graph pattern is matched if at least one of its pattern bodies is matched to the underlying model.

Example 7 As an example, the ants that are searching for food, but are not attracted by a pheromone trace, use the anyNeighborButHomegraph pattern to determine which field to move to. This pattern, used to match neighboring fields (excluding the AntHill) is shown in Figure 3.3.

This pattern uses alternate pattern bodies to represent moving in the forward or reverse direction of a path relation between Field1 and Field2. It also uses a NAC (depicted by the NEG keyword) to put the not AntHill type constraint on Field2.

3.3.2 Matching of Graph Patterns

Definition 6 A match of graph $G$ in a model $M$, where both conforms to a metamodel $MM$, is an injective, type conformant total morphism $\text{morph} : G \rightarrow M$, which means that

- Type conformance of nodes. $\forall \text{node} \in V_G, \exists \text{obj} \in V_M : \text{instanceOf}(\text{node}, z) \land \text{instanceOf}(\text{obj}, z) \land z \in MM \land m_G(\text{node}) = \text{obj}$;

- Type conformance of edges. $\forall \text{edge} \in E_G, \exists \text{link} \in E_M : \text{instanceOf}(\text{link}, z) \land \text{instanceOf}(\text{edge}, z) \land z \in MM \land m_G(\text{edge}) = \text{link}$;

- Injective mapping of nodes. $\forall \text{node}_1, \text{node}_2 \in V_G, \exists \text{node}_1 = \text{node}_2$;

- Injective mapping of edges. $\forall \text{edge}_1, \text{edge}_2 \in E_G, \exists \text{edge}_1 = \text{edge}_2$;

Definition 7 A match of graph pattern $GP = \bigvee_{i \in I} PB_i$ in an instance model $M$ denoted by $m : GP \rightarrow M$, means that there exists a pattern body $\exists PB_i = (SC_i, AC_i, NAC_{i,j} = (GP_{NAC_{i,j}}, p_{NAC_{i,j}})) \in GP$ where:

- $\exists m : SC_i \rightarrow M$ there exists an injective, type conformant total morphism $m$ from the graph $SC_i$ to the instance model $M$,

- $\nexists j \in J$ where $m' : GP_{NAC_{i,j}} \rightarrow M$ there is no match for any of its embedded NACs that extends the match of the pattern body $PB_i$.

- $\forall \text{attrcondition}_{i,j} \in AC_i : \text{attrcondition}(m) = true$. All attribute conditions defined in $AC_i$ are fulfilled by $m$. 

Figure 3.3: Graph pattern to select any field that is not a neighbor of the AntHill
For the rest of the thesis we will refer to a match as a match of a graph pattern unless stated otherwise.

**Example 8** A match of the `anyNeighborButHome` graph pattern on a small extract of the AntHill model is depicted in Figure 3.4. The pattern has two matches:

- In the first case – the match is depicted by grey dashed lines going from the pattern element to its matching pair in the instance model – the `CF1,P1,CF3` elements of the instance model are matched to the `Field1, P, Field2` pattern elements of the first pattern body.

- The other match consist of the `CF3,P2,CF2` elements (with ID 2) that are again matched to the `Field1, P, Field2` pattern elements.

Note that in practical model transformations, usually, not all elements of a pattern body are passed out in the match of the graph pattern; thus a signature or pattern head is defined that specify those elements that are part of the passed out match. In the current case these passed out pattern elements are `Field1` and `Field2` and thus the two matches are `CF1,CF3` and `CF1,CF3`.

### 3.3.3 Graph Transformation

**Definition 8** A graph transformation rule $\text{GT} = (\text{LHS}, \text{RHS}, \text{AMO}, p)$ is a tuple where,

- $\text{LHS}$ is a graph pattern determining the applicability of the rule

- $\text{RHS}$ is a restricted graph pattern specifying the result model after rule application. It can have only one pattern body that prescribes only structural constraints and has no embedded NACs, formally, $\text{RHS} = (\text{SC}_1, \emptyset, \emptyset)$

- $\text{AMO}$ is a set of attribute manipulation operations.
3.3. CONCEPTS OF GRAPH PATTERNS AND GRAPH TRANSFORMATION

- \( p : \text{LHS} \rightarrow \text{RHS} \) is a partial injective morphism (mapping), which identifies the model elements that are not manipulated by the rule. These are the elements that appear in both the \( \text{LHS} \) and the \( \text{RHS} \).

**Example 9** In the current thesis, for easier readability we will use a merged graphical representation initially introduced in [FNTZ00], where the union of the \( \text{LHS} \) and \( \text{RHS} \) graphs are presented rather than the two separate graphs as in Figure 3.2. Elements to be deleted or created are marked by the \( \text{del} \) or \( \text{add} \) keywords, respectively, while negative condition are denoted by the \( \text{NEG} \) keyword (see in Figure 3.3).

The \( \text{return} \) GT rule using the merged graphical representation is depicted in Figure 3.5

3.3.4 Application of Graph Transformation Rules

The application of a \( r = (\text{LHS}, \text{RHS}, \text{AMO}, p) \) rule to a host model \( G \) alters the model by replacing the pattern defined by \( \text{LHS} \) with the pattern defined by \( \text{RHS} \). This is performed in two phases and is equivalent with the single pushout approach as defined in [Roz97]:

- **Pattern matching:**
  1. Finding a match of the \( \text{LHS} \) pattern in model graph \( M \) (by graph pattern matching). A match of a graph transformation rule refers to the match of its \( \text{LHS} \) \( m : \text{LHS} \rightarrow M \)

- **Updating:**
  2. **Deletion:** Removing a part of the model graph \( M \) that can be mapped to \( \text{LHS} \) but not to \( \text{RHS} \), denoted by deletion sub-phase
  3. **Insertion**
     a) adding new elements which exist in \( \text{RHS} \) but not in \( \text{LHS} \)
     b) finally, performing the attribute manipulation operations described in \( \text{AMO} \).

**Definition 9** Given a match \( m \) for a rule \( r \) in model \( M \), the deletion phase of a rule application of the rule \( r \) is executed on a match \( m \) in the model \( M \) resulting in the context model \( M_c \), when

- we delete all objects, to which nodes appearing only in the \( \text{LHS} \) and are mapped by \( m \), formally, \( V_{M_c} = V_M \setminus \Delta V_M \), where \( \Delta V_M = \{ c \mid \exists x \in V_{\text{LHS}} \setminus V_{\text{RHS}} \wedge m(x) = c \} \); and
CHAPTER 3. MODEL TRANSFORMATION

- we delete all links, to which edges appearing only in the LHS (but not in RHS) and are mapped by \( m \), formally,
  \[
  \Delta E_1^- = \{ \text{link}_{del} | \exists \text{link}_{gt} \in E_{LHS} \setminus E_{RHS} \land m(\text{link}_{gt}) = \text{link}_{del} \}
  \]

- all dangling (i.e., incident) edges are deleted, formally,
  \[
  \Delta E_2^- = \{ \text{link}_{del} | \exists \text{obj}_{gt} \in V_{LHS} \setminus V_{RHS} \land src(\text{link}_{del}) = a \land trg(\text{link}_{del}) = b \land (m(\text{obj}_{gt}) = a \lor m(\text{obj}_{gt}) = b) \}
  \]

Finally, the deletion of links is performed as \( E_{M_c} = E_M \setminus \{ \Delta E_1^- \cup \Delta E_2^- \} \).

**Definition 10** Given a match \( m \) for a rule \( r \) in model \( M \) that is a valid instance of \( MM \), the insertion phase of a rule application of the rule \( r \) is executed on a match \( m \) in the context model \( M_c \) resulting the model \( M' \), if a match \( m_{RHS} \) can be prepared for the RHS to model \( M' \) in the following way.

- Each mapped node \( \text{node}' \) of RHS is mapped by match \( m_{RHS} \) to the node object that has been assigned to its origin node \( \text{node} \) by match \( m \), formally, \( \forall \text{node}', \text{node} \in V_{LHS} \cup V_{RHS} : p(\text{node}) = \text{node}' \Rightarrow m_{RHS}(\text{node}') = \text{node} \)

- For each unmapped node \( \text{node}' \) of RHS, a new object \( \text{obj}_{new} \) is created with the same type as of \( \text{node}' \) and it is assigned to \( \text{node}' \) by match \( m_{RHS} \), and finally, it is added to the set of inserted objects \( \Delta V_M \), formally,
  \[
  \forall \text{node}' \in V_{RHS} \setminus V_{LHS} : new(\text{obj}_{new}) \land instanceOf(\text{node}', A) \land instanceOf(\text{obj}_{new}, A) \land A \in MM \land m_{RHS}(\text{node}') = \text{obj}_{new} \land \text{obj}_{new} \in \Delta V_M^+
  \]

  For easier definition we use the \( new \) function that represents the concept that a new object is picked from the unused model element universe \([Var08] \).

- All inserted nodes are added to \( M \), formally, \( V_{M'} = V_M \cup \Delta V_M^+ \).

- Each mapped edge \( \text{edge}' \) of RHS is mapped by match \( m_{RHS} \) to the same link that has been assigned to its origin edge \( \text{edge} \) by match \( m \), formally, \( \forall \text{edge}', \text{edge} \in E_{LHS} \cup E_{RHS} : p(\text{edge}) = \text{edge}' \Rightarrow m_{RHS}(\text{edge}') = \text{edge} \)

- For each unmapped edge \( \text{edge}' \) of RHS, a new link \( \text{link}_{new} \) is created with the same type as of \( \text{edge}' \) and it is assigned to \( \text{link}' \) by match \( m_{RHS} \) is a source and target preserving way, and finally, it is added to the set of inserted links \( \Delta E_M^+ \)
  \[
  \forall \text{edge}' \in E_{RHS} \setminus E_{LHS} : new(\text{link}_{new}) \land instanceOf(\text{edge}', A) \land instanceOf(\text{link}_{new}, A) \land A \in MM \land m_{RHS}(\text{edge}') = \text{link}_{new} \land \text{link}_{new} \in \Delta E_M^+
  \]

- When all the inserted links are collected, they are added to model \( M \), formally,
  \[
  E_{M'} = E_M \cup \Delta E_M^+.
  \]

- Finally, the attribute manipulation operation are executed on the resulting \( M' \)
3.4. SUMMARY

**Figure 3.6: Application of the return GT rule**

**Definition 11** Given a metamodel $MM$, and a match $m$ for a rule $r$ in model $M$, **rule $r$ is applied to the match $m$ in the model $M$ resulting in $H$** (denoted by $G \xrightarrow{r,m} H$), if the updating phase first executes the deletion and then the insertion sub-phases and finally the attribute manipulation operation resulting in $H$ a well-formed instance of metamodel $MM$.

It is important to mention that in this definition, the well-formedness of the derived model is prescribed as a right application condition [EEPT06]. In practice, this means that effects of rule application have to be rolled back, if the resulting model is not well-formed e.g., due to multiplicity constraint failure, the existence of parallel edges of the same type or attribute manipulation errors.

**Example 10** An example application of the return GT rule on a small extract of the AntHill model is depicted in Figure 3.6. The Ant, OuterNeighbor, InnerNeighbor pattern nodes of the LHS of the return GT rule are matched to the CA1 CarrierAnt, CF2 and CF1 CornerField objects of the model, respectively. Similarly, the OldHA returnPath and RP returnPath relations of the LHS pattern are matched to the H1 and R1 links of the model.

Following the pattern matching phase of the GT rule application in the updating phase the $H1$ hasCarrierAnt link is deleted and a new $H2$ hasCarrierAnt link is created between the CA1 and CF1 objects. This way the CA1 ant moved from the CF2 field to the CF1 field.

For easier readability the elements of the LHS and RHS pattern bodies are connected to their matching model elements by dotted grey lines. The deleted $H1$ relation is highlighted in red, while the newly created $H2$ relation in green.

3.4 Summary

In this chapter, the concepts of graph patterns and transformation rules have been presented as a declarative rule-based specification language for querying and manipulating graph models, respec-
tively, along with their precise formalization. These concepts have been exemplified on our AntWorld running example.
Part II

Advances in Graph Pattern Matching
Introduction

While nowadays model-driven system development is being supported by a wide range of conceptually different model transformation tools, nearly all of these tools have to solve a common problem: the efficient query and manipulation of complex graph-like model structures that in certain scenarios may range in the hundreds or even million elements [13].

Tools based on the rule and pattern-based formal paradigm of graph transformation (GT) [Roz97, EKR99] already integrate research results of several decades. In these tools, a matching of the left-hand side (LHS) of a graph transformation rule is being sought by some graph pattern matching algorithm, which might be invalidated by valid matchings of negative application conditions (NAC).

Graph pattern matching leads to the subgraph isomorphism problem that is known to be NP-complete in general [Ata99], which means that highly time-consuming computations are expected for the worst-case scenario from theoretical aspects. However, practical model transformation problems rather have a regular and sparse graph structure, which drastically reduces the execution time of graph pattern matching [Var08]. The two most widely applied approaches in all the major graph transformation tools are based on:

- **Local search based** or sometimes referred as **search plan driven** approaches (like PROGRES [Zün96], Dörr’s approach [Dör95], FUJABA [NNZ00] or GReAT [KASS03]). In all of these approaches, pattern matching is driven by a search plan, which provides an ordering for traversing and matching nodes and edges of a graph pattern.

- **Incremental pattern matching** [VVS06b, BOR+08, MMS07, HLR06, MMLA10] using extensive caching of partial matching and updating the caches while listening to the model changes.
CHAPTER 4. INTRODUCTION

4.1 Motivation: Hard Coded Heuristics for Local Search Based Pattern Matching

Research on LS based approaches [Zün96, GSR05, GBG+06, VVF05] has been focusing mainly on the performance optimal ordering of elementary pattern matching operations like (i) the enumeration of objects and links of a certain type, (ii) the navigation along links of a given type, and (iii) the existence checks for links. On the other hand, the ordering of (iv) attribute, (v) injectivity and (vi) NAC constraint checking operations has been hard wired into the graph transformation engines by using some simple heuristics.

For instance, in case of NAC checking operations, two wiring strategies are known in GT tools. The “as soon as possible” (ASAP) style positioning (used by Fujaba) places the NAC checking operation to the first possible location where all its arguments are bound. Intuitively, when the size of the NAC pattern is small compared to the unexplored part of the LHS pattern, a quickly retrieved match for the NAC may significantly reduce the search space by avoiding the unnecessary traversal of the remaining part of the LHS. On the other hand, when the NAC is large, the corresponding check operation can be time consuming, so a delayed execution may provide better overall performance for the pattern matching of the LHS. This idea is implemented by the “as late as possible” (ALAP) strategy (used by PROGRES), which executes NAC checking only when a complete matching for the LHS has been found.

These best engineering practices are acceptable for performing cheap checks like checking attribute and injectivity constraints, but when a single search plan operation represents a complete pattern matching process like in case of checking a NAC or calling native external libraries (as in AGG or ViATRA2), hard-wired positioning may cause performance degradation as (a) it lacks flexibility and extensibility and (b) it ignores the complexity of the actual search plan operation. However, checking NAC is critical to model transformation problems in order to forbid multiple application of a rule on the same matching.

This is a common situation in the case of model-specific search plans [VVF05, GBG+06] where the cost of search plan operations depends on the actual graph being transformed. However, even in the case of (traditional) metamodel-specific search plans (like in FUJABA, GREaT or PROGRES), the bindings of input parameters of rules may have a huge impact on the optimal ordering of complex search plan operations. Intuitively, if many input parameters are passed to a rule, the ASAP strategy can be too expensive for complex NACs.

Contribution to Local Search Based Pattern Matching

In order to address this problem, we propose a general framework for uniformly representing a large variety of search plan operations by expressing them as cost-weighted predicates. As an appropriate ordering of these predicates defines an executable search plan, this approach is able to uniformly guide the pattern matching process for advanced graph patterns regardless of how we assign the actual costs to different search plan operations. As a result, better performance is expected, especially, for checking negative application conditions, which avoids the previous problems.

The main practical advantages of our approach are modularity, flexibility, and extensibility. The different phases of pattern matching (e.g. cost assignment, generation of search plans, execution of search plans etc.) are fully separated and independent, thus they can be adapted to very different graph transformation engines and strategies (metamodel-based vs. model-based search plans). Furthermore, new types of predicates can be introduced easily by assigning appropriate costs without altering the algorithms for search plan generation.
4.2 Motivation: Combining Different Pattern Matching Approaches

As an alternative, incremental pattern matching (INC) approaches have recently become a popular approach in the model transformation community. The core idea is to improve the execution time of the time-consuming pattern matching phase by additional memory consumption. Essentially, the (partial) matches of graph patterns are stored explicitly, and these match sets are updated incrementally in accordance with elementary model changes. While model manipulation becomes slightly more complex, all matches of a graph pattern can be retrieved in constant time in exchange by eliminating the need for recomputing existing matches.

Initial benchmarking \cite{17,2} has shown that in many scenarios, the incremental pattern matching approach (as implemented in the VIATRA2 framework) leads to orders-of-magnitude increases in speed. However, an important implication of caching match sets is increased memory consumption, which needs to be taken into account when scaling up to large models. Unfortunately, in many practical applications of model transformations, available memory is frequently constrained (e.g., when they are executed on average desktop computers and not on high performance servers).

Contribution to Hybrid Pattern Matching

To overcome this limitation, we propose a hybrid pattern matching approach which enables the transformation designer to combine local search based and incremental pattern matching to adapt to memory constraints. At design-time, transformation engineers may select whether a graph pattern should be matched using the LS or the INC strategy separately for each pattern. Moreover, based upon runtime monitoring, the execution engine may automatically switch from incremental pattern matching to local-search based technique when a certain memory limit has been reached.

However, selecting the appropriate matching strategy for complex model transformation programs requires a deep understanding of both pattern matching algorithms. Therefore, we examined typical transformation scenarios from the literature \cite{VSV05,24,17,27}. As a result, we found a list of various factors (metrics) \cite{2}, which we experienced to have significant effect on run-time performance and memory consumption. Based on this analysis, we defined guidelines for transformation designers when a graph pattern should be matched using INC or LS algorithm.

The main highlight of our approach, compared to other adaptive pattern matching algorithms \cite{VVF05,JBK10,VDWS12}, is the ability to combine conceptually different matching strategies. With each of them optimal for different scenarios, the transformation designer is able to fine-tune the performance of the transformation engine with regard to execution speed as well as size scalability.

4.2.1 Structure

The current part of the thesis is structured as follows.

- Chapter 5 introduces the concepts of search plan driven pattern matching by defining the concepts of search graphs (in Section 5.1) and search plans (in Section 5.2), followed by the algorithms used for their generation from graph patterns in Section 5.3. Finally, details on how the proposed approach was realized in the VIATRA framework is discussed in Section 5.4.

- Chapter 6 gives an overview on how hybrid pattern matching can be introduced to our search graph driven pattern matching concept. In Section 6.1 the basics of RETE based incremental pattern matching is discussed, followed by Section 6.2 defining the extensions to our search graph concept to support hybrid pattern matching. The benchmark evaluation results on our
AntWorld case study using the introduced hybrid approach is discussed in Section 6.3. Moreover, key metrics for selection strategies between LS and INC approaches are presented in Section 6.4.

- Finally, related work and conclusions are discussed in Chapter 7.
Search Plan Driven Graph Pattern Matching

In the last decades many local search based pattern matching algorithms ([Zün96, Dör95, GHS09] or [18]) have been developed. Within the graph transformation community, all of these are variant of the well-known Ullmann [Ull76] algorithms. Usually, the implementations of LS based graph pattern matching algorithms differs from Ullman’s work in how the candidates are computed and the extended matches are checked.

The generation of search plans [Zün96] is a frequently used and efficient strategy to drive the execution of these local search based pattern matching algorithms. Informally, a search plan defines an order of pattern nodes, in which they are bound to objects of the instance model during pattern matching. In addition to simply specifying the binding order of pattern nodes, it often also includes an order of elementary operations that have to be executed to drive pattern matching.

In the current thesis, we understand search plan driven pattern matching as a two phase process [Var08]:

- At compilation time a search graph is constructed for each graph pattern bodies. It is a joint representation of pattern graph elements and operation constraints that drives the pattern matching process.

- At runtime, when patterns are invoked with bound input parameters, the search graph is adorned, which denotes that the a given pattern node or edge is initially bound or free. Based on this adorned search graph, a search plan is generated that is one possible traversal of the search graph. Finally, this traversal defines a totally ordered list of search operations, which represents the atomic units of pattern matching.
5.1 Search Graphs

A search graph is a joint representation of pattern body graph elements and operation constraints that drives the pattern matching process. In our interpretation, a search graph is a hypergraph representing a constraint net, where graph nodes reflect variables, and hyperedges express constraints (predicates, similarly to Datalog [AV88]) between the variables. A search graph is directly derived from the pattern body graph as follows:

- **Pattern variable**: Each element (node or edge) of the pattern graph is mapped to a pattern variable. These elements represent the arguments of the constraints. This uniform representation allows to support any kind of operation constraint between edges and nodes of the pattern body. This may be restricted if the underlying metamodeling language does not support this kind of freedom. For the ease of understanding, we will not make any restrictions on operation constraints on edges.

  There is a subset of pattern variables called **constants**. They represent elements from the metamodel on which the graph pattern is defined.

- **Operation Constraint**: Each constraint on the pattern graph (e.g., connectivity, instanceOf, injectivity etc.) is mapped to an n-ary (usually binary) edge of the search graph. The edge connects elements that are part of the constraint it defines (e.g., a source constraint connects the pattern variables of the source node and the edge). They represent operation predicates, that have to be fulfilled during the matching process.

**Example 11** The search graph of the first pattern body of the `moveAnywhereButHome` (see in Figure 3.1(e)) GT rule is illustrated in Figure 5.1.

![Search graph of the first pattern body of the moveAnywhereButHome graph rule's precondition](image)

Figure 5.1: Search graph of the first pattern body of the moveAnywhereButHome graph rule’s precondition

The search graph contains nine pattern variables; `SearcherAnt`, `Path`, `hasSearcherAnt` and `Field` represent the type elements (constants) of the pattern graph (denoted by black ovals), while variables `Ant`, `OldHasAnt`, `Field1` and `Field2` represent the nodes and edges of the pattern graph itself (denoted by black ovals). The operation predicates directly define the constraints of the pattern graph: `src`, `trg` edges define the source and target node of an edge (denoted by green rectangles). For example, `Field1` is the source of edge `P` and its target is `Field2`. The `inst` edges represent the direct
instance of relations between a pattern variable and its type constant. For example as between the 
P pattern variable and its path type. The nac1 edge represents the negative application condition 
constraint with its single input parameter Field2. Finally, the two inj edge define the injectivity check 
between their input variables like between the two edges OldHasAnt, P. Note that the inj constraint 
for common sense have been defined separately between the edges and nodes of the search graph.

**Definition 12** Given a metamodel MM and a graph pattern GP with a pattern body PB, the search 
graph $SG = (V_{SG}, E_{SG}, b)$ is a hypergraph with nodes $V_{SG}$ and edges $E_{SG}$ and a backward mapping 
b : $SG \rightarrow PB$, which maps nodes and edges of the search graph $SG$ to the pattern body. The 
structure of the graph is described by the following rules:

- **Pattern variable:** Nodes $V_{SG}$ of the search graph can be partitioned into two sets: (i) the 
constant node set that contains all elements that are constant in the pattern body $V_{SG}^{const}$ and (ii) the 
$V_{SG}^{PV}$ pattern variables. Formally, $V_{SG} = V_{SG}^{const} \cup V_{SG}^{PV}$ and $V_{SG}^{const} \cap V_{SG}^{PV} = \emptyset$. The mapping 
rules are the following:

  - Each type element type of the pattern body PB is mapped to a constant node $x$ in the 
search graph. Formally, $\forall instanceOf(A, x) \in SCP_B, \exists x \in MM, \exists type \in V_{SG}^{const}, : 
b(x) = type.$
  - Each node $n$ of the PB pattern body is mapped to a pattern variable $var$ in the search 
graph. Formally, $\forall n \in V_{SCP_B}, \exists var \in V_{SG}^{PV} : b(var) = n.$
  - Each edge $e$ of the PB pattern body is mapped to a pattern variable $var$ in the search 
graph. Formally, $\forall e \in E_{SCP_B}, \exists var \in V_{SG}^{PV} : b(var) = e.$

- **Operation Constraint:** Edges of the search graph can be partitioned into: (i) source edges $E_{src}^{SG}$, 
(ii) target edges $E_{trg}^{SG}$, (iii) instance of edges $E_{inst}^{SG}$, (iv) injectivity check edges $E_{inj}^{SG}$, (v) nac 
check edges $E_{NAC}^{SG}$, and finally (vi) attribute condition check edges $E_{attr}^{SG}$. Formally, 
$E_{SG} = E_{src}^{SG} \cup E_{trg}^{SG} \cup E_{inst}^{SG} \cup E_{inj}^{SG} \cup E_{NAC}^{SG} \cup E_{attr}^{SG}$ and $\forall i, j \in \{ src, trg, inst, inj, NAC, attr \}, i \neq j : E_{SG} \cap E_{SG} = \emptyset$. The mapping rules for the edges are the following:

  1. Simple predicates represent core constraints between two pattern variables.

    - Each edge $e$ connecting the node $u$ to node $v$ in the pattern body is mapped to a pair of 
source src$(x, y)$ and target trg$(x, y)$ edges in the search graph connecting the 
corresponding pattern variables of $u, v$ to the pattern variable of $e$. Formally,

      $\forall e \in E_{SCP_B}, src_{SCP_B}(e) = u, \exists src(pvar_e, pvar_u) \in E_{SG}^{src} :$
      
      $b(src(pvar_e, pvar_u)) = e \land b(pvar_e) = e \land b(pvar_u) = u \land pvar_u, pvar_e \in V_{SG}^{PV}$

      $\forall e \in E_{SCP_B}, trg_{SCP_B}(e) = v, \exists trg(pvar_e, pvar_v) \in E_{SG}^{trg} :$
      
      $b(trg(pvar_e, pvar_u)) = e \land b(pvar_e) = e \land b(pvar_v) = v \land pvar_v, pvar_e \in V_{SG}^{PV}$

    - Each instanceOf relationship between the node $u$ and type $type$ is mapped to an 
instance of edge inst$(pvar_u, pvar_type)$ in the search graph connecting the 
corresponding pattern variables of $u$ and $type$. Formally, 

      $\forall instanceOf(u, type) \in SCP_B, \exists inst(pvar_u, pvar_type) \in E_{SG}^{src} :$
CHAPTER 5. SEARCH PLAN DRIVEN GRAPH PATTERN MATCHING

(b\((\text{inst}(p\text{var}_u, p\text{var}_\text{type}))) = \text{instanceOf}(u, \text{type}) \land b(\text{pvar}_u) = u \\
\land b(\text{pvar}_\text{type}) = \text{type} \land \text{pvar}_u \in V^\text{PV}_{SG} \land \text{pvar}_\text{type} \in V^\text{const}_{SG})

2. Complex predicates are defined between an arbitrary number of pattern variables.

- One-one \(\text{inj}(\text{pvar}_1, \ldots, \text{pvar}_n)\) \textit{injectivity (hyper)edge} is created between all pattern variable representing nodes \((\text{inj}_{\text{node}})\) and edges \((\text{inj}_{\text{edge}})\) of the pattern body. Formally, \(\forall n \in V_{SC_{PB}} \exists \text{inj}_{\text{node}} \in E^\text{inj}_{SG} : b(\text{pvar}_n) = n \land \text{pvar}_n \in \text{inj}_{\text{node}}\) and \(\forall e \in E_{SC_{PB}} \exists \text{inj}_{\text{edge}} \in E^\text{inj}_{SG} : b(\text{pvar}_e) = e \land \text{pvar}_e \in \text{inj}_{\text{edge}}\).

- Each negative application condition \(\text{NAC}_i = (\text{GP}_{\text{NAC}_i}, \text{PNAC}_i)\) of the \(\text{PB}\) pattern body is mapped to a \textit{NAC (hyper)edge} \(\text{nac}_i(\text{pvar}_1, \ldots, \text{pvar}_n)\) in the search graph, where each corresponding \(\text{pvar}_u\) pattern variable of node or edge \(u_i\) of the structural constraint that is shared with the negative application condition pattern (as defined by \(\text{PNAC}_i\)), is part of \(\text{nac}_i\). Formally, \(\forall \text{NAC}_i, \exists \text{nac}_i \in E^\text{NAC}_{SG} : b(\text{nac}_i) = \text{NAC}_i\) and \(\forall j = 1 \ldots n, u_j \in (V^\text{PB}_{SG} \cup E^\text{PB}_{SG}), \exists \text{PNAC}(u_j) : \exists b(\text{pvar}_u_j) = u_j \land u_j \in \text{nac}_i\).

- Each attribute constraint \(\text{AC}_i\) of the \(\text{PB}\) pattern body is mapped to an \textit{attr (hyper)edge} \(\text{a}_i(\text{pvar}_1, \ldots, \text{pvar}_n)\) in the search graph, where each corresponding \(\text{pvar}_i\) pattern variable of node \(u_i\) of the attribute constraint is part of \(a_i\). Formally, \(\forall \text{AC}_i, \exists a_i \in E^\text{attr}_{SG} : b(a_i) = \text{AC}_i\) and \(\forall j = 1 \ldots n, u_j \in \text{AC}_i : b(\text{pvar}_u_j) = u_j \land u_j \in a_i\).

5.2 Adorned Search Graphs and Search Plans

\textit{Adorned search graph} is a search graph, in which pattern variables are adorned depending on the initial binding of their pattern node origin (see in [Ull89]). The set of pattern variables that represent pattern nodes that are already matched when the pattern matching starts are called \textit{bound nodes (B)}, while the initially unmatched pattern variables are called \textit{free nodes (F)}.

A \textit{search plan} is a sequence (totally ordered list) of the edges of the search graph. Based on this sequence a search plan also defines elementary \textit{search operations} to precisely specify the operations needed to be executed in the pattern matching process. A search operation consists of a constraint and an adornment. A search operation is either an \textit{extend type} operation which extends the matching by generating matching candidates for a \textit{free node} (e.g., match the target node along an edge), or a \textit{check type} operation used for checking constraints between bound pattern variables (e.g., whether an edge runs between two nodes).

\textbf{Example 12} An adorned search graph of the first pattern body of the \textit{moveAnywhereButHome} (see in Figure 3.1(e)) with the \textit{Ant} pattern variable as an input bound parameter is illustrated in Figure 5.2(a).

The adornment of the pattern variables are illustrated by white \(F\) and \(B\) characters written on the grey ovals representing the variables, for example, the \textit{Ant} pattern variable is bound at the start of the pattern matching process and thus has an \(F\) written on its representing pattern variable. The tables next to the operations define the cost of each search operations which will be described later in Section 5.3.1.

A possible search plan generated from the adorned search graph is depicted in Figure 5.2(b). It defines the sequence of elementary search operations that are needed to be executed in order to match the pattern body. Basically, the process starts from the \textit{Ant} pattern variable and extend the
5.2. ADORNED SEARCH GRAPHS AND SEARCH PLANS

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst(^{BB})(Ant, SA)</td>
<td>check</td>
<td>Ant is instance of SA</td>
</tr>
<tr>
<td>trg(^{FB})(OHA, Ant)</td>
<td>extend</td>
<td>target of OHA is Ant</td>
</tr>
<tr>
<td>inst(^{BB})(OHA, hSA)</td>
<td>check</td>
<td>OHA is instance of hSA</td>
</tr>
<tr>
<td>src(^{BF})(OHA, F(^1))</td>
<td>extend</td>
<td>source of OHA is F(^1)</td>
</tr>
<tr>
<td>inst(^{BB})(F(^1), P)</td>
<td>extend</td>
<td>F(^1) is instance of Field</td>
</tr>
<tr>
<td>src(^{BF})(P, OHA)</td>
<td>check</td>
<td>P is instance of Path</td>
</tr>
<tr>
<td>inst(^{BB})(P, Path)</td>
<td>check</td>
<td>OHA not equals to P</td>
</tr>
<tr>
<td>trg(^{BF})(P, F(^2))</td>
<td>extend</td>
<td>source of P is F(^2)</td>
</tr>
<tr>
<td>inst(^{BB})(F(^2), F)</td>
<td>check</td>
<td>F(^2) is instance of Field</td>
</tr>
<tr>
<td>nac(^1)(F(^2))</td>
<td>check</td>
<td>nac(^1) check</td>
</tr>
<tr>
<td>inj(^{BBB})(Ant, F(^1), F(^2))</td>
<td>check</td>
<td>Ant != F(^1) != F(^2)</td>
</tr>
</tbody>
</table>

**Figure 5.2:** An adorned search graph and a possible search plan where Ant is a bound input parameter

match through the edges up to the Field2 (in the table abbreviations are used for easier readability) and executing the necessary check operation along the way. Finally, it checks the nac condition defined on the Field2 node and the injectivity constraint between the pattern variables representing a node in the pattern body.

**Definition 13** An adorned search graph \(ASG\) is a run-time representation of a search graph \(SG\), in which pattern variables are further partitioned into bound nodes \(V^B_{SG}\) and free nodes \(V^F_{SG}\) depending on whether the corresponding node of the \(PB\) pattern body is already bound when the pattern matching starts. The pattern variables in the constant subset are always considered bound. Formally,

\[ V^P_{SG} = V^B_{SG} \cup V^F_{SG}, \text{ and } V^B_{SG} \cap V^F_{SG} = \emptyset \text{ and } V^\text{const}_{SG} \subseteq V^B_{SG}. \]

**Definition 14** Given an adorned search graph \(ASG\), a search plan \(SP = (e_1, e_2, \ldots, e_{|E_{ASG}|})\) is an ordered list (sequence) of the adorned search graph operation constraints (edges). Formally, \(SP = (e_1, e_2, \ldots, e_n)\) where \(n = |E_{ASG}| \land \forall j = 1 \ldots n : e_j \in E_{ASG}.\)

By definition, a search plan specifies the order of the operation constraints, additionally, a search plan may also define elementary search operations and order them to completely and precisely define the process of pattern matching. The concrete operation a constraint represents is based on the adornment of its nodes. For example, a source navigation constraint with an \(FB\) or \(BF\) adornment on its nodes represents an extend operation, where it needs to generate candidates for the free variable, while in case of \(BB\) adorned nodes it represents a check edge operation, which simple checks that there is an edge between the nodes.
Simple operations are the ones that are based on simple constraints. The meaning of letter $B$ in an operation adornment is that the variable is bound to a value in that position, while $F$ means that it is free. Formally, $\forall i = 1 \ldots m, n_i \in e : (n_i \in V_{SG}^B \implies ad[i] = 'B') \lor (n_i \in V_{SG}^F \implies ad[i] = 'F')$.

**Definition 16** A search operation $so = (e, ad)$ of the search plan $SP_{P_B}$ consists of an operation constraint $e(n_1, n_2, \ldots, n_m) \in SP$ and a valid operation adornment $ad$. These two together represent a single atomic step in the pattern matching process for the $PB$ pattern body.

Though $2^n$ different adornments can be assigned to each n-ary operation constraint in theory, only a subset of these adornments are used, which respect elementary complexity consideration. In our case, for the simple constraints the permitted operation adornments usually are $FB; BF; BB$, while $FF$ represents a far too expensive operation, as we need to cumulate all pairs of elements in the model.

For complex constraints only the $B \ldots B$ adornments are permitted, as all variables must be bound to an element in case of injectivity checking, NAC checking and Boolean term evaluation.

**Example 13** In Figure 5.2(a), permitted adornment values are illustrated with small tables near the constraints. The table has two columns, which show the operation adornments and the operations cost (which is discussed in Section 5.3.1), respectively. For a source operation constraint between the $oldHasAnt$ and the $Field1$ pattern variable the $FB, BF, BB$ operation adornments are allowed.

**Search Operations in Adorned Search Graph Driven Pattern Matching**

Based on the operation constraints defined in Section 5.1, we defined the following search operations for a match $m$ (where $m(x)$ represents a candidate mapping for the $x$ pattern element of the pattern body) based on a search plan $SP$:

**Simple operations** are the ones that are based on simple constraints.

- Operations for the **source** constraint based on its operation adornment:
  - $src^{FB}(pvar_x, pvar_y)$ represents a navigation in backward direction from $m(y)$ to $m(x)$. Possible candidates for the mapping of pattern node $x$ are such links that have $m(y)$ as their source $src(m(x)) = m(y)$.
  - $src^{BF}(pvar_x, pvar_y)$ represents a navigation in forward direction from $m(x)$ to $m(y)$. Possible candidates for the mapping of pattern node $y$ are such objects that are the source of $src(m(x)) = m(y)$.
  - $src^{BB}(pvar_x, pvar_y)$ represents a check edge existence operation prescribing the existence of a link $m(x)$ where the source of the link is $m(y)$ and $src(m(x)) = m(y)$.

- Operations for the **target** constraint based on its operation adornment:
  - $trg^{FB}(pvar_x, pvar_y)$ represents a navigation in backward direction from $m(y)$ to $m(x)$. Possible candidates for the mapping of pattern node $x$ are such links that has $m(y)$ as their target $trg(m(x)) = m(y)$.
  - $trg^{BF}(pvar_x, pvar_y)$ represents a navigation in forward direction from $m(x)$ to $m(y)$. Possible candidates for the mapping of pattern node $y$ are such objects that are the target of $trg(m(x)) = m(y)$.
5.3. SEARCH PLAN GENERATION

- \( \text{trg}^{BB}(\text{pvar}_x, \text{pvar}_y) \) represents a check edge existence operation prescribing the existence of a link \( m(x) \) where the target of the link is \( m(y) \) and \( \text{trg}(m(x)) = m(y) \).

- Operations for the \( \text{instanceOf} \) constraint based on its operation adornment:
  - \( \text{inst}^{FB}(\text{pvar}_x, \text{pvar}_y) \) represents an iteration by all elements of a type operation. Possible candidates for the mapping of pattern node \( x \) are such elements (objects or links) that are instances of \( \text{type} \), formally, \( \text{instanceOf}(m(x), \text{type}) \).
  - \( \text{inst}^{BB}(\text{pvar}_x, \text{pvar}_y) \) represents a check type operation prescribing that \( m(x) \) is an instance of \( \text{type} \).

Complex operations can only be executed if all of their input parameters are bound.

- \( \text{inj}^{BB...B}(\text{pvar}_x_1, \text{pvar}_x_2, \ldots, \text{pvar}_x_n) \) prescribes an injectivity check operation which checks that neither of its input are mapped to the same objects or link in the instance model, formally, \( \forall j = 1 \ldots n, j = 1 \ldots n, : m(x_i) \neq m(x_j) \).

- \( \text{nac}^{BB...B}(\text{pvar}_x_1, \text{pvar}_x_2, \ldots, \text{pvar}_x_n) \) prescribes the execution of a full-featured pattern matching for the NAC pattern with the \( m' \) initial matching, where all shared pattern elements from \( m \) are bound in \( m' \), formally, \( \forall m'(x_i), i = 1 \ldots n, \exists m(y) \neq \text{null} : m'(x_i) = m(y) \). Note that the \( \text{nac} \) check operations return value is either \( \text{true} \), which means that there is no match for the NAC pattern and thus the original pattern matching can proceed, or a \( \text{false} \) meaning that there is a match for the NAC pattern and thus the original pattern matching fails and the pattern matcher has to search for other candidate objects.

- \( \text{attr}^{BB...B}(\text{pvar}_x_1, \text{pvar}_x_2, \ldots, \text{pvar}_x_n) \) prescribes a term evaluation operation on the attribute constraint defined by the pattern body.

Note that the defined operations can be easily extended by simply defining new types of edges for the search graph and define their corresponding operation based on the their bound and free pattern variables. Additionally, the way we defined the traversal of a simple edge in the instance graph (separated into two operation \( \text{src} \) and \( \text{trg} \)) allows to treat edges similarly to nodes thus if the metamodel would allow edges between edges the proposed search graph could be used without any modifications.

5.3 Search Plan Generation

At this point, we have the complete set that we are using to generate search plans (as an ordered list of search operations). However, in order to achieve effective pattern matching we have to be able to select a search plan that requires the smallest search space tree to be traversed during pattern matching.

A search space tree (STT) – as defined by Gergely Varró [Var08] – is an organized way to represent how the search plan tries to find a valid match. It contains all the decisions that can be made at a certain point during pattern matching. Its root node represents a partial matching as provided by fixing the input parameters of the graph pattern. Each path of a SST starting from the root node extends this partial matching by the matching of a fresh (unmatched) node in the graph pattern. A valid match in the STT is a leaf that is on the \( k^{th} \) level of the tree, where \( k \) is the number of free pattern variables in its adorned search graph (the elements that are needed to be matched).
To achieve good performance the assignment of a cost to each search plan is a widely accepted approach. If this cost is in strong correlation with the size of the search space tree, then the execution of the minimum-cost search plan yields the fastest algorithm variant for pattern matching. This fact calls the attention to the importance of search plan generation algorithms that aim at finding a minimum-cost search plan in a given search graph. Note that an overall speed-up in execution time is only acceptable if the execution of a better search plan can compensate the additional time spent on the generation of the plan.

5.3.1 Cost of Search Plans

The current section (based on [Var07]) highlights the most frequent measures being used for characterizing search plans, and their corresponding search plan generation algorithms. In order to have a uniform notation, let \( w_k \) denote the weight of the \( k \)th operation according to the order defined by the search plan. Let us further suppose that the search plan consists of \( n \) operations.

- **Sum of weights.** The first and most intuitive approach is to use the sum of weights of the edges that comprise the search plan, formally, \( w_{\sum}(SP) = \sum_{j=1}^{n} w_j \). Its main advantage is that the Chu-Liu / Edmonds algorithm [CL65, Edm67] can quickly generate minimum-cost search forests as the algorithm has a time complexity \( \Omega(ne) \), and search graphs have at most a few dozen nodes and edges in a typical application scenario. However, the cost of a forest is completely insensitive to the different orderings of its tree edges, thus, sum-based cost functions provide a poor estimate for the size of the search space tree, which means that even the minimum-cost search plan does not necessarily lead to a fast pattern matching process.

- **Product of weights.** The implementation of the GT tool being presented in [JBK10] uses the product of weights as a cost function \( w_{\prod}(SP) = \prod_{j=1}^{n} w_j \). However, this is quite similar to the above mentioned approach as by taking the logarithm of the cost, we get the sum of the logarithms of weights. This cost function gives a better estimate for the size of the search space tree, but it is still highly insensitive to the different orderings of the search forest edges. Similar algorithms can be used to calculate the search plan as mentioned before.

- **Sum of Products.** In [VVF05, Var08] Gergely Varró proposed to calculate the cost function as \( w_{\text{sum}}(SP) = \sum_{i=1}^{n} \prod_{j=1}^{i} w_j \), which is a correct estimation for the size of the search space tree, if weights of search graph edges denote branching factors, which are collected from the actual model on which graph transformation is performed. The main drawback of this technique is the lack of algorithms, which could find a minimum-cost search plan according to this special cost function. Varró proposed to use again the Chu / Edmonds algorithm and argued that in practical scenarios it provides acceptably low cost for the computed search plans. However, it is important to mention that in [VDWS12] a dynamic programming based algorithm is proposed that in certain scenarios theoretically is capable of calculating the optimal solution for his own cost function.

For the weight of the various operations we use a simple metamodel based approach. Weighting the simple operation follows the guidelines of edge multiplicity based cost functions (e.g., if an edge multiplicity is one-to-many, then its cost is higher then if it is one-to-one) with the following restriction: the lowest cost is assigned to the \( BB \) adornments (check type operation), and there is no dif-
5.3. SEARCH PLAN GENERATION

ference between the cost of FB and the BF (extend type operation). Among the constraints we use the cost ordering based on our earlier transformation experiments [27] and [17]: trg = src < inst.

In case of complex constraints, assigning costs to operations is easier on the one hand as they have only one permitted adornment $B \ldots B$, but on the other hand better cost prediction is possible using a priori knowledge. In case of inj and attr constraints the number of input parameters provides a good prediction for complexity, while in case of a nac constraint the whole NAC pattern graph matching cost can be evaluated at compile time. The cost functions are the following: (i) for inj and attr operations the cost function is linear in the number of parameters and (ii) for nac operation the cost function is proportional to the number of constraints in the search graph of the NAC pattern. The idea behind this selection is that a NAC check may cut the search space significantly when its pattern is small.

In our realization of the adorned search graph driven pattern matching (implemented in Via-tra2 and described in Section 5.4) we opted for the Sum of Products function as it provides a good estimation on the runtime characteristics of the generated search plans.

Example 14 In our running example, we defined explicit operation cost values (see in Figure 5.2(a)) for all operations of the example adorned search graph. The concrete number values are only used to illustrate the order of magnitude of operation costs. For example, the inst constraint for the $P$ pattern variable either has the cost of one in case of two bound pattern variables as it is only a simple type check operation, while if the edge has $P$ as a free pattern variable than it is an extend type operation where $P$ will be mapped to all path typed link of the instance model. The complete cost using the Sum of Products function is calculated as follows:

$$w_{sum}(SP) = 1 + 1 \cdot 5 + 1 \cdot 5 \cdot 1 + 1 \cdot 5 \cdot 1 \cdot 5 + 1 \cdot 5 \cdot 1 \cdot 5 \cdot 1 + 1 \cdot 5 \cdot 1 \cdot 5 \cdot 1 \cdot 5 \cdot 1 \cdot 5 + 1 \cdot 5 \cdot 1 \cdot 5 \cdot 1 \cdot 5 \cdot 1 \cdot 3 + 1 \cdot 5 \cdot 1 \cdot 5 \cdot 1 \cdot 3 \cdot 5 + 1 \cdot 5 \cdot 1 \cdot 5 \cdot 1 \cdot 3 \cdot 5 \cdot 1 \cdot 3 \cdot 5 \cdot 1 \cdot 3 + 1 \cdot 5 \cdot 1 \cdot 3 \cdot 5 \cdot 1 \cdot 3 \cdot 5 \cdot 4 = 32625.$$ 

5.3.2 Algorithm for Finding a Low Cost Search Plan

For generating the actual search plans, we applied a slightly modified version of the Chu/Edmonds algorithm as described in [VVF05]. Two traditional greedy algorithms are used to solve the problems of finding (i) a low cost search tree for a given adorned weighted search graph and (ii) a low cost search plan for a given search tree.

Finding a minimum search tree.

For finding a minimum search tree in a weighted adorned search graph, the Chu-Liu / Edmonds algorithm [CL65, Edm67] is used. This algorithm searches for a spanning tree in a directed graph that has the smallest cost according to a cost function defined as the sum of weights.

To take into account the practical side we adopted the algorithm to our hypergraph representation using the following considerations:

- Simple edges are modeled using two references between the nodes, thus if any operation is executed on a reference its opposite reference also automatically receives the same operation.

- Hyperedges are modeled as a node which has specific references to the nodes of the hyperedge. In any case one of these specific references are selected for any operation during the search plan generation the algorithm automatically executes the operation on all of the references of the hyperedge.
• The circle detection algorithm has to be slightly modified to be able to handle our hyperedge representation. Any time a hyperedge is selected all of its nodes are added to the circle candidate set and the search continues on all these nodes.

• The weight values of the edges are based on the following simple rules (i) edges between bound input values or constants are considered with their all bounded weight and (ii) all other cases are calculated as $FB$ or $BF$, which for this reason has the same weight values for all simple operations.

• The starting node is either a bound input pattern variable or a constant.

Using these considerations the algorithm is outlined in Algorithm 5.1.

Algorithm 5.1 The Used Variant of the Chu-Liu / Edmonds Algorithm

The input is an adorned weighted search graph with a selected starting node.

Step 1: Discard the edges entering the starting, bound nodes and constant nodes.

Step 2: For each free node, select an incoming edge with the smallest weight. Let the selected $n - 1$ edges be the set $S$.

Step 3: If there are no cycles formed by the edges of $S$, then the selected edges constitute a minimum spanning tree of the graph and the algorithm terminates. Otherwise the algorithm continues.

Step 4: For each cycle formed, contract the nodes in the cycle into a pseudo-node $k$, and modify the weight of each edge entering node $j$ in the cycle from some node $i$ outside the cycle according to the following equation.

$$w(i, k) = w(i, j) - [w(x(j), j) - \min_l \{w(x(l), l)\}]$$

where $w(x(j), j)$ is the weight of the edge in the cycle which enters $j$.

Step 5: For each pseudo-node, select the entering edge, which has the smallest modified weight. Replace the edge, which enters the same real node in $S$ by the new selected edge.

Step 6: Go to step 3 with the contracted graph.

Finding a Low Cost Search Plan

In case of finding a low cost search plan in a given search tree, a simple greedy algorithm is used, which is introduced in Algorithm 5.2. It is important to mention that the value of the edges are the values before the Chu-Liu / Edmonds algorithm was used.

It is important to note that the algorithm grows the spanning tree starting from either a bound or constant node thus we can be sure that in all cases at least one node will be bound for each operation when it is selected as an earlier operation has already provided a mapping candidate to its representing pattern variable. This hinders the need for $FF$ adorned simple operations.

5.4 Realization of an Adorned Search Graph Driven Pattern Matcher in ViATRA2

The local pattern matcher engine of the ViATRA2 transformation framework implements the approach of the search plan driven graph pattern matching based on adorned search graphs as discussed
Algorithm 5.2 A greedy algorithm for generating a low cost search plan

Given an adorned search graph with a selected spanning tree its input.

**Step 0:** Let $B$ be the set consisting of the bound nodes.

**Step 1:** Put all input bound node and constant to $B$.

**Step 2:** Select the smallest tree operation (edge) $e$ that leads out from $B$.

**Step 3:** Put all operations that goes between the nodes in $B$ to list $SP$ that has smaller weight than $e$ (smallest first).

**Step 4:** Add the operation $e$ to $SP$.

**Step 5:** Add the target node of $e$ to $B$.

**Step 6:** If the search forest still has a node that is not in $B$, then go back to Step 2.

**Step 7:** Put all remaining operations to $SP$ in the order of their weight.

In Section 5.3.

In the current section, we discuss implementation specific consideration how we realized the local search based pattern matching engine of the ViTRE2 framework. The proposed workflow of implementing the search plan driven pattern matching engine of ViTRE2 is summarized in Fig. 5.3.

![Figure 5.3: Overview of the ViTRE2 local search based pattern matching approach](image)

We separate compile time parts from run-time parts, where each part consists of the following steps:

- **At compile time** each step is calculated once for each pattern description.
  
  - First, for each pattern description a *call tree* is generated capturing how patterns call other patterns (see in Section 3.1.1. A call tree is a directed bipartite tree describing the structural dependencies of a given pattern by encapsulating the alternative pattern bodies and pattern invocations.
  
  - Then for each call tree a corresponding *search graph* is generated. In order to yield better search plans, the operation scope of the optimizer module is increased by *flattening* the call tree and by merging pattern bodies and pattern invocations into a common search graph. This allows the use of our optimization techniques on a global scale rather than on isolated pattern bodies.

- **After initializing the previous data structures at compile time**, run-time steps have to be calculated for each separate pattern invocation.
– Search plan is generated from the search graph based on the parameter binding to drive the pattern matching process.
– Finally, after executing the search plan matches relevant to the input parameter bindings are passed out.

5.5 Compile Time Steps

In this section we briefly introduce the data structures and algorithms needed for the compile time tasks of the local search based pattern matcher.

5.5.1 Call Tree and Flattening

A call tree is a directed bipartite tree describing the structural dependencies of a given pattern. It is constructed by a traversal process, which explores the possible body alternatives of a pattern and all the pattern invocations in a depth first manner.

Nodes on the odd levels of the call tree represent pattern heads (denoted as simple rectangles), while nodes on the even levels denote pattern bodies (symbolized with numbered circles). The fact that a body is a disjunctive alternative of a pattern head is expressed by an edge connecting the corresponding pattern head to the body. Edges connecting bodies to pattern heads represent non-recursive invocations.

Flattening of Call Trees As previous optimization techniques [Zün96, VVF05] have been developed for simple patterns, they operate on the scope of pattern bodies, which means that a separate optimization procedure is executed for the set of constraints defined by a given body. This approach often results in poor search plans when extensive reuse of pattern bodies are defined, due to the lack of global view for the optimizer on the overall set of structural constraints.

In order to get better search plans, the operation scope of the pattern matcher optimizer module is increased by flattening the call tree and by merging pattern bodies.

The flattening process produces all possible traversals of a call tree from the root to the leaves. These traversals are calculated similar to the and-or tree [Nil80] goals, as pattern body and pattern heads represent conjunction and disjunction nodes, respectively. The algorithm starts from the root node and is recursively invoked on the extended nodes. For a conjunctive node the actual traversal is extended by each child nodes, while for a disjunction node the actual traversal is extended by the child nodes one-by-one representing separate traversals of the call tree.

For a conjunction node all subtree traversals are added to the actual traversal, while in case of a disjunction node, each subtree represents separate traversals. Each traversal is mapped to a flattened pattern, a vector containing the visited pattern body and pattern call nodes. As a result, a flattened call tree is obtained in which the new flattened pattern bodies are direct children of the root pattern head node.

Example 15 The call tree of the missingCircleLink graph pattern (see its ViATRA2 in Listing 5.1 and its graphical representation in Figure 5.4(b)) is illustrated in Figure 5.4(a). The pattern is used in the area management during the boundary breached phase when the world is enlarged. It is responsible to query those HorizontalSideFields and VerticalSideFields that are generated when the boundary is breached and not yet have a circlePath edge between them. It is used by a GT rule that generates the missing links for the newly generated fields.
The **missingCircleLink** pattern (head) has two pattern bodies depicted by circles with numbers 1 and 2. Each pattern body invokes the **circled** pattern head, which only one pattern body and thus for the call tree the invoked pattern bodies through their head are numbered as 3 and 5.

Its flattened version is depicted in Figure 5.4(c). The **missingCircleLink** graph pattern has two flattened pattern bodies (#13 and #24) denoted by vectors, containing the numbers of the constituting body nodes.

For example, the flattened pattern #24 is constructed by starting from the root (disjunctive) node selecting the pattern body 2. From 2 the **circled** pattern head is traversed and the pattern body 4 is selected.

```java
pattern circled(Field1, Field2) = { 
  field(Field1);
  field(Field2);
  field.circlePath(CP, Field1, Field2);
}

pattern missingCircleLink(Field1, Field2) = { 
  find circled(InnerField1, InnerField2); // pattern call
  verticalSideField(Field1);
  verticalSideField.verticalReturnPath(VRP1, Field1, InnerField1);
  verticalSideField(Field2);
  verticalSideField.verticalReturnPath(VRP2, Field2, InnerField2);
  neg find circled(Field1, Field2); // neg finds are not part of the call tree
} or { // multiple pattern bodies
  find circled(InnerField1, InnerField2); // pattern call
  horizontalSideField(Field1);
  horizontalSideField.horizontalReturnPath(HRP1, Field1, InnerField1);
  horizontalSideField(Field2);
  horizontalSideField.horizontalReturnPath(HRP2, Field2, InnerField2);
  neg find circled(Field1, Field2); // neg finds are not part of the call tree
}
```

Listing 5.1: VIATRA2 source code for the **missingCircleLink** graph pattern
### 5.5.2 Search Graph

For each flattened pattern body a separate search graph is generated, where a search graph is built by merging the constraint of the contained pattern bodies of a flattened pattern body, i.e., all formal parameters of the invoked pattern head are substituted with the corresponding actual parameters of the caller. From this merged representation the generation of its corresponding search graph is executed as discussed in Section 5.3.

**Example 16** The search graph of the flattened pattern #13 of the missingCircleLink graph pattern is depicted in Figure 5.5(a). It contains the constraints of both pattern bodies. It follows the structure of the pattern body #1 as the pattern body #3 defined by the circled graph pattern only defines the circlePath edge constraints between the InnerField1 and InnerField2 pattern variables.

### 5.6 Runtime Steps

After calculating and initializing the previous data structures at compile time, the rest of the pattern matching process is carried out at run-time.

### 5.6.1 Search Plan Generation

Generating of a search plan for a specific flattened pattern body requires two input parameters; (i) its corresponding search graph itself and (ii) the binding of its input parameters (adornment on its

---

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{inst}^{FB}(CP, CiPa))</td>
<td>extend</td>
<td>CP is instance of CirclePath</td>
</tr>
<tr>
<td>(\text{trg}^{BF}(CP, IF_2))</td>
<td>extend</td>
<td>target of CP is IF2</td>
</tr>
<tr>
<td>(\text{inst}^{BB}(IF_2, F))</td>
<td>check</td>
<td>IF2 is instance of Field</td>
</tr>
<tr>
<td>(\text{src}^{BF}(CP, IF_1))</td>
<td>extend</td>
<td>source of CP is IF1</td>
</tr>
<tr>
<td>(\text{inst}^{BB}(IF_1, F))</td>
<td>check</td>
<td>IF1 is instance of Field</td>
</tr>
<tr>
<td>(\text{trg}^{BB}(VRP_1, IF_1))</td>
<td>extend</td>
<td>target of VRP1 is IF1</td>
</tr>
<tr>
<td>(\text{inst}^{BB}(VRP_1, VeRePa))</td>
<td>check</td>
<td>VRP1 is instance of VeRePa</td>
</tr>
<tr>
<td>(\text{trg}^{BB}(VRP_2, IF_2))</td>
<td>extend</td>
<td>target of VRP2 is IF2</td>
</tr>
<tr>
<td>(\text{inst}^{BB}(VRP_2, VeRePa))</td>
<td>check</td>
<td>VRP2 is instance of VeRePa</td>
</tr>
<tr>
<td>(\text{inj}^{BBBB}(CP, VRP_1, VRP_2))</td>
<td>check</td>
<td>CP != VRP1 != VRP2</td>
</tr>
<tr>
<td>(\text{src}^{BF}(VRP_1, F_1))</td>
<td>extend</td>
<td>source of VRP1 is Field1</td>
</tr>
<tr>
<td>(\text{inst}^{BB}(F_1, VeSiFi))</td>
<td>check</td>
<td>F1 is instance of VeSiFi</td>
</tr>
<tr>
<td>(\text{src}^{BF}(VRP_2, F_2))</td>
<td>extend</td>
<td>source of VRP2 is Field2</td>
</tr>
<tr>
<td>(\text{inst}^{BB}(F_2, VeSiFi))</td>
<td>check</td>
<td>F2 is instance of VeSiFi</td>
</tr>
<tr>
<td>(\text{nac}^{BB}_1(F_1, F_2))</td>
<td>check</td>
<td>(nac_1) check</td>
</tr>
<tr>
<td>(\text{inj}^{BBBB}_1(F_1, IF_1, F_2, IF_2))</td>
<td>check</td>
<td>F1 != IF1 != F2 != IF2</td>
</tr>
</tbody>
</table>

---

Figure 5.5: Search graph and search plan for the compound pattern body #13 of the missingCircleLink graph pattern.
pattern variables). The generation is realized as described in Section 5.3. We implemented a simple caching mechanism to prevent the generation of search plans that are already available with a specific adornment. Additionally, there is an option for the transformation designer to generate search plans for all graph patterns with all kinds of binding. However, it turned out that it resulted in a large overhead at compile time with a very moderate speed up at runtime and this advantage was only measurable in case of small instance models.

Example 17 One possible search plan for the flattened pattern body of the missingCircleLink graph pattern with all input parameters considered as unbound is depicted in Section 5.5(b). The interesting parts are that it starts from the CP relation using its CirclePath constant node and from there extends the matching in both direction towards Field1 and Field2. Additionally, due to the iteration over all CirclePath typed link in the instance model as the starting operation the algorithm came up with a search plan that tries to use the injectivity check between the edges as soon as possible in order to keep the search tree small.

5.6.2 Executing the Operations

Finally, the generated search plan is executed using a simple algorithm that tries to execute the operations of the search plan and backtracks if one fails (check) or could not provide additional matching candidate for the specific pattern variable (extend). In this way the execution of the search plan produces a match if the algorithm executes the last operation.

The skeleton of the actual Java implementation of the algorithm in the Viatra2 framework is depicted in Listing 5.2. The input frame object represents a container that stores all pattern variables and their matching candidate values. All operation is a subtype of the SearchPlanOperation class and the operations array contains the pattern matching operations of the search plan. The code first simply initializes the frame that sets the constant and bound pattern variables as required by the search plan and then starts executing the operations in the following while-do construct. The while terminates if either the last operation is executed and thus the frame object contains a valid match or the loop backtracked to the first operation. Within the loop the actual operation which is saved by the currentOperation variables is executed. If its execution returns with a true the execution continues with the following operation in the next round if not, than the operation is backtracked using the postprocess method of the operation. In some cases preprocessing is also required that is handled by calling the preprocess method.

It is important to note that the extend operations hold their actual state and contain information on the already tried candidates. For example, in case of an operation that iterates through the elements of a specific type (see in Example 17 the first operation of the search plan that extends through the CirclePath type as candidates for the CP pattern variable) it saves which CirclePath links were already tried as a candidate during its executions.

Finally, the match is filtered out of those elements of that do not correspond to a parameter of the pattern signature.

```java
public class SearchPlan {
    private SearchPlanOperation[] operations;
    private int currentOperation;
    ...
    /** Executes the Search Plan
     * @param frame The container for the match. It holds the values of the
     *    pattern variables
     * @return true if there is a match found and false if there is no match
     *          @throws PatternMatcherRuntimeException */
```
CHAPTER 5. SEARCH PLAN DRIVEN GRAPH PATTERN MATCHING

```java
public boolean execute(MatchingFrame frame) throws PatternMatcherRuntimeException {
    int upperBound = operations.length - 1; //init the upper bound
    init(frame); //init the constant or bound values in the frame
    //the execution stops if all operations are executed or backtraced to the beginning
    while (currentOperation >= 0 && currentOperation < upperBound) {
        if (operations[currentOperation].execute(frame)) { //executes the operation
            currentOperation++;
            operations[currentOperation].preprocess(frame); //complex operations require preprocessing
        } else {
            operations[currentOperation].postprocess(frame); //backtracks the operation
            currentOperation--;
        }
    }
    return (currentOperation == upperBound); //true if it reached the last operation
}
```

Listing 5.2: Skeleton of the operation execution code

5.6.3 Performance

The performance of our implementation of the adorned search graph driven pattern matcher in the ViATRA2 framework was assessed in many graph and model transformation tool contests either as a separate pattern matcher [27,17] or as part of the ViATRA2 hybrid pattern matcher engine [16,24,21]. Its evaluation as a standalone pattern matcher on the Sierpinski case study introduced at the 2007 AGTIVE tool contest is discussed in Section 5.6.3.1, while as part of the hybrid engine on the AntWorld running example is discussed in Section 6.3.3.

5.6.3.1 Summary of the Sierpinski Graph Transformation Tool Contest Case Study

The task in the case study is to simulate the creation of a Sierpinski graph. Originally constructed as a mathematical curve, this is one of the basic examples of self-similar sets, i.e. it is a mathematically generated pattern that can be reproduced at any magnification or reduction. It is created by the following algorithm:

1. Start with any triangle in a plane (any closed, bounded region in the plane will actually work). The canonical Sierpinski triangle uses an equilateral triangle with a base parallel to the horizontal axis.

2. Shrink the triangle by $1/2$, make two copies, and position the three shrunken triangles so that each triangle touches the two other triangles at a corner.

3. Repeat step 2 with each of the smaller triangles.

**The Solution** Our solution follows the simple structure that it first matches all triangles that are needed to be extended, then generates the required additional triangles. Figure 5.6 illustrates our simple metamodel.

Based on this metamodel, we used a very simple pattern (see in Listing 5.3) to match to the required triangles. It simply defines the triangle in a clockwise order that, due the definition of our metamodel, matches only on those triangles that are needed to be extended in the following round.
Listing 5.3: Pattern for finding the required triangles

```java
pattern triangle(A, B, C, EAB, EBC, ECA) = {
    a(A); node.e(EAB, A, B);
    b(B); node.e(EBC, B, C);
    c(C); node.e(ECA, C, A);
}
```

Listing 5.4: The triangle generator transformation

```java
rule generate(in MaxRounds) =
    let RoundCount = 0 in seq {
        iterate if(RoundCount < MaxRounds) seq {
           forall A, B, C, EAB, EBC, ECA with find triangle(A, B, C, EAB, EBC, ECA) do seq {
                // The old edges are deleted:
                delete(EAB);
                delete(EBC);
                delete(ECA);
                // The new nodes are created:
                new(ab(AB) in models("model"));
                new(bc(BC) in models("model"));
                new(ac(AC) in models("model"));
                // The new edges are created:
                new(node.e(EAAB, A, AB)); new(node.e(EABB, AB, B)); new(node.e(EACBC, AC, BC));
                new(node.e(EABAC, AB, AC)); new(node.e(EBBC, B, BC)); new(node.e(EBOC, BC, C));
                new(node.e(EACCA, AC, A)); new(node.e(ECAB, BC, AB)); new(node.e(ECAC, C, AC));
            }
            update RoundCount = RoundCount + 1;
        }
        else fail;
    }
```

Control Flow  Listing 5.4 describes the main control flow of our Sierpinski generator transformation program. It uses the `triangle` pattern in the `forall` to match to all required triangles and then in all iterations it first deletes the old edges, then creates the new triangle and connects it to the rest of the model.

The outer iteration makes sure that the Sierpinski generation is not executed more than its input maximum iteration number.

Conclusion  On an Intel T2400@1830 MHz notebook with 2 GByte memory, we managed to handle 10 iterations and received a JVM out of memory exception on the eleventh. It is roughly a little less than one million model elements (nodes and edges altogether). As for the runtime performance, due to some randomization in the matching process, we measured huge differences between different executions, but overall the generation took from a couple of minutes to more then 10 minutes for the 10. iteration. Detailed results for the different iterations are depicted in Figure 5.7.
In overall, based on this case study and all of our other experiences \cite{27,17,2} the local search based pattern matcher is capable of handling problems within the few hundred thousand model elements with acceptable performance.

5.7 Summary

In order to address the performance aspects of graph pattern matching, we proposed a general framework for uniformly representing a large variety of search plan operations by expressing them as cost-weighted predicates. As an appropriate ordering of these predicates defines an executable search plan, this approach allowed to uniformly guide the pattern matching process for advanced graph patterns regardless of how the actual costs to different search plan operations are assigned.

As a result, the different phases of pattern matching (e.g. cost assignment, generation of search plans, execution of search plans etc.) are fully separated and independent, thus they can be adapted to different graph transformation engines and strategies (e.g., metamodel-based vs. model sensitive search plans \cite{VVF05,VDWS12}). Furthermore, new types of predicates can be introduced easily by assigning appropriate costs without altering the algorithms for search plan generation.

However it is important to note that our latest implementation of the introduced techniques does not support – due to a restriction in the model management framework of ViATRA2, and not in the approach itself – model sensitive search plan generation. The only differences in the case of runtime weighting are (i) the usage of runtime statistics collected from the instance model, (ii) and the more precise weighting of simple extend type predicates (e.g., the weight of an $instanceOf$ constraint is based on actual number of the instance elements in the model), and $nac$ check operations, where the cost can be directly derived from the pattern graph of the NAC.

Limitation of the Adorned Search Graph Driven Approach: It should be emphasized that this chapter presented practical heuristics for pattern matching. Neither the technique of adorned search graph driven pattern matching, nor the approach of metamodel based weight selection combined with the two greedy-algorithms can be provenly optimal by their nature. The problem lies in the \textit{Sum of Products} cost function that has been used for our search plan generation. The customized
greedy algorithms (see in Section 5.3.2) can only provide low cost, but not necessarily optimal search plans. From a mathematical point of view, it is easy to find counterexamples for the optimality of the presented algorithms (e.g., complete graph with only one type). Additionally, the weights for the different constraint do not reflect precisely the cost of the operations it would require as it is based on the metamodel and not on the actual instance model.

However, we believe that these problems are rarely occur in practical application domains as the implemented approach did quite well in many graph transformation tool contests [27,21].
Hybrid Graph Pattern Matching

Practical experience has shown that optimizing model transformations is an important part of applying model-driven techniques for system development. First, as models are increasing in size and complexity, transformations need to be able to transform them efficiently. Secondly, as transformations are becoming hidden (e.g. embedded in a design tool), they should execute seamlessly - quickly and using as little resources as possible.

As a result of these experiences we found that the incremental pattern matching approaches may lead to orders-of-magnitude increases in speed. However, an important implication of caching match sets is increased memory consumption, which needs to be taken into account when scaling up to large models. Unfortunately, in many practical applications of model transformations, available memory is frequently constrained (e.g. when they are executed on average desktop computers and not on high performance servers).

We believe that many transformations could benefit even more from combining these two approaches to use the most suitable pattern matcher engine for each graph pattern. We propose a hybrid pattern matching approach which enables the transformation designer to combine local search-based and incremental pattern matching to adapt to memory constraints. At design-time, transformation engineers may select whether a graph pattern should be matched using the LS or the INC strategy separately for each pattern. Moreover, based upon runtime monitoring, the execution engine may automatically switch from incremental pattern matching to local-search based technique when a certain memory limit has been reached.

The structure of the current chapter is the following. Section 6.1 briefly introduces incremental pattern matching. Section 6.2 outlines how the hybrid engine based upon the adorned search graph representation. Then Section 6.3 describes the runtime results of the AntWorld case study on the different pattern matching strategies. Finally, Section 6.4 defines metrics based on our earlier experiences how to select between the different strategies and additionally, we present an adaptive runtime technique to switch to LS strategy in case of low memory.
6.1 Incremental Graph Pattern Matching

Incremental pattern matching [VVS06b, BOR+08] offers a different execution model compared to local search-based implementations. The match sets for all patterns involved in the graph pattern are computed in an initialization phase prior to execution (e.g., when the model itself is loaded into memory), and as the transformation progresses, this match set cache is incrementally updated as the model graph changes (update phases). Thus, model search phases are reduced to fast read-from-cache operations, in exchange for the overhead imposed by cache update phases which occur synchronously with model manipulation operations. There are several incremental pattern matching approaches [GJR10, LS05, VVS06b, BOR+08, MMS07] available for graph pattern matching, in the current section, we give a short overview on our approach based on RETE nets [For82] as realized in the VIATRA2 framework.

Our current introduction is based on [BOR+08].

6.1.1 RETE-based Incremental Graph Pattern Matching

RETE-based pattern matching relies on a network of nodes storing partial matches of a graph pattern. A partial match enumerates those tuples of model elements which satisfy a subset of the constraints described by the graph pattern. In a relational database analogy, each node stores a view. Matches of a pattern are readily available at any time, and they will be incrementally updated whenever model changes occur.

Information is represented by a tuple consisting of model instance elements. Each node in the RETE net is associated with a (partial) pattern and stores the set of tuples that conform to the pattern. This set of tuples is in strong analogy with the relation concept of relation algebra.

Within the RETE net we define the following type of nodes:

- **Input nodes** that serve as the underlying knowledge base representing the instance model. There is a separate input node for each entity type (class), containing a view representing all the instances that conform to the type. Similarly, there is an input node for each relation type, containing a view consisting of tuples with source and target in addition to the identifier of the edge instance.

  Additionally, when a pattern calls another pattern, it can simply use the appropriate production node of the called pattern to obtain the set of tuples conforming to the other pattern. This special input node is called pattern input node

- **Intermediate nodes** store partial matches of patterns, or in other terms, matches of partial graph patterns. In our approach we use the following type:

  - The most widely used node type is the join node. It is created as the child of two parent nodes, that each have an outgoing RETE edge leading to the newly created join node. Its role can be best explained with the relational algebra analogy: it performs a natural join on the relations represented by its parent nodes.

  - **Negative node** (or antijoin node) has two distinctive parents: primary and secondary inputs, respectively. The negative node contains the set of tuples that are also contained by the primary input, but do not match any tuple from the secondary input, which is analogous to the antijoins in relational algebra.

  - **Term evaluation node** simply represents the attribute condition defined in a graph pattern. It propagates only those tuples that pass the given attribute condition.
Finally, production nodes represent the complete pattern itself. Production nodes also perform supplementary tasks such as filtering those elements of the tuples that do not correspond to symbolic parameters of the pattern (in analogy with the projection operation of relational algebra) in order to provide a more efficient storage of models. Additionally, in case of multiple pattern bodies the production node is responsible for filtering out duplicate matches.

It is important to mention that different pattern bodies of a graph pattern are matched using separate matchers (RETE nets) for each body, while sharing the production node, which will perform a true union operation on the sets of the tuples conforming to each pattern body.

**Example 18** As an illustration, Figure 6.1 shows a simplified RETE network matcher built for the `missingCircleLink` (see Figure 5.4(b)) graph pattern.

All nodes share a similar graphical representation, where their lower part represent the pattern variables of their tuple and their name. For example, the `CirclePath` node has the `F_SRC`, `CP` and the `F_TRG` pattern variables in its tuple and is an input node which produces all `CirclePath` typed link from the instance model, where, based on the metamodel we know that its source target object will be an instance of `Field`. Additionally, for easier readability each node highlights the part of the original graph pattern that is taken into account by the RETE net up to the output of the node. It means that the output tuples of the node fulfil the highlighted part of the original graph pattern. For example, the second join would produce all matches of the complete pattern without the negative application condition check defined for the `circlePath` edge. Finally, in case of join and anti-join nodes the pattern variables used for the join (or anti-join) are illustrated in the top blue box. For example, in case of the anti join the `F1` and `F2` pattern variables are anti joined with the `F_SRC` and `F_TRG` pattern variables, respectively.

The RETE net can be separated into three parts. The first upper part consists of the three input nodes. The `VerticalReturnPath` and the `CirclePath` represent edge inputs – of corresponding types – while the `circled` node is a pattern input node for the circled graph pattern matched by an other RETE net. The center part defines the two join and one anti-join nodes that filter out the matching candidates for the pattern by applying the connectivity and negative application condition constraints on the pattern variables. First it matches the complete pattern and then uses the anti join to filter out those tuples that satisfies the `circled nac` pattern. Finally, the lower part consisting of the production node that is used to filter out the output pattern variables as defined in the head of the graph pattern in our case the `Field1` and `Field2` pattern variables and execute the required injectivity checks on the complete matching.

Note that the first pattern body of the `missingCircleLink` graph pattern is also depicted (on the right side) in the figure to ease the understanding of the RETE net.

### 6.1.2 Updates After Model Changes.

Model changes are propagated through the network, modifying the match sets stored at the nodes incrementally, since each node only recomputes a partial matching. Thus, the pattern matcher is capable of incrementally tracking the match set of a complex pattern by decomposing the pattern into constraints, constructing a RETE network based on that decomposition, and updating it as models change.

Input nodes receive notifications about each elementary model change (e.g. when a new model element is created or deleted) and release an update token on each of their outgoing edges. Such an update token represents changes in the partial matches stored by the RETE node. Positive update tokens reflect newly added tuples, and negative updates indicate tuples being removed from the set.
Figure 6.1: Sample RETE net of the first pattern body of the missingCircleLink graph pattern
Upon receiving an update token, a RETE node determines how the set of stored tuples will change, and release update tokens of its own to signal these changes to its child nodes. This way, the effects of an update will propagate through the network, eventually influencing the result sets stored in production nodes.

The match set can be retrieved from the network instantly without re-computation, which makes pattern matching very efficient. As a trade-off, there is increased memory consumption, and update operations become more complex.

6.1.3 Construction of the RETE net

The key is perceiving a pattern as a collection of constraints (similarly to the concept of search graphs as defined in Section 5.1) imposed on subsets of the group of pattern variables. The construction algorithm processes these constraints one by one, and continues a connected sequence of nodes to match larger and larger partial patterns, eventually using up all constraints and connecting the last node to the production node.

Without going into details the RETE builder in the VIATRA2 framework is built upon the following guidelines: for simple entity, relation (1) the appropriate input node or intermediate is accessed; (2) a join node will be attached as a child to it and also to the end of the line; (3) the join node will be prepared to match against variables that are involved in the constraint and are already introduced in the line. For negative application conditions, a negative node is used instead of the join node in an otherwise similar setup. A different setup is required for check conditions (and some where a single filtering node (in this case, a term evaluator node) is attached at the end of the line.

6.2 Combining Local Search based and Incremental Graph Pattern Matching

In the VIATRA2 framework, a transformation designer can fine-tune the performance or memory consumption of graph pattern matching – within a complex transformation (program) – by prefixing the different graph patterns with @localsearch or @incremental annotations to select the designated pattern matching strategy for each graph pattern separately. This way the interpreter automatically uses the defined pattern matcher during the transformation execution.

This feature also holds for composite pattern (bodies) which allows the definition of different matching strategies for certain parts of the pattern. This way the search plan generated for these composite patterns are optimized to favor (already) incrementally matched patterns traversal in the early steps of the matching process to bind elements for the later LS matched part. The same algorithm as for LS is used to generate these search plans. It differs only in two parts: (i) the flattening process is not invoked on the incrementally matched patterns and (ii) during execution the incrementally matched pattern invocations are transformed into one search operation that bound its interface pattern variables from its cache. The high-level workflow of this technique is illustrated in Fig. 6.2.

Example 19 To illustrate how hybrid pattern matching is performed (in VIATRA2), Figure 6.3 shows the search plan and adorned search graph of the first pattern body of the missingCircleLink graph pattern. The pattern is invoked using the LS matcher, however, the circled graph pattern is already matched by the INC matcher thus the LS matcher uses the already cached matches of the circled pattern when creating the search plan for the missingCircleLink. The search graph created from this combination is depicted in Figure 6.3(a).
CHAPTER 6. HYBRID GRAPH PATTERN MATCHING

Figure 6.2: Selecting pattern matching strategies

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$inv_{c}^{BF}(IF1, IF2)$</td>
<td>extend</td>
<td>circled invocation (INC)</td>
</tr>
<tr>
<td>$trg_{FB}^{BF}(VRP1, IF1)$</td>
<td>extend</td>
<td>target of VRP1 is IF1</td>
</tr>
<tr>
<td>$inst_{BB}^{BF}(VRP1, VeRePa)$</td>
<td>check</td>
<td>VRP1 is instance of VeRePa</td>
</tr>
<tr>
<td>$trg_{FB}^{BF}(VRP2, IF2)$</td>
<td>extend</td>
<td>target of VRP2 is IF2</td>
</tr>
<tr>
<td>$inst_{BB}^{BF}(VRP2, VeRePa)$</td>
<td>check</td>
<td>VRP2 is instance of VeRePa</td>
</tr>
<tr>
<td>$inj_{BB}^{BB}(VRP1, VRP2)$</td>
<td>check</td>
<td>VRP1 != VRP2</td>
</tr>
<tr>
<td>$src_{BP}(VRP1, F1)$</td>
<td>extend</td>
<td>source of VRP1 is Field1</td>
</tr>
<tr>
<td>$inst_{BB}(F1, VeSiFi)$</td>
<td>check</td>
<td>F1 is instance of VeSiFi</td>
</tr>
<tr>
<td>$src_{BP}(VRP2, F2)$</td>
<td>extend</td>
<td>source of VRP2 is Field2</td>
</tr>
<tr>
<td>$inst_{BB}(F2, VeSiFi)$</td>
<td>check</td>
<td>F2 is instance of VeSiFi</td>
</tr>
<tr>
<td>$nac_{BB}(F1, F2)$</td>
<td>check</td>
<td>nac1 check</td>
</tr>
<tr>
<td>$inj_{BB}(F1, IF1, F2, IF2)$</td>
<td>check</td>
<td>F1 ! = IF1 ! = F2 ! = IF2</td>
</tr>
</tbody>
</table>

(b) Search Plan

Figure 6.3: Search graph and search plan for the first pattern body *missingCircleLink* graph pattern using hybrid pattern matching

The novel part of the search graph (compared to the search graph illustrated in Figure 5.5(a)) is the new complex predicate $inv$. It represents that the *InnerField1* and *InnerField2* pattern variables are needed to be a valid match of the circled graph pattern (as they are already matched by the incremental matcher). Due to the direct invocation of the circled graph pattern its constrains are not part of the current search graph. Besides this difference that search graph is similar to the one created for the local-search based matcher.

Unsurprisingly, the search plan (see in Figure 6.3(b)) generated from the search graph with no
bound input parameters, executes the \( \text{inv}_{\text{circled}}^{FF} \) operation as its first operation as it only requires a simple read from the cached match set of the circled graph pattern’s production node. The cost of the \( \text{inv} \) operation is usually very low as it only requires a simple read operation from a cached set, thus considered as one of the lowest cost extend operation. From that point on the search graph is similar to the discussed in Section 5.5.2 as it is matched with the local search based matcher.

It is important to mention that the current hybrid combination for matching the missingCircleLink is used only for demonstration purpose (ease the comparison of the different pattern matching approaches) and in our submitted AntWorld case study we did not apply this combination. How we defined our matching strategy is shown in Appendix A.

6.2.1 Extensions to the Adorned Search Graph

In order to be able to handle pattern invocations in our search graph concept defined in Section 5.2 we have to define the \( \text{inv} \) complex predicate and its corresponding operation.

For this, we need a slight modification for the definition of a pattern body

**Definition 17** Given a metamodel \( MM \), a **pattern body** \( PB = (SC, AC, \forall j \in I NAC_j, \forall i \in IP_i) \) is a tuple consisting of the same definitions as described in Def. 4 with the following additional item:

- **invoked pattern** \( IP_i = (GP_{PH_i}^{invoked}, sign_{GP_{PH_i}^{invoked}}) \), defined by a graph pattern \( GP_{PH_i}^{invoked} \) that prescribes an additional contextual conditions for the original pattern which are needed to be fulfilled in order to find a successful match of the original pattern body. The nodes and edges shared between the invoked pattern and the container pattern body \( PB \) are defined by the injective partial morphism \( sign_{GP_{PH_i}^{invoked}} : SC_{PB} \to GP_{PH_i}^{invoked} \).

This modification has a small implication on the definition of a match of a graph pattern (see in Def. 7 adding the following item to its definition

- \( \forall i \in I \) where \( m' : GP_{PH_i}^{invoked} \to M \) there is a match for all of its invoked patterns that extends the match of the pattern body \( PB_i \).

Using this definition the predicate is defined as follows:

**Definition 18** Each invoked pattern \( IP_i \) of the \( PB \) pattern body is mapped to a **pattern invocation (hyper)edge** \( \text{inv}_i(pvar_1, \ldots, pvar_n) \) in the search graph, where each corresponding \( pvar_u \), pattern variable of node or edge \( u_i \) of the structural constraint that is shared with the invoked pattern (as defined by \( sign_{GP_{PH_i}^{invoked}} \)) is part of \( \text{inv}_i \). Formally, \( \forall IP_i, \exists \text{inv}_i \in E_{SG}^{INV} : b(\text{inv}_i) = PH_i \) and \( \forall j = 1 \ldots n, u_j (V_{PBSC} \cup E_{PBSC}) \), \( sign_{GP_{PH_i}^{invoked}}(u_j) : \exists b(pvar_u) = u_j \wedge u_j \in \text{inv}_i \).

Additionally, the \( \text{inv} \) hyper edges represent a separate partition within the edges of the search graph labeled as \( E_{SG}^{INV} \).

**Definition 19** \( \text{inv}Xx_{\text{patternname}}(pvarx_1, pvarx_2, \ldots, pvarx_n) \), where \( X = 'B' \) or ‘F’ prescribes the execution of a full-featured **incremental pattern matching** for the invoked pattern with the name \( \text{patternname} \) with the \( m' \) initial matching, where all shared pattern elements from \( m \) are free or bound in \( m' \) as its operation adornment prescribes, formally, \( \forall m'(x_i), i = 1 \ldots n, \exists m(y) \neq \text{null} : m'(x_i) = m(y) \). Note that in any case where there is one pattern variable that is free then the operation is considered as an extend type operation and in case of an all bound adornment it is a check.
It is important to mention that in all cases due to the incrementally matched patterns that are
behind these invocation operations they represent only a simple operation. This operation is a read
from their corresponding match set cache and in case of bound input pattern variables an additional
simple check that the bound parameters are part of the match.

Finally, the algorithms defined (see in Section 5.3.2) for the generation of the search plan do not
need any modification to be able to handle the invocation operation. The only exception is that it
is allowed to start the search plan generation from any incrementally matched pattern invocations
as in certain cases there may not be any constant or bound pattern variables. This way its pattern
variables become bound and the algorithm can continue as described. As for the cost on the inv
operation, we wanted to be very cheap and therefore it has similar cost as a simple navigation along
an edge.

6.3 Benchmark Evaluation on the AntWorld Case Study

In this section, we present our experiments to assess our hybrid engine’s performance on the
AntWorld case study. Our main goal with benchmarking is two-fold: (i) to demonstrate how the
performance of ViATRA2 evolved with the hybrid pattern matching approach (Section 6.3.3), and (ii)
to present useful design-time optimizations and fine-tuning options in Section 6.3.1 which can have
significant impact on performance.

6.3.1 Fine-tuning of Pattern Matching Strategies

We designed our implementation to effectively support a hybrid pattern matching approach (see
in Section 6.2) that trades runtime performance for memory consumption compared to the pure
incremental solution. This hybrid solution was based on the following considerations:

- Considerable memory can be saved by ensuring that the map (fields and path relations) is
  not contained in the RETE net, as these are the types with the highest number of instances.
  Patterns concerning these model features should be assigned to the local search based matcher,
  to keep the RETE net small. As these patterns happen to establish simple local relationships of
  low complexity, they are efficiently matched using the local search based engine.

- To achieve high performance through avoiding expensive repeated searching, the incremental
  pattern matcher was selected to deal with the hasFood, location, hasPheromone, bound-
  ary relations. This allowed useful collections such as ants stumbling upon food, ants reaching
  the boundary, or pheromones that are still strong enough to attract ants to be incrementally
  maintained.

- Some patterns contain model features of both kinds. On certain occasions, a subpattern
  was extracted for the incremental pattern matcher, and the local search based matcher uti-
  lized this cache in a true hybrid fashion. For example see this in the precondition of the
  moveTowardsAttractingPheromone GT rule (see in Figure 3.1(d)).

6.3.2 Measurements

We conducted benchmark measurements on our test system with a quad-core Intel Xeon CPU clocked
at 2.00 GHz and 12 GBs of system memory. We used the OpenJDK 64-bit Server VM (IcedTea6 1.3.1
build 12) on Linux 2.6.18 with 10GBs of memory allocated to the JVM.
6.3. BENCHMARK EVALUATION ON THE ANTWORLD CASE STUDY

6.3.2.1 Variants

Our experiment group was performed to demonstrate the difference between various pattern matching strategies (Section 6.3.1).

We configured the transformation program (available in Appendix A) with annotations to create the following run configurations:

1. the local search solution made exclusive use of the traditional, local search-based pattern matcher implementation described in Section 5.3.

2. in contrast, the incremental solution relied solely on the RETE-based pattern matcher described in Sec.6.1.

3. finally, we combined the pattern matching strategies with techniques described in Section 6.4.1 and 6.3.1 to create a hybrid solution.

6.3.2.2 Telemetry

To obtain numeric results, we designed the simulation transformation to generate XML output containing execution time and memory usage telemetry data. Every 25 rounds, telemetry data was written to an output buffer, which was flushed to a file after the transformation has terminated.

Overall, we executed five 500-round simulation runs for each variant, with the exception of the local search solution where only 150 rounds were executed (since it is significantly slower than the other two variants). Memory consumption measurements were performed in separate execution runs to avoid a potential negative performance impact.

6.3.3 Analysis of the Results

Results were analyzed by transforming the XML output to CSV spreadsheets which were processed in OpenOffice.Org 3.0. We combined the results from each of the five separate execution runs to create a data series consisting of 100 records for INC and HYB (30 records for LS).

6.3.3.1 Complexity class analysis

By looking at the data, we found that there is a very high correlation (correlation coefficient $R^2 > 0.995$) between the time needed to execute a round and the number of ants (Figure 6.4). There is also a fairly high correlation ($R^2 > 0.97$) between the number of fields in the grid and the measured memory consumption (Figure 6.6). The number of rounds, however, has a significantly weaker correlation ($R^2 < 0.9$ for some solutions) with both the round time and the memory footprint size. Thus, we generated charts which show cumulative and per-round execution times against the number of ants, and heap usage compared to the number of fields.

The cumulative execution time chart with linear scales is shown in Figure 6.5. While the local search variant exhibits a high-order polynomial increase as the ant population is growing, both the pure incremental and hybrid variants perform significantly better, following a low-order polynomial characteristic. How these correlations were computed is discussed in Appendix B.

---

1 Execution time was measured by the `System.currentTimeMillis()` Java call, while heap usage was estimated by performing garbage collection calls (`System.gc()`) and recording the result of `Runtime.totalMemory() - Runtime.freeMemory()`.
6.3.3.2 Effects of Hybrid Pattern Matching

Figure 6.6 shows memory consumption data comparing pure incremental pattern matching with our hybrid approach. In both cases, the overall heap consumption of the ViATRA2 engine grows linearly with the number of grid fields, however, the gradient for the hybrid run is lower (for a given number of fields, the pure incremental variant consumes approximately 1.5 times more memory than the hybrid variant). Since the execution time per iteration values are also linear for the hybrid variant (Figure 6.4), it can be concluded that the hybrid pattern matching approach performs in the same complexity class as the pure incremental version. In other words, for a linear decrease in memory consumption, a linear decrease in execution speed can be expected (as supported by the constant
6.3. BENCHMARK EVALUATION ON THE ANTWORLD CASE STUDY

![Figure 6.6: Memory Delta](image)

### Optimization summary

The summary of the results obtained from the various graph pattern matching strategies is shown in Table 6.1.

To see how far ViAtRa2 can go with the most optimized implementation on our test hardware, we conducted a final test run which ran until the 10GB JVM heap space was exhausted. The results are shown in Table 6.2.

<table>
<thead>
<tr>
<th>PM strategy</th>
<th>Performance</th>
<th>Memory footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS</td>
<td>High-order polynomial</td>
<td>Constant</td>
</tr>
<tr>
<td>Switch to INC</td>
<td>Polynomial order reduction</td>
<td>Linear increase with model size</td>
</tr>
<tr>
<td>Switch to Hybrid</td>
<td>Linear 50% loss</td>
<td>50% reduction</td>
</tr>
</tbody>
</table>

Table 6.1: Optimization strategies

<table>
<thead>
<tr>
<th>Variant</th>
<th>Iterations</th>
<th>Model elements</th>
<th>Total time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>INC</td>
<td>1200</td>
<td>~1.5M</td>
<td>4969</td>
</tr>
<tr>
<td>Hybrid</td>
<td>1400</td>
<td>~2.0M</td>
<td>11907</td>
</tr>
</tbody>
</table>

Table 6.2: Statistics for the maximum possible iteration count

difference in the logarithmic plot in Figure B.2).
6.4 Towards Intelligent Selection of Matching Strategies

In this section, we first identify various factors (qualitative metrics) which help transformation designers decide when a certain pattern matching strategy (LS or INC) would be beneficial (Section 6.4.1). Then, in Section 6.4.2, we discuss how a simple adaptive run-time behaviour can be obtained by monitoring relevant metrics, and switching from one strategy to the other at runtime. Compared to existing adaptive pattern matching solutions [VVF05, JBK10, VDWS12], the main novelty of this approach lies in the fact that we are able to automatically switch between two entirely different pattern matching strategies to increase performance. The high-level workflow of these techniques is illustrated in Figure 6.7.

As identified in Section 6.3 (and Appendix C), several factors may influence the behaviour of the pattern matching algorithms. Static factors like (i) static attributes of graph patterns (e.g. pattern size, fan-out, structural complexity) and (ii) control structures of model transformations (e.g. forall, iterate) determine operative characteristics which, in combination with the characteristics with the different pattern matcher strategies, greatly influence the cost of pattern matching.

<table>
<thead>
<tr>
<th>Strategy selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design-time</td>
</tr>
<tr>
<td>Transformer</td>
</tr>
<tr>
<td>Load pattern</td>
</tr>
<tr>
<td>Estimate static metrics</td>
</tr>
<tr>
<td>Decide strategy to match pattern</td>
</tr>
<tr>
<td>Run-time</td>
</tr>
<tr>
<td>Execution engine</td>
</tr>
<tr>
<td>Monitor dynamic metrics</td>
</tr>
<tr>
<td>Switch PM strategy (INC-&gt;LS)</td>
</tr>
<tr>
<td>Memory low?</td>
</tr>
</tbody>
</table>

Figure 6.7: Selecting pattern matching strategies at design-time and runtime

In contrast, dynamic factors change in-between transformation runs on the same system, and also with different target execution platforms: (iii) model-specific graph characteristics like qualitative attributes related to structure (e.g. average fan-out) and quantitative parameters related to model size (e.g. total number of model elements) may change as the transformation is changing the underlying model. Moreover, (iv) memory limitations impose external constraints which are related to the execution environment.

6.4.1 Factors for design-time selection of matching strategies

Based on our previous experience with performance benchmark transformations [17, 2, 24] and practical model transformations of large complexity [KLMB08] and [6], we identified the following factors to be important for transformation designers to choose between LS and INC strategies:
6.4. TOWARDS INTELLIGENT SELECTION OF MATCHING STRATEGIES

(i) Graph pattern static attributes

- **number of graph patterns** in a transformation program has a huge impact on the memory consumption – especially in resource constrained environments like embedded systems. The cache size of the pattern increases memory consumption when matched by INC strategy.

- **pattern size**, in practical applications, we experienced that the number of matches gradually decrease as the pattern to be matched becomes more and more complex (contradicting the theoretical complexity, which predicts that large patterns will have more matches. As a result, large patterns should be preferably matched by INC.

- Considerable **memory** can be saved by ensuring that the map (fields and path relations) is not contained in the RETE net, as these are the types with the highest number of instances. Patterns concerning these model features should be assigned to the local search based matcher, to keep the RETE net small. As these patterns happen to establish simple local relationships of low complexity, they are efficiently matched using the local search based engine.

(ii) Control structures

- **parameter passing** is using the result of rules or patterns as an input of other rules or patterns. This technique increases efficiency in LS as search operations are much more efficient if one or more pattern variables are bound, i.e. their values are known at time of the query. INC performance is not affected.

- **usage frequency** of patterns is relevant, since the more often a pattern is used, the more advantage INC has. Frequently used patterns can be identified by static analysis of the transformation code, e.g. by marking patterns that are used from within a loop. Trace analysis can yield more valuable estimates, if typical example inputs are available, by executing the transformation on these inputs and counting the times each pattern is accessed.

- **model update cost**: if program code analysis can reveal that model element types belonging to a certain pattern are rarely (or never) manipulated, the model manipulation costs imposed by INC can be neglected.

(iii) Model dependent pattern characteristics

- **node type complexity**, a rough upper bound on the number of potential matches can be obtained as the product of the cardinalities (number of model instances) of the types of each node in the graph pattern. This estimate is, of course, accurate as there are also edges in the pattern to constrain the possible combinations of nodes. However, high complexity may result in high memory consumption for INC, and long search operations for LS.

- **model statistics** generally extend graph pattern static attributes to the entire instance model the transformation is working on. A well-known practical statistics on pattern complexity is the search space tree cost, that has already been used to adaptively select the search plan for LS-based matchers [VVF05]. It uses model statistics to assess the branching factors (node type complexity) during the search process. Other important factors like fan-out, hierarchy depth and model symmetries can also effectively make the estimation of match set sizes and time complexity of the pattern matching more precise.
6.4.2 Adaptive runtime optimization

Dynamic factors like memory consumption can quite easily change in-between transformation runs (even on the same system), especially using INC pattern matching, leading to performance degradation or insufficient memory. The current section focuses on an adaptive approach that can intervene in the predefined matching strategy in order to adapt to the altered environment.

In accordance with the general strategy described in Section 6.4.1, the adaptive engine generally prefers using the incremental pattern matcher for all graph patterns. When shortage of available memory is detected, pattern match set cache structures are gradually abandoned. For constructing such an adaptive approach monitoring, the following parameters are actually considered:

- During the execution of a ViATRA2 transformation the memory consumption is directly observable through the Java Virtual Machine (JVM), which provides a straightforward way for monitoring available memory.

- Simple model space statistics (e.g. the total number of model elements) are automatically registered by the ViATRA2 engine, along with sizes of match sets available from the incremental pattern matcher that can also be used as a model-specific indicator for actual memory consumption and to dynamically detect situations where run-time adaptive matching selection strategy switching is needed.

For the actual strategy the priority order for the cache removal is determined by the largest-first principle, where the pattern match cache structure with the largest overall memory footprint is selected for removal resulting in that the forthcoming pattern match operation requested for the corresponding pattern will always be executed by the LS-based pattern matcher leading to a smaller memory consumption. In our case, memory shortage is detected when the available heap memory is less than 15%, which initiates dropping PM caches and switching to LS strategy.

In order to evaluate the efficiency and impact of this approach, we ran the ORM benchmark experiment described in Appendix C.2.1 with the adaptive implementation. The results for this measurement were obtained in a different software environment: we used the 64-bit version of IcedTea 1.3.1 as a JVM (hence the larger memory consumption figures) based on an unique prototype ViATRA2 Release 3 build 2009.02.03. Execution times can be observed in Table 6.3.

Unsurprisingly, the execution time of the hybrid adaptive approach is between the fastest INC, the static hybrid approaches and a pure LS run. Note that memory was constrained for hybrid runs, marked with *; with memory constraints, INC would not run successfully in this case.

<table>
<thead>
<tr>
<th>PM strategy</th>
<th>Used heap [MB]</th>
<th>Transform phase time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS</td>
<td>201</td>
<td>77054</td>
</tr>
<tr>
<td>INC</td>
<td>353</td>
<td>13693</td>
</tr>
<tr>
<td>Static hybrid</td>
<td>220*</td>
<td>10958</td>
</tr>
<tr>
<td>Adaptive hybrid</td>
<td>235*</td>
<td>35716</td>
</tr>
</tbody>
</table>

Table 6.3: Match Set Memory and Performance of the Adaptive Hybrid Strategy

A more detailed comparison on the effects of the different factors on graph pattern matching performance in the AntWorld case study is available in [2].
Overall, this technique prevents the transformation engine from thrashing due to memory starvation, but is theoretically sub-optimal since the largest match set caches may not be the best choice for abandonment when optimizing for the shortest possible execution time.

A straightforward approach for future optimization is adjusting the priority order based on static analysis of the transformation program [10]. However, as of the beginning of 2013, we could not define precise metrics on which, such an adaptive technique could be based.

### Summary

We proposed a hybrid pattern matching approach, which enables the transformation designer to combine local search based and incremental pattern matching to adapt to memory limits. It is based on our general search plan representation where incremental pattern matching is introduced as a separate search plan operation. We demonstrated that in certain application scenarios the hybrid approach outperforms the other two approaches and provides a good balance between memory consumption and runtime performance.

However, we noticed that selecting the appropriate matching strategy for complex model transformation programs requires a deep understanding of both pattern matching algorithms. Therefore, by examining different transformation scenarios we have identified a list of various factors (metrics), which we experienced to have significant effect on run-time performance and memory consumption. Based on this analysis, we defined guidelines for transformation designers when a graph pattern should be matched using INC or LS algorithm.

Additionally, we proposed an automatic approach based upon runtime monitoring, where the execution engine could automatically switch from incremental pattern matching to a local-search based technique; thus reducing the overall memory consumption for degraded runtime performance.

### Limitations of the Hybrid Pattern Matching Approach

The presented hybrid approach is only a step forward for combining different pattern matching algorithms and in its current form has certain limitations. One of its main drawbacks is that it cannot support arbitrary combination of LS and INC pattern matching strategy for composite graph patterns. The hybrid engine uses the incremental matcher as a black box component and thus any graph pattern matched incrementally (including its invoked patterns) will has to be matched using the INC matcher. This way the transformation designer has to carefully structure the composite graph patterns in order to achieve the advantages of the hybrid approach. Additionally, in its current form the hybrid approach has a very limited support for adaptively changing the matching strategy for certain graph patterns during the transformation program execution and thus, in real world case studies, the designer has to precisely define the matching strategy on a pattern by pattern basis in order to achieve the required advantages.
Summary and Related Work in Graph Pattern Matching

7.1 Related Work

Pattern matching plays a key role in the efficient execution of all model transformation engines. In case of graph transformation based approaches, the goal is to find the occurrences of a graph pattern, which contains structural as well as type constraints on model elements. During pattern matching, each variable of a graph pattern is bound to a node in the model such that this matching (binding) is consistent with edge labels, and source and target nodes of the model.

Local Search based Approaches

Fujaba [NNZ00] performs local search starting from the node selected by the system designer and extending the matching step-by-step by neighbouring nodes and edges. Fujaba fixes a single, breadth-first traversal strategy at compile-time (i.e. when the pattern matching code is generated) for each rule. Fujaba uses simple rules of thumb for generating search plans. A sample rule is that navigation along an edge with an at most one multiplicity constraint precedes navigations along edges with arbitrary multiplicity.

PROGRES [SWZ99] uses a very sophisticated cost model for defining costs of basic operations (like enumeration of nodes of a type and navigation along edges). These costs are not domain-specific in the sense that they are based on assumptions about a typical problem domain on which the tool is intended to be used. Operation graphs of PROGRES, which are similar to search graphs in the current paper, additionally support the handling of path expressions and attribute conditions. The compiled version of PROGRES generates search plan at compile-time by a greedy algorithm, which is based on the a priori costs of basic operations.
The pattern matching engine of GReAT [KAS03] uses a breadth-first traversal strategy starting from a set of nodes that are initially matched. This initial binding is referred to as pivoted pattern matching in GReAT terms. This tool uses the Strategy design pattern for the purpose of future extensions and not for supporting different pattern matching strategies like in our approach.

GrGen.NET [JBK10] is based on plan graphs – similar to search graphs – for capturing the constraints defined by a graph pattern. From these plan graphs the search plans are generated using a simple heuristic optimization algorithm that minimize the most significant term occurring in its cost function defined as a product of the costs of the simple search operations. Their main advantages are that (i) their cost values are calculated from the underlying instance model and thus provides a precise prediction on the search space tree and (ii) the overhead for calculating the simple algorithm is very small. In overall the GrGen.NET framework is among the fastest model transformation frameworks when comparing pure graph pattern matching performance.

Varró et al. [VDWS12] propose a novel model-sensitive optimization approach based on dynamic programming. Its main advantage that it can handle arbitrary n-ary constraints and combined with the model-sensitive cost calculation for simple operations it can produce (at least theoretically) optimal solutions for complex cost functions. In their implementation this feature is exemplified with his own cost function also used in our own approach.

Giese et al. [GHS09] introduces a novel greedy approach that does not generate sophisticated search plans but instead uses an interpreted, dynamic approach, which always selects the next reference to traverse that contains the lowest number of target elements. One of its key advantage is that it always takes into account the underlying model when executing pattern matching and thus in general provides a good performance on various instance models. However, it is important to mention that the presented greedy approach may provide suboptimal pattern matching performance, where no bound input parameter is defined and thus the selection of the starting node is based only on their number in the underlying instance model. Additionally, some of the authors [BGST05] also provided an algorithm for calculating worst-case-execution-time for their pattern representation (called story diagrams) regardless of the applied local-search based pattern matching algorithm. This helps to apply their approach in a hard-real time environments, which is an evolving area in the model@runtime community.

**Constraint satisfaction based** Algorithms that handle pattern matching as a constraint satisfaction problem (CSP) like [LV02] in AGG [ERT99] do not directly involve the concept of search plans. However, the underlying constraint solver engine has to define a variable binding order, which can be considered as a search plan derived dynamically at run-time. As a consequence, CSP-based graph transformation engines by their nature support that dynamicity that has been achieved by our approach for local search based algorithms. However, as constraint solver implementations typically use the first-fail principle for determining the variable binding order, this technique still schedules the attribute, injectivity and NAC checking operations to the earliest possible location.

**Incremental Model Transformation Approaches**

Varró et al. in [VVS06b] proposes a graph pattern matching technique, which constructs and stores a tree for partial matchings of a pattern, and incrementally updates it, when the model changes. As a novelty, notification arrays are introduced for speeding up the identification of such partial matchings that should be incrementally modified. The main advantage of this solution is that only matchings, which appear as leaves of the tree, have to be physically stored, which possibly saves a significant amount of memory.
The model transformation tool TefKat includes an incremental transformation engine [HLR06] that also achieves incremental pattern matching over the factbase-like model representation of the system. The algorithm constructs and preserves a Prolog-like resolution tree for patterns, which is incrementally maintained upon model changes and pattern (rule) changes as well.

Giese et al. [GW06] present a triple graph grammar (TGG) based model synchronization approach, which incrementally updates reference (correspondence) nodes of TGG rules, based on notifications triggered by modified model elements. Their approach share similarities with our RETE based algorithm, in terms of notification observing, however, it does not provide support for explicit querying of (triple) graph patterns.

As a new effort for the EMF-based model transformation framework ATL [JT10], incremental transformation execution is supported, including a version of incremental pattern matching that incrementally re-evaluates OCL expressions whose dependencies have been affected by the changes. The approach specifically focuses on transformations, and provides no specific incremental query interface as of now.

VMTS [MMLA10] uses an on-line optimization technique to define (partially) overlapping graph patterns that can share result sets (with caching) during transformation execution. Compared to our approach, it focuses on simple caching of matching result with a small overhead rather than complete caching of patterns.

Hybrid Approaches

Up to our knowledge, there has been no other graph transformation system that applied any similar hybrid approach for graph pattern matching. However, similar concepts have already been applied in neighbouring research domains:

Expert Systems  As a conceptual analogy for our hybrid approach, research in expert systems [WM03] demonstrated that an integration between two different incremental strategies can be advantageous with respect to memory consumption and execution time. While the successful RETE algorithm has numerous variations itself, there are also several alternatives, many of which more or less resemble the idea behind RETE. The most important target of improvement is the high memory consumption of the RETE network.

TREAT [ML91] aims at minimizing memory usage while retaining the incremental property of pattern matching and instant accessibility of conflict sets. Only the input facts and the conflict sets (match sets) are stored, no memories are used for partial patterns. Up to now it seems that these algorithms are more or less equal in speed and memory consumption [NGR88].

Finally, the LEAPS algorithm [Bat94] is a fully incremental approach with minimal cache; similarly to TREAT, no partial matches are stored, only the match sets. Its main novelty is its lazy evaluation approach to avoid manifesting tuples unnecessarily, and by the introduction of timestamps to be able to reconstruct earlier conditions (“time travel”) for the lazy evaluation. It is believed that currently the LEAPS algorithm provides the best performance among rule based systems [Bat94].

Relational Databases  Additionally, in the context of relational databases, the cached result of a query is called a materialized view. These materialized views then can be used in SQL queries to speed up execution time as commercial database engines provide this feature along with an option of automatic and incremental maintenance. This results in a conceptually similar hybrid approach as certain parts of the query are stored in caches while other segments are evaluated at execution (querying) time (similarly to the local search based pattern matcher). However, in main stream
databases this non-standard feature of materialized views is typically restricted to a subset of SQL queries which is insufficient to express complex graph patterns (especially NACs).

Some novel results

7.2 Summary

To sum up, I propose a hybrid pattern matching approach by combining local search and incremental pattern matching using a generalized search graph concept based upon hypergraphs to represent the various search operations. My results on graph pattern matching are formulated as thesis contributions as follows:

1. **Generalized search graphs concept.** I defined a general search graph representation based on hypergraphs \([4, 18]\), which can guide graph pattern matching for advanced graph patterns like edge identities, type variables, negative application conditions, attribute conditions, and injectivity constraints. Based on this representation, all search plan operations are uniformly represented as special predicates with heuristically assigned costs (see in Section 5.1).

2. **Hybrid graph pattern matching.** I elaborated a hybrid graph pattern matching algorithm \([2, 16]\), which is able to combine local search based and incremental graph pattern matching algorithms to select a good graph pattern matching strategy for composite graph patterns (presented in Section 6.2).

3. **Identification and categorization of key-factors for matching strategy selection.** By analyzing the local search based and incremental graph pattern matching algorithms, I identified and categorized key factors of typical usage scenarios, which have significant impact on execution time and the selection of the matching strategy \([2, 16, 17]\) (see in Section 6.4.1).

Additionally, I demonstrated the feasibility of the advanced graph pattern matching approaches by experimental evaluation. The evaluation was carried on several graph and model transformation tool contest problems to demonstrate the effect of the various optimization techniques \([2, 27, 21, 24]\) (see in Section 6.3 and 5.6.3.1 and Appendix C).

The work on local pattern matching techniques presented in this thesis have been carried in cooperation with Gergely Varró. Varró laid down the basics of our search graph driven pattern matching approach in his thesis \([\text{Var08}]\) and also is the founder of the recursive pattern matching algorithm detailed in \([18]\). My own contributions in our cooperation are the generalization of the search graph concepts using hypergraphs to be able to represent any search operation in a generic way and the adaptation of the compile and runtime algorithms and data-structures to this novel hypergraph representation. The hybrid approach is completely my own contribution.

The identification and categorization of the key-factors for selecting proper matching strategy has been carried out with help from Gábor Bergmann. It is important to note that the main focus of Bergmann’s thesis is on developing and evaluating incremental pattern matching approaches, while my work aims at efficiently combining incremental and local-search based techniques.

Finally, we achieved further results with Gábor Bergmann for scaling model transformations to industrial size problems as presented in \([8]\).
Part III

Design Space Exploration
Design Space Exploration

8.1 Introduction

Evolutionary design space exploration

Design space exploration is a process to analyze several “functionally equivalent” implementation alternatives, which meets all design constraints in order to identify the most suitable design chosen based on various quality metrics such as performance, cost, power, reliability, etc. Typically, the best solution is flexible in the sense that it provides a trade-off between the optimal solutions with respect to a single quality metrics. Design space exploration is thus a challenging problem in many application areas including critical embedded systems or IT system management, where model-driven engineering (MDE) techniques have already been quite popular. Design space exploration in an MDE context is frequently tackled as specific sort of constraint satisfaction problem [Nee01].

Traditionally, most of these constraints and quality attributes were numeric in nature to express time, throughput, budget, memory limits, etc. However, the birth of modular software architectures in critical systems (like AUTOSAR [AUT] in the automotive or IMA in the avionics domain) introduced a novel type of complex structural constraints, which express connectivity restrictions for the graph-based model of the system under design. Complex structural constraints may include restrictions on allocation (e.g. separate critical components from non-critical ones), communication (e.g. use a secure communication channel between two channels), etc.

In addition, in many practical models@Runtime scenarios (like IT system management or service oriented architectures), design space exploration is further complicated by the continuous evolution of the system, which imposes further constraints and quality metrics. For instance, in IT system management and service-oriented architecture, both the actual system, the quality of service requirements and measured parameters, and reconfiguration policies may change quite frequently. Moreover, design space exploration also needs to incorporate the “distance” between the current and the designated configuration, as a reconfiguration to the mathematically “optimal” system configuration may
be too complex or costly to implement. In the current chapter, we aim to tackle evolutionary design space exploration to flexibly identify the most suitable design meeting complex structural constraints and numeric constraints where the underlying constraints may evolve in time, and the evolution of the best design is also restricted by allowed operations and/or quality metrics.

Solving the constraint satisfaction problem over models

The aim of the constraint satisfaction problem (CSP) is to find a solution to a set of constraints that impose conditions which have to be satisfied by a set of variables. Each variable takes its value from a predefined domain. A solution is one (or all) assignment of variables which satisfy each constraint.

Constraint satisfaction techniques have been successfully applied for various problems of model-driven engineering such as to apply design patterns [EBM08], to support domain-specific modeling [WSNW07] or model transformations [PBM09]. As a commonality, all these approaches translate high-level models to an existing, off-the-shelf constraint solver (like, e.g., [Inta, O]) to provide embedded design intelligence for modeling.

However, advanced constraint solvers typically apply certain restrictions for the CSP problem. For instance, the domains of variables are frequently required to be (a priori) finite; moreover, most approaches disallow the dynamical addition or retraction of constraints [MS00]. Furthermore, mapping graph models obtained in model-driven engineering to variables with finite domain can be a non-trivial task, especially when considering the evolution of models. As a summary, existing constraint solvers fail to adequately handle flexible and dynamic structural constraints over graph-like models, which is necessitated for evolutionary design space exploration.

Model-driven techniques for solving the CSP over models

Since model-driven engineering techniques are widely used in our designated application areas, it is worth evaluating how existing model or graph-based techniques could be used to solve dynamic and flexible constraint satisfaction problems with complex structural constraints.

Unfortunately, traditional model transformation tools (like ATL [ATL]) do not support backtracking when executing a model transformation for performance reasons, and thus they cannot traverse alternate transformation paths. Rare exceptions (like PROGRES [Sch90] which support backtracking) need complex control structures to drive the transformation, lack support for the efficient exploration of an alternate path after backtracking, and fail to handle dynamic changes of constraints or rules.

Sophisticated model or graph based verification tools (like GROOVE [Ren04a] or Alloy [Jac02]) need to store the entire state space during traversal, which is very resource consuming. Furthermore, they usually use generic bounded state space traversal strategies, which makes it difficult to fine-tune and effectively control how the most promising next candidate should be selected with respect to the CSP problem itself. As a summary, current model transformation based tools do not address this problem domain effectively and solvers based on these tools require additional fine-tuning for effectively handling structural constraints over graph–like models.

Contributions

In order to address this problem, we investigated how advanced model transformation technology can contribute to solving dynamic constraint satisfaction problems with global constraints over the domain of model graphs. We extended the definition of constraint satisfaction problems by using graph patterns to define structural (first-order logic) constraints, and graph transformation rules [Roz97].
as labeling operations referred. These labeling operators are used to carry out model manipulations on the underlying instead of simple variable substitution.

We defined three different solution criteria that addresses both standard, flexible and dynamic constraint problems:

- Informally, for a standard problem all graph pattern constraints need to be satisfied by the underlying model when searching for a specific goal.

- Flexible CSP supports the relaxation of constraints (referred to as soft constraints) to accept solutions that do not satisfy all given constraints. Our flexible approach uses a numeric weight function to capture the satisfiability criteria of the solution state, thus allowing the relaxation of constraints on a fine-grained state-by-state basis. The weight function also gives flexibility for the definition of an optimal solution.

- Dynamic CSP addresses the case when the original problem definition itself is changed (e.g. a constraint, or operation is added or removed), and our intention is to find a new solution in an incremental way, i.e. without restarting the solving process from scratch.

Furthermore, we developed a prototype constraint solver on top of the VIATRA2 [VB07] model transformation framework by using incremental constraint evaluation and various search strategies and heuristics. We evaluated our solver using two allocation problems taken from the avionics and the virtualized infrastructure domains. Finally, we compared the performance of our approach with existing (academic and industrial) tools.

The relevance of our work to model-driven engineering is threefold.

1. It defines standard, dynamic and flexible constraint satisfaction problem with complex structural and numeric constraints over graph-based models as means to formalize evolutionary design space exploration problems.

2. It provides an intuitive way to capture evolutionary design space exploration problems using techniques (e.g. graph patterns and graph transformation rules) which are closely related to MDE best practices.

3. It proposes actual solving strategies using incremental model transformation techniques, which are especially suitable for automating dynamic and flexible constraint solving for complex structural constraints.

Structure

The rest of the chapter is structured as follows: Section 8.2 presents two motivating case studies from the avionics and the service allocation domains, in Section 8.3 we introduce our graph pattern and transformation based constraint solver, while Section 8.4 extends the formalism with support for flexible and dynamic constraint problem definition. Section 8.5 introduces optimization and implementation details of our solver and performance measurements are evaluated in Section 8.6. Finally, related work is assessed in Section 8.7 and Section 8.8 concludes our work.

8.2 Motivation

System modeling and design space exploration are key issues in the design and synthesis of complex embedded and IT systems. Model-Driven Engineering has already contributed languages and tools
for capturing high-level system models and design constraints using graph-based models. However, in early phases of design, models are not sufficiently detailed to serve as an input for automated synthesis tools. In fact, in practice, the design space is constituted by multiple models representing different valid design candidates. The design space exploration process aims at searching through these candidates defined in the design space to find solutions that satisfy the requirements (constraints) and provide a balanced choice with respect to (a combination of) quality metrics. These complex exploration processes involve both critical design decisions made by the system architect and semi-automated techniques.

To introduce our CSP(M) formalism and demonstrate how it can help in solving various design space exploration problems, we selected two motivating allocation case studies from the mission critical embedded system in Section 8.2.1 and the service allocation over virtualized infrastructure Section 8.2.2 domain, derived from our ongoing research projects. The embedded system case study describes a typical design space exploration problem with static, non-flexible constraints. The service allocation on virtualized infrastructure case study represents a typical models@Runtime problem, where the requirements of system evolves over time any changes may occur to any part of its definition. These kind of problems can be handled using evolutionary design space exploration, where the solver can modify its constraint set according to the changes. Throughout the chapter, we will use these motivating case studies as our running examples and benchmarks.

8.2.1 Case Study: Allocation of an IMA system

Let us assume an integrated modular avionics (IMA) system composed of Jobs (also referred as applications), Partitions, Modules, and Cabinets. Jobs are the atomic software blocks of the system defined by their memory requirement. Based on their criticality level jobs are separated into two sets: critical and simple (non-critical). For critical jobs, double or triple modular redundancy is applied while for simple ones only one instance is allowed. Partitions are complex software components composed of jobs with a predefined free memory space. Jobs can be allocated to the partition as long as they fit into its memory space. Modules are SW components capable of hosting partitions. Finally, Cabinets are storages for a maximum (in our example) two modules used to physically distribute elements of the system. Additionally, a certain number of safety related requirements will also have to be satisfied: (i) a partition can only host jobs of one criticality level and (ii) different instances of a certain critical job cannot be allocated to the same partition and module. The task is to allocate an IMA system defined by its jobs and partitions over a predefined cabinet structure and to minimize the number of modules used.
8.2. MOTIVATION

(a) Starting model

(b) Allocated model

Figure 8.2: Example IMA system

A sample system composed of a critical job with two instances and two partitions with a single cabinet is shown in Figure 8.2(a) with a possible allocation depicted in Figure 8.2(b) defined over the metamodel captured in the VPM formalism [VP03] in Figure 8.1. Newly created elements are highlighted in grey.

8.2.2 Case Study: Service Allocation over Virtualized Infrastructure

Let us assume a synthetic virtualized platform providing a database service (as a typical models@Runtime scenario [YWGG11]). The system is composed of virtual and physical servers running a heterogeneous database infrastructure. Virtual servers are hosted by physical ones, where each physical server can host a predefined number of virtual ones. In the current configuration the virtualized platform uses three different types of fictitious databases to provide its service, namely: DBP, DB_C, and DB_V. A server can host at most one database at once and a physical server can either hold virtual servers or a database. Each database has different performance characteristic with regard to its underlying server captured by the following rules: (i) in general, DBs on virtual servers are performing almost half as fast as on physical, (ii) DB_V is slightly faster than the other two on virtual servers and also DB_C performs better than DB_P, (iii) however, DB_P is almost twice as fast on a physical server than the others and finally (iv) DB_C supports rapid clustering, where two instances can form a cluster-pair that counts as an additional virtual instance in their overall performance. The VPM metamodel of the service allocation case study is depicted in Figure 8.3.

The task is to allocate or in case a change occurs in the problem definition reallocate databases to produce the required overall performance over a physical server infrastructure, where both the number of licences for each database types and the required overall performance are predefined. The main difference between this and the IMA allocation is that in the current case not necessarily all databases are needed to be allocated to achieve a solution state. However, as business needs require, changes may occur in the problem definition over time that would require the reallocation of the databases.

A simple virtualized service configuration composed of two physical servers with one and two hosted virtual servers along with one DB_P and two DB_C databases is depicted in Figure 8.4. Two possible solutions with a required overall performance of 15 are also depicted in Figure 8.5(a) and Figure 8.5(b). The solution in Figure 8.5(a) uses the DB_P and a single DB_C database running over a
8.3 Constraint Satisfaction Programming

In this section, we provide a detailed description of our constraint satisfaction framework and its conceptual foundations and demonstrate how to apply it on the IMA system allocation problem introduced in Section 8.2.1.

8.3.1 Constraint Satisfaction Problem over Variables of Finite Domain

A CSP(FD) is a problem composed of a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restricts the values the variables can simultaneously take. In a more precise way a constraint satisfaction problem is a triple: \((Z, D, C)\) where \(Z\) is a finite
set of variables $x_1, x_2, ..., x_n$; $D$ is a function which maps every variable in $Z$ to a set of objects of arbitrary type; and $C$ is a finite (possibly empty) set of constraints on an arbitrary subset of variables in $Z$. The task is to assign a value to each variable satisfying all the constraints. Solutions to CSPs are usually found by (i) constraint propagation: a reasoning technique to explicitly forbid values or domains for variables by predicting future subsequent constraint violations and (ii) variable labeling: searching through the possible assignments of values to variables already restricted by the (propagated) constraints.

8.3.2 CSP(M): Constraint Satisfaction Problem over Models

An overview of the input and output artifacts of our CSP(M) formalism is depicted in Fig. 8.6. A CSP(M) problem consists of:

- An initial model representing the starting point of the problem. With the initial model the user can put additional knowledge into the system to give hint (e.g., in the form of a partial solution) to the solving process. This is a typical use case in design space exploration of embedded systems, where the system architect either reuses earlier solutions or use standard architecture patterns to start the evaluation from. Note, that the initial model can also be empty.

- The goal representing the conditions that need to hold in a valid solution of the problem. For example, in model-based modular embedded software design this can mean a certain level of redundancy that the system needs to implement or a connectivity restriction on the communication network of the system.

- A set of global constraints representing a special subset of constraints that needs to be satisfied by all models (states) traversed during the search for a solution. The use of global constraint is not mandatory, but they can effectively prune the search space by early detection of invalid models. For example, when allocating software components a global constraint can define the maximum number of allowed components on a CPU. This can have two advantages: (i) pruning out all invalid models where too many components were allocated to a CPU and also (ii) ensures valid models when the sequence of applied rules are also interesting for defining the operations to achieve the solution model.

- A set of labeling rules capturing the permitted operations. These labeling rules are conceptually similar to operations in planner algorithms [Wel94], which aim to restrict the possible transitions in the search space. For example, in case of software component allocation a labeling rule
can describe the underlying model manipulations required to allocate a free (non-allocated) SW component to a CPU.

- Finally, the solution of the exploration is called a solution model, which satisfies the goals and is achieved from the initial model through a sequence of labeling rule applications, where each intermediate state fulfills the global constraints. Additionally, in many application scenarios this sequence of applied labeling rules is also part of the solution (e.g., how to do the reallocation of services rather than just to have the final configuration).

**Definition 20** Formally, a CSP(M) \((M_0, C, G, L) : (M_s, T)\) is a structure where: \(M_0\) is the initial model; \(C\) is a set of global constraints; \(G\) is a set of subgoals which together in conjunction form the goal; and \(L\) is a set of labeling rules. The output \((M_s, T)\) is the solution that satisfies:

1. \(M_0 \sim M_s\); there exists a trajectory \(T : M_0 \xrightarrow{l_1} M_1 \xrightarrow{l_2} .. \xrightarrow{l_s} M_s\) where \(i = 1..s : l_i \in L\). Informally, \(M_s\) is reachable from \(M_0\) through a sequence of applied labeling rules in trajectory \(T\).

2. \(\forall G_i \in G : M_s \models G_i\); \(M_s\) satisfies all subgoals \(G_i\)

3. \(\forall C_i \in C : M_s \models C_i\); \(M_s\) also satisfies all global constraints \(C_i\)

4. \(\forall M_i \in T, \forall C_j \in C : M_i \models C_j\); along the trajectory \(T\) from the initial to the solution model all visited model \(M_i\) satisfies each global constraint.

As models in MDE are usually described as graphs we instantiate our formalism on graph transformation a well-known model transformation language. In our instantiation both the initial and solution models are defined by typed graphs over a given metamodel. Based on this metamodel we use graph patterns to declaratively define both goals and global constraints. This way constraints are directly defined over the problem domain and no mapping to other formalisms (e.g., finite domain constraint logic programming) is required. Finally, model manipulation operations described by the labeling rules are captured by graph transformation rules. Altogether, the complete problem can be defined in a declarative manner using model-driven techniques making the whole formalism intuitive, especially for complex structural constraints.

Additionally, this instantiation allows to directly apply the GT-defined labeling rules on the underlying (graph) models, giving way to a better insight of the solving process with potential feedback on (i) valid and invalid goals and global constrains and (ii) applicable labeling rules in each states, allowing easier traceability of the solving process.

For the concrete definition of CSP(M) problems we used the ViATRA2 [VB07] transformation language without any restrictions on the used language constructs. However, this formalism can also be incorporated into other modeling approaches such as MOF models, OCL constraints and QVT rules.

### 8.3.2.1 Goal and Global constraints

Both subgoals and global constraints are defined by graph patterns. The goal \(G\) is the conjunction of subgoals where a subgoal (graph pattern) is a disjunction of alternate pattern bodies.

**Definition 21** A subgoal or global constraint \(C\) described by the graph pattern \(GP\) is either a positive or negative constraint. A negative constraint is satisfied by a model \((M \models C)\) if it does not have
a match in $M$, formally $\not\exists m : GP \rightarrow M$. While a positive constraint is satisfied if its representing graph pattern has a match in $M$: $\exists m : GP \rightarrow M$. A further restriction on positive constraints can be formulated by stating that they are satisfied if their representing graph pattern has a predefined minimum number of matches ($\text{Cardinality}$), formally $|\{m | m : GP \rightarrow M\}| \geq \text{Cardinality}$. In our IMA case study all patterns are considered as negative constraints.

Figure 8.7: Goal, Labeling rules and Global constraints of the IMA case study
8.3.2.2 Labeling rules

Definition 22 Labeling rules are described as graph transformation rules. A labeling rule \( l \) is enabled when the precondition \( LHS_l \) of its representing graph transformation rule is applicable to the underlying model \( M \), formally \( \exists m : LHS_l \rightarrow M \). However, additional properties are used to refine the execution order and semantics of an enabled rule application:

- **Priority (integer: 0..100):** Defines a precedence relation on labeling rules. It organizes the labeling rules into sets based on their priorities. In each state the solver selects its next step from the set with the highest priority. In our IMA case study we use the same priority for all labeling literals.

- **Execution mode (forall | choose):** Defines whether a rule is simultaneous applied at all possible matches (forall) (as a single transition) or only once on a randomly selected single match (choose). In the IMA case study all labeling rules are using choose type execution mode.

Example 20 Our IMA case study formalized as a CSP(M) problem is depicted in Figure 8.7. The jobInstancewithoutPartition, partitionwithoutModule and modulewithoutCabinet subgoals formulating the goal describe that in a solution model each JobInstance, Partition and Module is allocated to a corresponding Partition, Module and Cabinet, respectively. For example, the jobInstancewithoutPartition subgoal captures its requirement using a double negation (NAC and negative constraint) stating that there are no unallocated job instance JI in the solution model. Similar double negation is used in case of the other two subgoals.

Global constraints formulate the safety and memory requirements. The partitionMemoryHigherThan0 pattern captures the simple memory constraint that all partitions must have higher than zero free memory. The safety requirement stating that a partition can only host jobs of one criticality level is captured by the partitionCriticalityLevelSimilar pattern. As it is a negative constraint it describes the (positive) case where the P1 partition holds two job instances J1 and J2 of a simple and a critical job Job1 and Job2, respectively. The criticalInstanceonSamePartition and criticalInstanceonSameModule patterns imply in a similar way that no job instances J1 and J2 of a critical job Job can be allocated to the same partition P1 or module M1.

Finally, labeling rules describe the allocation operations. The allocatePartition graph transformation rule defines how a partition P can be allocated to a module M1. As a common technique in graph transformation based approaches, a negative application condition stating that the partition is not already allocated is used to indicate that the rule should only be used for unallocated partitions. On top of that the allocateModule rule uses an additional NAC to forbid allocation of module M to cabinet C1 when two other modules M1 and M2 are already presented on C1, while the allocateJobInstance defines an additional attribute operation to decrease the free memory value MP of partition P1 by the required memory MJ of the allocated job J. The createModule rule simply creates a module M without any precondition.

8.3.3 Solving CSP over Models

To traverse the search space of a constraint program introduced in Section 8.3.2, we define the solver as a virtual machine that traverse the state space consisting of states and transitions.

Definition 23 A state is a 4-tuple \((CG, CS, AM, LS)\), where
8.3. CONSTRAINT SATISFACTION PROGRAMMING

- **CG** is called the **current goal** that stores the subgoals that still need to be satisfied;
  \[ \forall G_i \in CG : G_i \in G, AM \not\models G_i \]

- **CS** is the **constraint store** that holds all constraints the solver has satisfied so far;
  \[ \forall C_i \in CS : C_i \in G \cup C, AM \models G_i \]

- **AM** is the **actual model** that represents the underlying actual model.

- Finally, **LS** is the **labeling store** that contains all enabled labeling rules. An element in the labeling store is a pair \((l, m)\), where \(l\) is a labeling rule and \(m\) is a valid match of its precondition \(LHS_l\) in the actual model;
  \[ \forall (l_i, m_i) \in LS : l_i \in L, m_i \subset AM : m_i : LHS_{l_i} \rightarrow AM. \]

**Definition 24** A **transition** in the search space is a pair of 4-tuples of \((CG, CS, AM, LS) \rightarrow (CG', CS', AM', LS')\), which describes a step between two states. A transition is possible iff \( \exists (l, m) \in LS \) where \( AM \xrightarrow{\text{L}} AM' \); i.e., a labeling rule can be applied on the actual model for a certain match. A goal \( G \) can be proved if there exists a trajectory of individual steps \((CG, CS, M_0, LS) \sim (\emptyset, CS', M_s, LS)\) for a satisfiable constraint store \( CS \). In other words, a solution model is found if there exists a sequence of labeling rule applications that lead to an empty \( CG \) and satisfiable \( CS \).

In Algorithm 8.1 the skeleton of the solver algorithm is presented. Initially, the **root** state of the traversal is initialized with the **goal** \( G \), **global constraints** \( C \) and the enabled **labeling rules** \( L \) of the CSP(M) problem, respectively, while \( AM \) is set to the initial model \( M_0 \) (Line 1). The solving process consist of a repeat-until loop that is executed until either a solution is found (Line 13) or the complete state space is traversed without any solution states (Lines 5-7). During one iteration the traversal proceeds by selecting an enabled **labeling rule** and applies it on the **actual state** resulting in the **next state** (Line 4).

If the function cannot produce the next state (Line 5) it can either mean that the actual state is a dead end and the traversal continues with backtracking (Line 9) or the actual state is the **root state** and it means that the traversal ended without finding a solution model (Line 7).

If a new next state is produced \( CS \) is checked for consistency (Lines 11-23). Based on the result the traversal can continue in the following ways: (i) when a global constraint is violated (Line 11) the traversal backtracks to a previous state (Line 18), (ii) when it satisfies all global constraints but violates a subgoal (Line 12) it continues from the new state (Line 15) and finally, (iii) if both the global constraints and subgoals are satisfied the solution state is returned (Line 13).

The **getNextState** function simply selects an enabled labeling rule from the labeling store of the actual state based on the traversal strategy (see in Section 8.5.1) and applies on the \( AM_{curr} \) model resulting in the \( AM_{next} \) model. After that all other segments of the state is calculated and the next state is returned. In case there are no enabled labeling rule in \( LS_{curr} \) the function returns null.

Note that, as in general termination of a graph transformation system (GTS) is undecidable [Plu05] and CSP(M) instantiated over graph transformation behaves exactly as a GTS – in case there is no solution it needs to traverse the complete state space similarly to GTS – it is also undecidable.

**Example 21** Let us consider that our IMA case study is in the initial state \( S_0 \) depicted in Figure 8.8. The actual model is the initial model \( M_0 \) (detailed in Figure 8.2(a)); the current goal \( CG \) contains the **jobInstanceWITHOUTPartition** and **partitionWITHOUTModule** subgoals; the constraint store
CHAPTER 8. DESIGN SPACE EXPLORATION

PROCEDURE solve($\text{CSPM}(M_o, C, G, L)$) \text{\triangleright} $M_o$ is the starting model, and $C$ holds the global constraints, $G$ holds the subgoals and finally $L$ holds the labeling rules

1: $\text{root} := \text{state}(G, C, M_o, \{∀l \in L, ∀m \subset AM \mid m : \text{LHS}_l \rightarrow AM\})$
2: $\text{currentState} := \text{root}$ \text{\triangleright} Initializes the root state with the input definition of the CSP(M) problem
3: repeat
4: $\text{nextState} := \text{getNextState}(\text{currentState})$ \text{\triangleright} Selects and executes a valid transition
5: if ($\text{nextState} == \text{null}$) \text{\triangleright} No fireable transitions is available from the currentState
6: if ($\text{currentState} == \text{root}$) \text{\triangleright} Complete state space explored without a solution found
7: return no solution found
8: else
9: $\text{currentState} := \text{backtrack}(\text{nextState})$ \text{\triangleright} an intermediate state without a valid transition
10: end if
11: if ($\text{isValidState}(\text{nextState})$) \text{\triangleright} Checks that the nextState satisfies all global constraints
12: if ($\text{isSolution}(\text{nextState})$) \text{\triangleright} Checks that the nextState satisfies all subgoals
13: $\text{return} \text{trajectory}(\text{root}, \text{nextState})$ \text{\triangleright} and the sequence of applied labeling rules
14: else
15: $\text{currentState} := \text{nextState}$ \text{\triangleright} A valid state that do not satisfy the goal
16: end if
17: else
18: $\text{currentState} := \text{backtrack}(\text{nextState})$ \text{\triangleright} an invalid state that violates a global constraint
19: end if
20: until (Solution found or state space explored)

PROCEDURE $\text{getNextState}(CG_{curr}, CS_{curr}, LS_{curr}, AM_{curr})$ \text{\triangleright} $CG_{curr}$ is the current goal of the current state, and $CS$ is the current constraints of the current state, $LS$ holds the enabled labeling rule from the actual model $AM_{curr}$

23: $(l_r, m_r) = \text{selectLabelingRule}(LS_{curr})$ \text{\triangleright} Selects an enabled labeling rule from LS
24: if ($l_r == \text{null}$) \text{\triangleright} applies the labeling rule resulting in the $AM_{next}$ model
25: $AM_{next} = \text{applyLabelingRule}(AM_{curr}, l_r, m_r)$ \text{\triangleright} Calculates the current goal for the next state
26: $CG_{next} = \{∀C_i \in G \mid AM_{next} ∋ G_i\}$ \text{\triangleright} Calculates the constraint store for the next state
27: $CS_{next} = \{∀C_i \in G \cup C \mid AM_{next} \models G_i\}$ \text{\triangleright} Calculates the labeling store
28: $LS_{next} = \{∀l_i \in L, m_i \subset AM_{next} \mid m : \text{LHS}_{l_i} \rightarrow AM_{next}\}$ \text{\triangleright} Returns the next state
29: return $\text{NextState}(CG_{next}, CS_{next}, LS_{next}, AM_{next})$ \text{\triangleright} No enabled labeling rule is available from $AM_{next}$
30: end if

Algorithm 8.1: The CSP(M) solving algorithm
8.4 Flexible and Dynamic Constraint Satisfaction Problems over Models

Our formalism also supports dynamic and flexible constraint satisfaction problems. In the current section we introduce how these different CSP definitions are adopted in CSP(M).

8.4.1 Flexible Constraint Satisfaction Problems

Classical constraint satisfaction techniques support only hard constraints specifying exactly the allowed combinations. Hard constraints are imperative (a valid solution must satisfy all constraints) and inflexible (constraints are either entirely satisfied or violated). In order to overcome these weaknesses, flexible CSPs introduced soft constraints [BMR95a] to relax these assumptions and allow solutions that do not satisfy all constraints.

One well-known approach, called weighted CSP [DL85], introduces the use of weights attached to each constraint indicating its relative importance. A solution is acceptable if the sum of the weight
of the satisfied constraints is higher than a predefined value. This weight function allows to define what a good (or optimal) solution means with respect to the defined goals of the problem domain.

By extending the classical CSP(M) formalism (in Section 8.3.2) we define weighted CSP(M)

**Definition 25** A weighted CSP(M) is a \((M_0, C, G, L, f_w, S_w) : (M_s, T)\). \(S_w\) is the predefined sum weight required for a solution model to be satisfied and \(f_w : G_s, M \rightarrow \mathbb{N}\) is a weight function, which takes as input a subgoal \(G_s \in G\) and a model \(M\) and produces the weight of the subgoal \(G_s\) in the model \(M\). The weight function is usually specific to each problem domain and can use the additional attributes of the satisfiability criteria of the subgoals. For example, this can be the number of matches in a specific state or the cardinality value of a positive subgoal.

The definition of a solution \((M_s, T)\) changes in the following way:

1. \(M_0 \sim M_s\); there exists a trajectory \(T : M_0 \xrightarrow{l_1} M_1 \xrightarrow{l_2} \ldots \xrightarrow{l_n} M_s\) where \(i = 1 \ldots s : l_i \in L\). Informally, \(M_s\) is reachable from \(M_0\) through a sequence of applied labeling rules in trajectory \(T\).

2. \(\sum_{\{G_i | G_i \in CS \land M_s \models G_i\}} f_w(G_i, M_s) \geq S_w\); In a solution model \(M_s\), the summarized weight of the satisfied subgoals \(G_i\) has to be greater or equal to the predefined \(S_w\) value.

3. \(\forall C_i \in C : M_s \models C_i\); \(M_s\) also satisfies all global constraints \(C_i\)

4. \(\forall M_i \in T, \forall C_j \in C : M_i \models C_j\); along the trajectory \(T\) from the initial to the solution model all visited model \(M_i\) satisfies each global constraint.

**Definition 26** This way the solving process described in Section 8.3.3 is slightly modified to; a solution model is found if there exists a sequence of labeling rule applications, that leads to a constraint store that fulfills the inequality defined in 2 and contains all global constraints. This small modification only affects the isSolution method in Line 12 that has to calculate the aforementioned inequality.

### 8.4.2 Dynamic Constraint Satisfaction Problem

A further limitation of classical CSP is in its assumption of a static problem. This means that once the constraints have been defined they are fixed for the duration of the solving process. However, in certain cases like models@Runtime [YWGG11] problems are subject to change either as a solution is being constructed or while the constructed solution is in use. Classical CSP usually can deal with this situation by considering the changed problem as an entirely new problem which needs to be solved from scratch.

Dynamic constraint satisfaction [DD88] addresses this kind of problems and allows to add and remove constraints from the actual problem definition as necessary. However, to utilize the advantage of dynamic constraint manipulation and re-use partial solutions obtained for a problem before it changes, additional techniques [VS94] are required.

In our case, it is possible to dynamically add or remove global constraints, labeling rules and goals from a problem definition in a solution state. However, not all combinations are worth to be carried out as a dynamic constraint satisfaction problem with respect to solution re-use:

Global constraint
In case a global constraint $C_r$ is removed from the constraint store, then all previously visited states remain valid except those leaf states, that were invalidated by the constraint, need to be recalculated as potentially valid states. The original problem is redefined as $(M_0, C, G, L) : (M_s, T) \rightarrow (M_0, C \setminus \{C_r\}, G, L) : (M_s', T')$. In this case all already visited states are left as valid states of the new problem.

If a global constraint $C_a$ is added then all already visited states need to be re-evaluated with the new constraint, which is almost identical to a fresh state space exploration from the initial state of the original problem. This means that the new problem is $(M_0, C \cup \{C_a\}, G, L) : (M_s', T')$. Assuming that $VS$ is the set of the already visited states of the original problem, the invalidated visited states are $\{S_i | S_i \in VS\}$.

Labeling rule

If a labeling rule $L_r$ is removed, then all transitions that used this rule are invalid. It means that all visited states after these transitions are also invalid and must be deleted from the already visited states. Depending on the actual traversal, this might affect the entire visited state space or no states at all. Informally, the new problem is $(M_0, C, G, L \{L_r\}) : (M_s', T')$, where the invalidated states by the removed labeling rule are $\{S_i | S_i, S_{0..n} \in VS \land S_0 \sim S_j \xrightarrow{L_r} S_{j+1} \sim S_n \land j < i \leq n\}$.

In case a labeling rule $L_a$ is added, then similarly to the global constraints all previously visited states need to be re-evaluated with the new rule as it can potentially create new branches for the exploration. However, these states are not invalid; thus they can re-evaluated on demand only when the solver algorithm revisits these states. In this way the new problem is $(M_0, C, G, L \{L_a\}) : (M_s', T')$, where the states to be re-visited are $\{S_i | S_i \in VS\}$.

Labeling rules and global constraints can be treated similarly in case of classical and flexible CSP(M) problems. However, as the definition of a satisfying solution is different in both cases, different actions needed to be carried out when a subgoal is dynamically added or removed:

Goal  Classical CSP(M)

If a subgoal $G_r$ is removed, then the problem definition changes to $(M_0, C \setminus \{G_r\}, L) : (M_s', T')$ and all visited states have to be re-evaluated, $\{S_i | S_i \in VS\}$. However, these updates are rather simple as only the subgoal $G_r$ needs to be removed from either the current goal or the constraint store and this does not involve constraint evaluation (pattern matching). Additionally, solution states remain valid and states $S_j$ where $G_r$ is the only unsatisfied subgoal becomes solution states ($G_r \in CG_j \land |CG_j| = 1$).

In case a subgoal $G_a$ is added ($\langle (M_0, C, G \cup \{G_r\}, L) : (M_s', T') \rangle$), then all visited states have to be updated with constraint evaluation in each state. Similarly to an addition of a global constraint the problem becomes identical with a fresh state space exploration of the original problem.
CHAPTER 8. DESIGN SPACE EXPLORATION

Figure 8.9: Global Constraints, Goals and Labeling Rules of the Service Allocation Case Study

- If a subgoal $G_r$ is removed $(M_0, C, G \backslash \{G_r\}, L, f_w, S_w) : (M'_s, T')$, then similarly to the classical case all already visited states have to be updated, $\{S_i|S_i \in VS\}$. The complexity of the update mainly depends on the weight function $f_w$ as it has to be recalculated on each already visited state along with the deletion of $G_r$ from the constraint store or the current goal.

- Similarly to the case in classical CSP(M) all already visited states have to be updated with complete constraint evaluation and weight calculation when a subgoal $G_a$ is added to a flexible constraint definition $(M_0, C, G \cup \{G_a\}, L, f_w, S_w) : (M'_s, T')$.

- Additionally, a flexible CSP(M) problem $(M_0, C, G, L, f_w, S_w) : (M'_s, T')$ can be changed through its weight function $f_w$ and solution weight $S_w$. A change in the weight function $f_w$ cannot be treated as a dynamic manipulation in the problem definition as it requires a complete recalculation of all visited states, which is identical to a fresh state space exploration of the changed problem. However, if the solution weight $S_w$ is changed, all already visited states remain valid and the state space exploration can continue from the solution state of the original problem. Formally, the new problem becomes $(M_0, C, G, L, f_w, S'_w) : (M'_s, T')$.

In case more than one constraint, goal or labeling rule is added or removed from the problem definition, then the union of the effects described has to be carried out.

Overall, dynamic CSP(M) can effectively and incrementally solved by reusing the previous solution in the following cases: (i) elements are removed from the problem definition, (ii) the solution weight is modified in a flexible CSP(M) definition, or (iii) depending on the solver algorithm in cases where labeling rules are added.

8.4.3 The Service Allocation Case Study as a Flexible Constraint Satisfaction Problem

Our service allocation example, for a typical models@Runtime scenario, formalized (see in Section 8.2.2 as a flexible CSP(M) problem is depicted in Figure 8.9. Similar to the IMA example the labeling rules capture the operations of the allocation.
• The allocateDatabase rule allocates the database DB to a server S1 if it is not already allocated.

• The createDB_C_Cluster rule simply creates a cluster-pair through the inCluster relation between the DB and DB2 databases of type DB_C if DB is not already in a cluster.

• The last shutDownV Servers labeling rule is used to turn off a virtual server VS hosted by the physical server PS if no database DB is running on VS. Only this simple rule is required to model the server infrastructure as our initial model will represent the state where each physical server is hosting its maximum allowed number of virtual servers.

As mentioned in Section 8.4.1 a solution state is defined by its structural goal G, its weight function \( f_w \) and the required solution weight \( S_w \). In this example the weight function is \( f_w(G_i, M_j) = \text{perf}_{G_i} * |\{m : G_i \rightarrow M_j\}| \). It means that the weight of a subgoal \( G_i \) in state \( M_j \) is equal to the number of its matches in \( M_j \) multiplied by a predefined constant performance indicator \( \text{perf}_{G_i} \). The performance indicator is a relative value derived from the requirements to capture the performance characteristic of the different database types.

The goal is captured by six positive subgoals, each with its own performance indicator depicted by the number in their top right corner.

• The DBonPServer and the DBonVServer patterns with performance indicators of 7 and 4 set the average performance of a server running on a physical or a virtual server, respectively. Compared with these average values, the other four patterns formulate the relative performance difference defined in the problem specification.

• The DB_VonVServer and the DB_ConVServer patterns capture the requirement that the database type DB_V is faster, with a performance indicator of 2, than the other types.

• Additionally, the DB_PonPServer pattern describes that the database type DB_P performs almost twice as fast on a physical server than the other types.

• Finally, the DB_C_Cluster pattern defines that if two DB_Cs are running on different servers and form a cluster-pair then they produce an additional performance of 5.

The negative global constraints onlyOneDBperServer and DB0-or0-VS0-onPhysical specify that no server can hold more than one database and a physical server can hold either a virtual server or a database, respectively.

As the specification does not precisely define the performance differences between the databases, the current definition of the problem can be a subject to change. Possible changes to the problem definition are discussed in Section 8.4.3.1 along with the required dynamic manipulation to model them in our formalism.

8.4.3.1 Dynamic Problem Extensions

However, it is possible that the imprecise assumptions on performance, newer versions of databases or a change in business rules can slightly modify the problem definition and it requires changes in its CSP(M) definition. These changes can be treated as separate dynamic constraint satisfaction problems of our service allocation example. To simulate such modifications we defined three different changes. These three modifications represent the practically relevant cases, where dynamic reevaluation does not require a fresh state space exploration and previous solutions can be partially reused.
• Let us assume that the additional plus 1 performance indicator defined by the $DB_{ConVServer}$ pattern for the $DB_C$ database is no longer required and needs to be removed from the problem definition.

• It is also possible that a newer version of the $DB_C$ database supports not only cluster-pairs but also cluster-triplets, where the performance output is doubled compared to three single instances. This modification can be captured by the $createDB_C_ClusterTriplet$ labeling rule (depicted in Figure 8.10). The double performance is calculated by the fact that the $DB_C_Cluster$ pattern matches three times on a single cluster-triplet.

• Finally, a third variant of dynamic change can be that business reconfiguration is no longer available due to other services provided by the virtualized platform. This case can easily be handled by removing the $shutdDownVServer$ labeling rule from the definition.

To assess the performance aspects of dynamic CSP(M) problem changes, Section 8.6.2 gives a first experimental evaluation of the introduced dynamic capabilities based on the cloud case study implemented in our CSP(M) solver.

8.5 Optimization Strategies and Implementation Details

The current section describes several optimization and implementation considerations built into our prototype CSP(M) solver. Section 8.5.1 briefly introduces the different search strategies applied for the guided state space traversal, while Section 8.5.2 details optimization techniques to reduce the traveled state space and finally, Section 8.5.3 focuses on concrete implementation details.

8.5.1 Search (Labeling) Strategies

Most algorithms for solving CSPs systematically traverse the possible search space. Such algorithms (often called as search or labeling strategies) are guaranteed (in case of finite search space) to find a solution, if one exists, or to prove that the problem is unresolvable. In our Algorithm 8.1 description these strategies are responsible for the selection of the labeling rules from the labeling store ($selectLabelingRule$) in Line 22.

The most common algorithm for performing systematic search is backtracking based on depth-first search. Backtracking incrementally builds candidates to the solutions and abandons each partial candidate (“backtracks”) as soon as it determines that it cannot possibly be completed to a valid
solution. In our case it means that in the actual state a global constraint is violated or its labeling store is empty; thus the system backtracks to the last applied step and continue with a different one.

One of the main drawbacks of the simple backtracking algorithm is *thrashing*; i.e., repeated failure due to the same reason. Thrashing occurs because the backtracking algorithm does not identify the root cause of a conflict, i.e., the unsatisfiable global constraint or subgoal leading to a dead-end. Therefore, search in different parts of the search space keeps failing for the same reason.

In order to overcome thrashing we implemented two additional search strategies:

### 8.5.1.1 Random Backjumping

is a backtracking strategy based on the assumption that a traversal might be in a dead-end if no solution was found within a certain amount of time (deadline). When the solver exceeds this deadline, it jumps back to a state at least as high as the half of the actual depth of the search space tree. This way, the solver can restart the traversal from an earlier state and continue on different random transitions. However, to keep the completeness of the traversal we implemented a simple policy introduced in [BMS00] that is to increase the height of the backjump each time it is used. This approach is obviously not effective to prove unsatisfiability because all the runs except the last are wasted, but has a good average performance in certain real-world scenarios.

### 8.5.1.2 Guided traversal by Petri net abstraction

is a state space traversal strategy which conducts search towards the most promising candidate paths calculated according to a Petri net abstraction of graph transformation systems introduced in [VGV06]. It introduces temporal numerical cuts to guide the state space exploration by temporally pruning the state space to postpone the unpromising paths. By formulating the solution state configuration as submarking of the Petri net, we can solve an integer linear programming problem of the derived Petri net using its incidence matrix to obtain an optimal *transition occurrence vector* leading to a designated target state (formulated as a target submarking). A transition occurrence vector prescribes how many times a labeling rule needs to be applied in order to reach the derived submarking of a solution model. Then the search strategy first explores those branches (i.e. labeling rule applications) which are consistent with this hint. This means that if a graph transformation (labeling) rule is applied more than prescribed in the vector, then the exploration of its branch is postponed. If no solution is found on the level of CSP(M), then the next optimal transition occurrence vector candidate is derived, and the exploration of the CSP(M) problem continues.

Note that due to the abstraction, the transition occurrence vector might not represent a feasible trajectory in the search space of the CSP(M) problem. However, it provides a good lower bound on the minimal number of labeling rule applications required to reach a solution model if its corresponding solution submarking can be precisely estimated or calculated. The first transition occurrence vector calculated for our IMA example is \((2, 1, 1, 1)\) meaning that to achieve a solution submarking derived from a solution model where all job instances and partitions are allocated, the `allocateJobInstance` rule has to be applied twice while the other three only once.

It is important to mention that in case of flexible CSP(M) problems the estimation of the solution occurrence vector heavily depends on the weight function. Additionally, in case of dynamic CSP(M) problems, in each case the problem changes the abstraction needed to be updated and recalculated. This traversal technique becomes less useful in these cases.
8.5.2 Optimization

To further reduce the size of the traversed state space, we introduce two additional optimization techniques that complement our search strategies described in Section 8.5.1.

8.5.2.1 Look-ahead pattern

Additional restrictions on the applicability of labeling rules can be formulated by incorporating a subset of global constraints called look-ahead constraints into the precondition (LHS) of rules. These constraints are validated in the precondition of labeling rules to prevent unnecessary steps which would violate these constraints. Currently, this is a manual hint by the designer, but in the future, we plan to automate this task by applying critical pair analysis [HKT02] or transformations of graph constraints to preconditions [EEHP06].

In our IMA example the allocateJobInstance rule can be further restricted regarding the memory consumption of the JIns job instance making the partitionsMemoryHigherThan global (look-ahead) constraint obsolete. Its modified version with the extra check condition on the required and available memory is depicted in Figure 8.11. Similarly, the global constraint onlyOneDBpreServer can be integrated as part of the allocateDatabase labeling rule in the service allocation example.

8.5.2.2 Exception priority

In order to explicitly restrict the number of application of labeling rules along a trajectory we introduced a priority class called exception. Exception rules have the lowest priority and will only be selected when no other labeling rules are enabled. In any trajectory if the number of applications of an exception rule exceeds its predefined value the solver backtracks and continues along another transition. Exception rules are used as hints by the search strategy to avoid state explosion, especially when the Petri net based abstraction cannot predict the number of labeling rule applications for element creation rules without preconditions such as the createModule rule in the IMA example.

8.5.3 Implementation

We implemented an experimental solver for CSP(M) (called ViATRA2 DSE) including all the techniques above on top the ViATRA2 model transformation framework, which offers efficient rule- and pattern-based manipulation of graph models by the means of graph transformation. In order to
implement the solver using graph-based state representation we had to address the problems of constraint evaluation, backtracking and typed graph comparison.

- For effective evaluation of constraint satisfiability we rely upon the incremental pattern matcher component [BOR+08] of the framework. In case of incremental pattern matching, the matches of a pattern are stored to be readily available in constant time, and they are incrementally updated when the model changes. As matches of patterns are cached, this reduces the evaluation of constraints and preconditions of labeling rules to a simple check. This way, VIATRA2 DSE has an incrementally maintained up-to-date view of its constraint store and enabled labeling rules. Furthermore, incrementality provides an efficient constraint propagation technique to immediately detect constraints violations after a labeling rule is fired.

- For backtracking between states, we implemented a simple transaction mechanism that saves the atomic model manipulation operations applied on the model in an undo stack. This stack not only allows us to backtrack the manipulations but also eases the computation of difference between neighbour states. However, the undo stack based implementation also has a drawback as backtracking is only possible from the actual state upward to the root and no jumping is supported between different paths of the state space. This means that traversal algorithms in the state space needs to follow a depth-first strategy.

- To be able to detect already visited states, we needed to store and compare states represented by graphs as whenever the solver traverse a new state it also checks that it has not already visited this state.

  For fast graph comparison we adapted the DSMDIFF [LGJ07] algorithm, which relies on (i) signatures (for nodes and edges) composed of type and name information and (ii) containment relations between nodes of the graph, both supported by VIATRA2. However, the general algorithm did not scale well with large models, especially when a significant part of the model is static and cannot change during evaluation but is always compared between states. To overcome this problem, we defined a domain-specific model comparator based on the general DSMDIFF algorithm. This new algorithm (i) compares only non-static parts of the model and (ii) the user can restrict elements (from the metamodel) to be used for the model comparison. In the current implementation these comparators are hand coded for each domain (meta)model.

- Finally, to keep the memory consumption low, we stored already visited states in a serialized form using a simple breadth-first algorithm and applied our graph comparison algorithm directly on this representation. Additionally, to reduce the number of candidates for comparison we also applied a hash function based on the number of elements on each level of the model containment hierarchy. However, to further reduce the number of comparisons the use of domain-specific hash functions are also supported by our implementation. Note that these domain-specific hash functions also have to satisfy the condition that, if two models are equal then their hash values are also equal.

The introduced VIATRA2 DSE framework was used in the context of the DIANA [DIA] European project as its underlying allocation engine for a system-level integration scenario for avionics software allocation. More details are available in Section 10.5.1.
8.6 Evaluation

To evaluate the performance of our CSP(M) solver, we carried out experiments both on our IMA (in Section 8.6.1) and service (in Section 8.6.2) allocation case studies for classical and dynamic/flexible CSP(M), respectively. Moreover, in order to compare our results with other available tools, we selected three from closely related fields:

- **Standard CSP** tools over finite integer domains are the most widely used and general purpose constraint solvers available. For our evaluation we selected the commercial SICStus Prolog [Theb] CLP(FD) library version 4.1.2.

- **Structural constraint solvers** similarly to our CSP(M) aim to find object graphs satisfying a given set of structural constraints. For our measurements we selected the original KORAT [MMMK] framework based on bounded exhaustive testing.

- Finally, we used the GROOVE 4.0.1 [Ren04a] a model checker for graph transformation system as our third tool due to its very close problem definition language. Note that only the IMA case study was implemented in GROOVE as it does not support flexible constraints.

For all of our measurements, we used an average PC with Mobile Core Duo@1.8 GHz and 3GB RAM running Windows XP and Java SDK 1.6.13. Prior to the actual experiments we expected that:

- The SICStus CLP(FD) library will outperform our approach in all cases by orders of magnitude.

- The KORAT constraint solver will be faster especially on large models where huge traversals are expected.

- Finally, the GROOVE model checker will have a comparable performance with our implementation on small problems and we would outperform it on larger problem sizes due to the exhaustive search algorithm of GROOVE.

8.6.1 The IMA Case Study

We assume that we have to allocate different software workloads (functionalities) on a system with three cabinets (which corresponds to the avionics architecture used in the DIANA project).

8.6.1.1 CSP(M) Solution

Each row in Table 8.1 defines a software workload allocation test case of different Size. The Simple Job, Critical Job, and Partition columns define the actual number of software components to be allocated, where critical jobs are separated based on their redundancy scheme into double (DMR) and triple (TMR) modular redundancy. All Job Instances represents the total number of job instances to be allocated. For our initial measurement (denoted by ATTR) we assume that each job requires the same amount of memory (30 units) and each partition offers the same free memory (300 units).

Runtime results of the four test cases are captured in Table 8.2. Due to the random strategy of our solver we considered an allocation completed if a solution was found within 200 seconds. In each case we executed the solver ten times and present the number of Finished Allocations. Runtime performance and the size of the traversed State Space for the completed allocations are also presented by their minimum (min), maximum (max) and average (avg) values for each test case.

the source code of the case studies is available from http://home.mit.bme.hu/~ahorvath/papers/sosymHVSource.zip
### Lessons Learned

During the analysis and profiling of our implementation we have discovered that the performance bottleneck in our system is mainly related to the model management component of the underlying ViATRA2 transformation framework (which is obviously not optimized for constraint solving purposes). In almost all cases we have observed that core attribute manipulation functions (e.g., `setValue`) are the most time consuming. This is due to the low-level notification mechanism that keeps the incremental pattern matcher up-to-date after changes in the model space, which is more effective for graph manipulations than for attribute changes.

Therefore we also evaluated our approach without attribute manipulation (i.e., memory requirements) on the IMA case study denoted by `NON ATTR`. In order to solve a conceptually similar problem we defined an additional global constraint stating that a partition cannot host more than ten job instances. Results show that (i) in both cases solutions were found traversing only a small number of states compared to the size of the problem, (ii) the `NON ATTR` implementation scales almost up to twice the size in the number of job instances to allocate and (iii) due to the heuristic character of the state space traversal the runtime performances can vary up to two orders of magnitude.

### 8.6.1.2 Other Approaches

We implemented the IMA case study on additional three different tools. In all three cases the maximum number of modules were explicitly given and any solutions within this given range were accepted.

#### SICStus Prolog CLP(FD)

The complete IMA problem was translated into a CLP(FD) problem, where both job instances, jobs, partitions, modules and all mappings between them were mapped to CLP variables. It is important to note that we optimized the labeling strategy to effectively search for

---

**Table 8.1: IMA Test Cases**

<table>
<thead>
<tr>
<th>Size</th>
<th>Simple job #</th>
<th>Critical Job DMR</th>
<th>Critical Job TMR</th>
<th>Partition #</th>
<th>All Job instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>19</td>
</tr>
<tr>
<td>Medium</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>Large</td>
<td>16</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>XLarge</td>
<td>20</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>51</td>
</tr>
</tbody>
</table>

**Table 8.2: Runtime Characteristic of the CSP(M) solution on the IMA Allocation Problem**

<table>
<thead>
<tr>
<th>Size</th>
<th>ATTR.</th>
<th>NON ATTR.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Finished Allocations (out of 10)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Small</td>
<td>Medium</td>
</tr>
<tr>
<td>min</td>
<td>1.1</td>
<td>4.2</td>
</tr>
<tr>
<td>max</td>
<td>196.4</td>
<td>145.8</td>
</tr>
<tr>
<td>avg</td>
<td>66.9</td>
<td>81.6</td>
</tr>
<tr>
<td>Runtime [sec]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>64</td>
<td>146</td>
</tr>
<tr>
<td>max</td>
<td>13 984</td>
<td>12 632</td>
</tr>
<tr>
<td>avg</td>
<td>4 802</td>
<td>8 296</td>
</tr>
</tbody>
</table>
the first solution rather than do a breadth-first like traversal to find all solutions. As a personal experience, the implementation of the IMA case study in SICStus CLP(FD) required far more man-hour (approximately, 30 with optimization and debugging) than the other three solutions. At the end the whole implementation consisted of 31 Prolog clauses in 150 lines of code.

KORAT It required three inputs for instance generation: (i) a Java class hierarchy of the problem domain that we derived directly from the IMA metamodel (see in Sec. 8.1) with minor modification as inheritance is not supported by the framework, (ii) a *finitization* statement that explicitly specifies bounds on the number of objects to be used for the instance construction and finally, (iii) an imperative predicate that specifies the desired structural constraints of the IMA case study, written as a Java method consisting of approximately 100 lines of code.

GROOVE Due to the similar graph transformation based specification language of GROOVE, we simply adopted the graph patterns and GT rules of the NON-ATTR version of the IMA case study. Additionally, the initial models of the test cases were also easily reused. Note that we used only the basic constructs of the GROOVE language and did not apply advanced features like nested graph transformation rules.

8.6.1.3 Evaluation of the Results

The results are shown in Fig. 8.12 with average execution times in a logarithmically scaled *Runtime* axis for all four test cases (see in Table 8.1). Test cases are identified by their size. All test cases were executed ten times. We also applied a 200000 milliseconds (200 seconds) upper limit on the execution times. Results exceeding this upper limit are not shown.

Within the 200 seconds limit both the KORAT and the GROOVE framework failed to provide a solution even for the smallest test case. In case of the GROOVE engine it is acceptable, as it had to generate the complete state space of the problem to check if there is a solution state that satisfies all given constraints. However, also KORAT failed to provide a solution and it was parametrized to stop after the first valid solution. During the analysis of KORAT, we have discovered that it always tried...
8.6. EVALUATION

Table 8.3: Service Allocation Test Cases

<table>
<thead>
<tr>
<th>Size</th>
<th>Physical Server</th>
<th>Virtual Server</th>
<th>DB_P</th>
<th>DB_V</th>
<th>DB_C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>5</td>
<td>10</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Medium</td>
<td>9</td>
<td>9</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Large</td>
<td>20</td>
<td>32</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

The results of this case study show that (i) our approach outperforms the GROOVE model checker that uses an exhaustive state space exploration, (ii) it finds a single solution significantly faster than the well-known KORAT algorithm based on bounded exhaustive testing and (iii) our current implementation is lagging behind classical CLP(FD) libraries with orders of magnitude.

8.6.2 The Service Allocation Case Study

We assume that we have to allocate a predefined number of different databases to an infrastructure consisting of virtual and physical servers and reach a predefined overall performance indicator value for the whole system.

Note that as a distinguished feature our approach also returned the sequence of applied rules that were executed in order to achieve the solution model from the initial model. In case the labeling rules represent direct operations that can be executed on the infrastructure the returned sequence can be used to define the actual reallocation steps.

8.6.2.1 CSP(M) results

Each row in Table 8.3 defines a separate service allocation allocation test case of predefined Size and different performance indicator to achieve. The number of servers in the cloud are defined by the Physical Server and Virtual Server columns. Similarly, the number of database licences are captured by the DB_P, DB_V and DB_C columns, respectively.

In each test case we did four different measurements (see in Table 8.6.2.1. First, we evaluated the flexible CSP(M) with the defined resources and required overall performance (see in Table 8.13(a)). Based on this flexible constraint satisfaction problem we assessed three different dynamic changes of the original problem. We evaluated the cases described in Section 8.4.3.1 where

- SubGoal removal: The subgoal DB_ConVServer was removed from the problem.
- Labeling rule removal: The labeling rule shutDownVServer was removed from the definition.
- Labeling rule addition: Finally, the labeling rule createDB_C_ClusterTriplet (depicted in Figure 8.10) was added. In the latter two cases we also modified the required overall performance indicator to balance out their effects.

In all three dynamic modifications we followed the considerations discussed in Section 8.4.2:
In case of the subgoal removal it means that all already traversed states were updated, but that only required a recalculation of the weight function on each state.

The labeling rule removal required the pruning of the already visited state space after any transaction that applied the shutDownVServer rule.

Finally, for the labeling rule addition we followed the strategy to continue the solving process after the modification without re-evaluating any already visited states. This was mainly used as our transaction mechanism does not effectively support jumping between states belonging to different branches.

All three dynamic changes were made in a solution state from where the evaluation of the modified constraint problem started. Their performance results using our CSP(M) framework are captured in Figure 8.13(b), 8.13(c) and 8.13(d), respectively.

For the flexible CSP(M) we measured the overall Runtime of the solving process and the number of traversed states. As for the three different dynamic modifications we assessed the number of newly traversed states (Traversed states) to solve the dynamically changed problem. Additionally, we measured the overall Runtime required for both the reevaluation of the state space and the new solving process. In all four measurements, we executed the solver five times and present the number of Finished Allocations using again a 200 seconds upper limit on execution time.

Lessons Learned As a summary, our solver is capable of handling reasonable sized flexible CSP(M)s. However, during the analysis of the traversed state space we have discovered that our search strategies do not always effectively guide traversals of flexible constraint satisfaction problems. Their main drawback is that they do not take into account the weight function when selecting the labeling rules to apply. We believe that effective guidance of flexible CSP(M) should adapt informed search strategies like A* [HNR68] with the estimated cost function directly derived from the weight function as it holds all relevant guidance information.

Similarly, the lack of guidance can be observed in case of the dynamic modifications. After the re-evaluation of the already visited states the traversals acted similarly as an exhaustive search, resulting in runtime performances that vary up to several orders of magnitude. For example, on one hand the addition of a labeling rule resulted in very fast traversals for the new solution of the modified problem. On the other hand, removal of the DB_ConVServer constraint from the problem definition resulted in a state space exploration that exceeded our 200 seconds upper limit. These differences were due to the fact that our engine preferred the use of clusters and the allocation of databases to virtual server rather than physical ones. In case of the addition of the createDB_C_ClusterTriplet labeling rule, it was able to easily produce the required cluster triplets from the already allocated cluster pairs. Moreover, the retraction of the shutDownVServer did not have any effect when a solution mainly allocated to virtual servers and extensively created clusters (like in case of our small and medium sized test cases). However, when solutions could only be found, which heavily relied on allocation to physical server, our approach had to traversed large state spaces.

8.6.2.2 Other Approaches

We implemented the service allocation case study using both SICStus and KORAT. As these approaches do not support dynamic manipulation of constraints, we separately evaluated the modified constraint problems starting from the original initial state.
8.6. EVALUATION

(a) Basic Flexible Problems

<table>
<thead>
<tr>
<th>Size/performance indicator</th>
<th>Small/65</th>
<th>Medium/100</th>
<th>Large/180</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finished Allocations</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>(out of 5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Runtime [sec]</td>
<td>min</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td></td>
<td>0.6</td>
<td>81.4</td>
<td>47.5</td>
</tr>
<tr>
<td></td>
<td>13.8</td>
<td>69.1</td>
<td>32.4</td>
</tr>
<tr>
<td></td>
<td>9.1</td>
<td>83.5</td>
<td>52.2</td>
</tr>
<tr>
<td>Traversed States #</td>
<td>min</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td></td>
<td>494</td>
<td>31 893</td>
<td>17 087</td>
</tr>
<tr>
<td></td>
<td>5812</td>
<td>24 243</td>
<td>13 325</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>5666</td>
<td>7 911</td>
</tr>
</tbody>
</table>

(b) DB_ConVServer SubGoal Removed

<table>
<thead>
<tr>
<th>Size/performance indicator</th>
<th>Small/65</th>
<th>Medium/100</th>
<th>Large/180</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finished Reallocations</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>(out of 5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Runtime [sec]</td>
<td>min</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td></td>
<td>186.7</td>
<td>198.8</td>
<td>192.5</td>
</tr>
<tr>
<td></td>
<td>0.8</td>
<td>197.8</td>
<td>81.7</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>134.5</td>
<td>41.2</td>
</tr>
<tr>
<td>Traversed States #</td>
<td>min</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td></td>
<td>40 343</td>
<td>43 253</td>
<td>41 798</td>
</tr>
<tr>
<td></td>
<td>110</td>
<td>44 711</td>
<td>23 655</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>12 478</td>
<td>3 860</td>
</tr>
</tbody>
</table>

(c) shutDownVServer Labeling Rule Removed

<table>
<thead>
<tr>
<th>Size/performance indicator</th>
<th>Small/65</th>
<th>Medium/85</th>
<th>Large/170</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finished Allocations</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>(out of 5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Runtime [sec]</td>
<td>min</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>2.4</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>4.5</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td>5.2</td>
<td>179.3</td>
<td>63.9</td>
</tr>
<tr>
<td>Traversed States #</td>
<td>min</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>45</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>451</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>889</td>
<td>8525</td>
<td>3 860</td>
</tr>
</tbody>
</table>

(d) createDB_C_ClusterTriplet Labeling Rule Added

<table>
<thead>
<tr>
<th>Size/performance indicator</th>
<th>Small/70</th>
<th>Medium/105</th>
<th>Large/200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finished Reallocations</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>(out of 5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Runtime [sec]</td>
<td>min</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td></td>
<td>0.46</td>
<td>0.67</td>
<td>0.64</td>
</tr>
<tr>
<td></td>
<td>0.12</td>
<td>0.42</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td>0.15</td>
<td>0.9</td>
<td>0.58</td>
</tr>
<tr>
<td>Traversed States #</td>
<td>min</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>24</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>31</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>234</td>
<td>101</td>
</tr>
</tbody>
</table>

Figure 8.13: Runtime Characteristic of the CSP(M) solution of the Service Allocation Case Study

SICStus Prolog CLP(FD)  Similarly to the IMA approach we translated the servers to CSP variables and modelled the available databases with the integer domain of these variables. Mapping of virtual servers to physical ones were implemented as a set of constraint over their CSP variables. Additionally, auxiliary CSP variables were used for the definition of clusters and for the evaluation of the weight function. The implementation consists of 28 Prolog clauses in approximately 170 lines of code. For the modified constraint problems, only small modifications were required on few clauses of the original code. Again this implementation took considerable more time than any other.

KORAT  KORAT cannot define constraints for a dedicated instance of a class (only for the class itself, to our best knowledge). We had to modify the problem definition that all servers can host the same amount of virtual servers. As a consequence, the case study where the shutDownVServer labeling rule were removed could not be effectively defined in the imperative predicate and therefore we omitted it from from the measurements. Similarly, the Java classes were derived directly from the Service Allocation metamodel (see in Sec. 8.3) and the imperative predicate were also given as a Java method consisting of approximately 140 lines of code.

8.6.2.3 Evaluation of the Results

The results are shown in Fig. 8.14 with separate figures for the basic flexible problem and its three modified version. Average execution times in milliseconds are presented in a logarithmically scaled
CHAPTER 8. DESIGN SPACE EXPLORATION

Figure 8.14: Runtime Results of all Approaches on the Service Allocation Case Study

Runtime axis. Each measurement was executed five times. We again applied a 200000 milliseconds (200 seconds) upper limit on the execution times and results exceeding this upper limit are not shown in Fig. 8.14.

Again, within the 200 seconds limit the KORAT framework failed to provide a solution even for the smallest test case for the original problem or its two modified versions. The main reason is that KORAT always preferred to allocate databases to physical servers rather than to virtual ones. This resulted in extremely large search spaces.

The SICStus implementations again produced very consistent execution times and in certain cases orders of magnitude faster than any other approach. However, for the Small sized test case in average our engine produced solutions within a comparable range. This was due to the fact that in our implementation the labeling algorithm of the SICStus engine always tried to allocate databases to physical servers and only slowly found solutions were both clusters and virtual servers were required. This is one of the main differences between the two solutions and reason for the diverse runtime performances.

Altogether, these measurements demonstrated that in four out of nine dynamic cases partially reusing the solution obtained from a previous traversal of the original problem is a competitive alternative. Additionally, in case of complex structural constraints, the way the search space is traversed has a significant impact on performance and effective solutions require explicit problem specific fine-tuning or hints to achieve acceptable performance.
8.6.3 Summary

Our measurements show that our constraint solver based upon incremental pattern matching is able to solve non-trivial classical and flexible problems of model oriented constraints. We also demonstrated that certain dynamic changes of constraint definitions can be effectively handled with a good level of solution reuse. More specifically:

- Constraint satisfaction problems with complex structural constraints (e.g., definition of a circular constraint in graph structures) can be intuitively captured by our proposed formalism combining graph patterns and graph transformation rules. In contrast, expressing structural constraints in the traditional CLP(FD) formalism requires significant modeling workaround or may result in extensively large constraint set.

- Our approach outperformed in all cases the well-known academic KORAT structural constraint solver. We believe that this is a combined effect of using (i) incremental pattern matching to efficiently detect possible continuations and (ii) explicit labeling rules to guide the traversal.

- Unsurprisingly, exhaustive generation of the state space (like in case of GROOVE) is not a feasible solution for constraint satisfaction problems without further support. Ongoing research in the GROOVE framework aims to restrict state space travels by adding a conjunction of global constraints.

- As we expected, the industrial SICStus CLP(FD) library outperformed our engine in the static cases by orders of magnitude. However, in case of dynamic constraint satisfaction problems our approach resulted in comparable (in certain cases even better) runtime thanks to good level of solution reuse.

However, as two distinguishing features from CLP(FD) our approach is (i) capable of defining problems over an infinite search space and (ii) able to provide the applied labeling operations to achieve the solution from the initial model.

- Additionally, in almost all cases to achieve acceptable performance problem specific hints or fine-tuning is advantageous. However, these fine tuning hints would increase the complexity of the problem definition. We believe that our graph transformation based CSP formalism gives a good trade-off between easy declarative problem definition and fine-tuning.

- Due to the nondeterministic nature of our traversal strategy, execution times may vary significantly. For this reason we plan to better exploit adaptive search algorithms.

It is also important to note that these measurements were carried out on specific problems derived directly from our ongoing research projects. Despite the large set of predefined synthetic case studies used mainly for performance measurements in model transformation tool contests ([AGT07, Tra10]), very few cases (e.g., the live contest at [Gra08]) address related challenges (like backtracking or flexible problems). We plan to submit our case studies to future editions of these tool contests.

Our further investigations have to be directed to (i) combine our constraint definitions with constraints over regular attributes, (ii) develop specific informed search strategies for traversals of flexible CSP(M)s and (iii) further examine the effects of dynamic constraint changes to enhance solution reuse.
8.7 Related Work

Applications of CSP in MDE. Constraint satisfaction techniques have been successfully applied in the context of MDE. [SBP07] proposes an approach for partial model completion based on constraint logic programming. [WSNW07] support efficient domain specific modeling by transforming constraints to a Prolog representation. In [EBM08], poor design patterns are detected by using off-the-shelf CSP techniques and tools. [JKW08] defines an interactive guided derivation algorithm to assist model designers by providing hints about valid editing operations that maintain global correctness of models.

In the context of model transformations, [Rud00] proposes constraint solving as a graph pattern matching strategy. [PBM09] proposes Constraint Relation Transformation an extension of QVT Relations with numerical constraints by integrating local numerical constraint solving (over attributes of model elements).

Recent approaches like [ABGR09, WTEK08a, JS07] aim at automatically creating instance models, which conform to a given metamodel and a set of constraints. This model generation problem is solved by existing back-end tools like Alloy as in [ABGR09], or by a dedicated theorem prover for Horn-like clauses as in [JS07]. This problem can also be interpreted as a special (restricted) CSP problem without numeric constraints on attributes. Additionally, UMLtoCSP [CCR07] verifies certain correctness properties of OCL adorned UML model by translating them into the ECLiPSe CSP solver and executes a bounded instantiation search.

In all these papers, constraint satisfaction techniques are used to assist model-driven engineering. The main innovation of our work is just the opposite: it investigates how model transformation techniques can contribute to solve complex constraint satisfaction problems over complex structural constraints and dynamic labeling rules.

Model-driven design space exploration techniques. The DESERT tool suite [NSKB03] provides model synthesis and constraint-based DSE for DSMLs with structural semantics using ordered binary decision diagrams for encoding and pruning the design space. [SK10] presents a generic DSE framework extending upon DESERT by supporting arbitrary analysis tools and includes model transformations for mapping design problems to intermediate and low-level formats. The OCTOPUS Toolset [BvB+10] uses an intermediate representation for design problem specification and performs DSE using integrated analysis tools. These are all compiled approaches, where the design problems are specified as models and model transformations are applied to derive inputs for analysis tools. Furthermore, the analysis tools perform the DSE, while in our approach, they only provide hints for the exploration. Schätz et al. [SHL10] developed an interactive, incremental process using declarative transformation rules for driving the exploration. The rules are modified interactively to improve DSE performance, which can be considered as a guidance. However, the hints do not originate from analysis, contrary to our approach.

Structural constraint solving allows finding object graphs that satisfy given constraints both on attributes and (object) structures for systematic testing by exploring a (usually) bounded number of possible object graphs. Many promising approaches exist like the CUTE [SMA05] framework that uses a combination of symbolic and concrete execution to derive path constraints for each separate execution paths, the Java PathFinder [VPK04] that is based on Generalized Symbolic Execution [KPV03] that first introduced the idea to use model checkers for solving structural constraints, Alloy [Jac02] a lightweight object modelling framework using a simplified Z notation that is translated to boolean formulas for SAT based evaluation or KORAT [BKM02] that performs specification based testing by using a predicate representing the properties (constraints) of the desired output structures and explores the input state space of the predicate using bounded exhaustive testing.
8.7. RELATED WORK

It is common in these approaches that each solution satisfies all given constraints similar to our approach; however, their main difference is that they cannot define restrictions on how these solutions are achieved from the initial state, meaning that no constraints can be defined to hold on states visited during a solution trajectory, which in our case is supported by the global constraints.

**State Space Exploration for GT.** There are several state space exploration approaches to analyze graph transformation systems (GTS).

Augur2 [KK06] is a GTS model checker that tackles the complexity associated with independent rules by condensing the entire state space into a single graph with unfolding semantics. It also provides some approximative techniques to deal with infinitely large state spaces, and counterexample-guided refinement of this abstraction.

GROOVE [Ren04a] is a model checker over graph transformation systems. Its main benefit is the ability to verify model transformation and dynamic semantics through applying CTL model checking on the generated state space of the GTS. It is mainly used for modeling and verifying the design-time, compile-time, and run-time structure of object-oriented systems.

It is common in these solutions that they store system states as graphs and directly apply transformation rules to explore the state space similar to our approach. Their main difference is that they use an exhaustive state space exploration to verify certain conditions in the graph transformation system, while our approach relies on guided traversals.

**Graph constraints** were first introduced in the context of negative application condition and later extended as a specification formalism [OEP08, Ore08] to define constraints associated to visual modelling formalisms and reason about them with a set of sound and complete inference rules. Our graph pattern based constraint specification is based on these foundations, however, we use a different pattern language that allows recursive pattern compositions but more restrictive on formulas and does not support all connectives e.g., implication. However, we believe that CSP(M) can also be instantiated over this graph constraint formalism.

Graph transformation rules are also used in [Baa05] to define a non-restrictive contract specification language by the means of pre- and postconditions. It combines GT rules with OCL in order to be able to capture non-deterministic specifications and overcome the frame-problem [BMR95b]. Its language is far more expressive than our, however due to this expressiveness no implementation is available to evaluate its performance as in our case.

**Constraint based graphic systems** define complex (graph based) drawings and diagrams using constraints on their graphical objects and relationships.

ThingLab [Bor81] is an extensible constraint solver for graphical simulation. In ThingLab, constraints are imperatively defined providing functions to solve individual constraints, and the solver attempts to invoke them in an appropriate order for solving the complete constraint store. It also supports definition of constraints in an object oriented manner, allowing inheritance of constraints along the supertype relationship.

DeltaBlue [SMFBB93] is a perturbation based constraint hierarchy solver, maintaining solutions incrementally as constraints are dynamically added or removed. Additionally, it minimizes the cost of finding a new solution after each change by exploiting its knowledge of the last solution.

Juno-2 [HN94] is a constraint-based double-view drawing editor for the definition of interactive graphics. It uses an extensible declarative constraint language including non-linear functions and ordered pairs compiled into effective constraint primitives for interactive feedback.

On one hand, common in these approaches that they support only a limited set of constraints and (except Juno-2) cannot define cyclic constraints (e.g., simultaneous equations and inequalities on the variables). On the other hand, many techniques applied in these approaches for handling large number of constraints such as (i) packing and unpacking constraints into constraint primitives and
(ii) propagation of values through predefined constraint hierarchies can be partially adopted to our framework giving space for future research.

CSP-specific Research in the field of constraint satisfaction programming has been conducted towards flexible and dynamic constraints [Sch94, MS00]. Our approach shows similarities with both approaches as (i) it also allows to add (or remove) additional constraints during the solution process as defined in the dynamic extension, and (ii) can give support for cost based optimization defined over the constraint (flexible) even in the case of complex structural constraints.

Additionally, our state space exploration approach also builds on the idea of random traversals described in [BMS00] to solve large problems.

8.8 Summary

In order to address design space exploration in problem domains containing complex structural constraints, dynamic problem definition changes (models@Runtime) or based on graph models, we presented a novel approach for defining constraint satisfaction problems directly over models using graph transformation rules and graph patterns. Compared to traditional CSP,

- We extended the labeling process by using model manipulation as provided by graph transformation to dynamically create and delete model elements.

- We introduced dynamic constraint satisfaction programming over models that allows to dynamically add or remove global constraints, subgoals and labeling rules to alter the problem definition.

- We have presented a weighted extension to classical CSP(M) that supports flexible constraint satisfaction problems based on relaxable soft constraints.

- Additionally, as a distinguishing feature our approach is capable of providing the sequence of labeling rules applied to reach the the solution from the initial model.

We have also built a prototype solver implementation on top of the ViATRA2 model transformation framework using incremental pattern matching that provides an efficient constraint propagation technique to immediately detect constraint violation. Moreover, the solver integrates various strategies (e.g. random backjumping, directed search) to guide the state space traversal.

In addition, we carried out various comparative measurements to assess the performance of our approach, which demonstrated that our solver based upon incremental pattern matching is able to solve non-trivial classical, flexible and dynamic problems for structural constraints.

As a summary, we argue that model transformation technology can efficiently contribute to formulate and solve certain constraint satisfaction problems with complex structural constraints and dynamic labeling rules. It can provide a natural way for handling problems in the models@Runtime domain. For example, we have already applied the approach for generating quick fixes for business process models [11]) in a graphical editor.

The result of this chapter are formulated as thesis contributions as follows:

1. **Structural CSP problems.** I elaborated a novel way to define static [29, 15], dynamic and flexible [1] CSP problems with complex structural constraints over graph based models (see in Sections 8.3.2 and 8.4).
2. **Structural constraint language based on graph transformation.** I defined a structural constraint language for the proposed CSP problems [1, 15], where constraints are defined by graph patterns and domain specific manipulation operations are specified as graph transformation rules (presented in Sections 8.3.2.2 and 8.3.2.1).

3. **Efficient solver for the structural constraint problems defined by graph transformation rules.** I developed efficient algorithms [15] based upon incremental graph pattern matching for solving static, dynamic and flexible constraint satisfaction problems over models (see in Section 8.3.3).

4. **Heuristic based traversal optimization.** I elaborated a guided traversal algorithm [1] using efficient heuristics based upon a Petri-net based abstraction [VVGE+06] to minimize the traversed state space (presented in Section 8.5.2).

Additionally, by experimental evaluation and comparison with other structural constraint solvers, I proved the feasibility of the developed structural constraint solver implemented in the Viatra2 framework [1].

The implementation of the constraint satisfaction programming framework was carried out within the DIANA research project [DIA] and it is integrated into the Viatra2 [Via] framework. It is built upon the incremental graph pattern matcher of the Viatra2 framework, which is part of the PhD work of Gábor Bergmann.

Additionally, this work is continued in the PhD work of Ábel Hegedüs who further advanced the capabilities of the CSP(M) framework by using rule-dependency and occurrence vector based guiding strategies [12,3].

Finally, the Petri net based abstraction technique is a work of Szilvia Varró-Gyapay and Dániel Varró. My contribution lies in its adaptation as a guidance strategy for design space exploration.
Part IV

Model-Driven Engineering for Avionics Systems
Introduction to Aeronautics

The aim of this chapter is to provide an overview on current civil aeronautical context with a focus on: avionics platforms for civil aircrafts in Section 9.1, regulatory bodies and organizations in Section 9.2 and finally, currently used and future certification guidelines and standards in Section 9.3.

The chapter follows the introduction to certification of civil aircrafts as described in [DIA07a].

9.1 Avionics Architectures: from Federated to Integrated Modular Avionics

The term avionics\textsuperscript{1} - aviation electronics - appeared and was popularized when electronic devices were introduced and generalized, mainly for the military aviation in the early '70.

Electronics provided an opportunity to reduce costs by sharing a common definition and development of certain parts. It also facilitated - limited - exchanges of information between avionics devices. Introduction of software provided more flexibility than pure analogue electronics, inside devices and in interconnection of those devices. Even with this, each piece of avionics equipment was specific: specific processor, memory, programming language, etc, resulting in today’s most widely used approach: federated avionics architecture.

In these systems resource sharing occurs only at the last link in the information chain, via the controls and displays. Several standard data processors are often used to perform a variety of low-bandwidth functions such as navigation, file management and flight control. The data processors are interconnected by time-division multiplex buses which provide low data rate capability (e.g., STANAG 3910 and MIL STD 1553). Low interconnection bandwidths and central control is possible because high speed signalling requirements such as A/D conversion and signal processing occurs within vendor specific black boxes through interconnections within dedicated backplanes in each

\textsuperscript{1}popularized by Philip J. Klass
of the federated communication chains. The architecture is made of a number of loosely coupled equipments that are connected through dedicated buses.

One of the main disadvantages of federated systems is that each component has a specific function, with specifically developed hardware and software. Usually, each system is developed from scratch, with the lack of technology and certification re-use. Another disadvantage is the increased weight and power consumption as each unit carries its own dedicated environmental protection measures and power management system. Moreover, as data is not shared among the different federated systems it requires in additional dedicated communications that leads to an increase in overall weight resulting in higher fuel consumption.

Example 22 A sample federated avionics system is depicted in Figure 9.1 based on [WW07]. It consists of a user interface defined by an air conditioning processing unit, display and control. This user interface is used to control the actuators based upon feedback collected from a sensor. These are developed as three separate units connected by dedicated communication channels (three CPUs, five I/O modules and Network interfaces and four physical communication channel).

However, as the complexity of airborne systems has exponentially grown in the last decades [Kni02] and avionics took on an ever larger part of the total aircraft development time and financial budget. It resulted in the need for a simpler general architecture that (i) allows the sharing of certain computational and communication resources for tighter integration and less weight and power consumption and (ii) supports incremental certification, where a change in any component does not require the recertification of the complete system. Thus a new kind of architecture called Integrated Modular Avionics (IMA) was developed [RTCd]. It is built around the concept of common resource elements that are shared between the different applications. Therefore, Integrated Modular Avionics can be considered as a step forward for high-level (or physical) resource sharing and succeeds to federated avionics that did not reach this level of sharing between both data and resources.

9.1.1 Integrated Modular Avionics

The Integrated Modular Avionics (IMA) concept is to have an architecture integrating many services on the same platform and decoupling the applications from the hardware. With the result of (i)
enabling a cost efficient handling of hardware obsolescence and (ii) promising significant weight reduction and maintenance savings.

Generally there are two types of IMA architectures: Open and Closed. An open IMA architecture utilizes interfaces that are non-proprietary, and adhere to interface definitions available in the public domain (e.g., ARINC 653 [ARib]). Closed IMA architectures utilize proprietary interfaces that are custom implementations and are optimised for the specific applications [WW07]. In the current thesis we focus only on the open IMA architecture concept as a publicly available technology.

The core element in an IMA architecture is the platform. A platform in itself is not performing any avionics function, but provides all necessary functions for avionics applications like communication, computing and resource management. The platform has a generic processor that runs a real-time operating system (RTOS). The RTOS hosts the avionics specific applications through the standardized ARINC 653 APplication EXecutive (APEX) API. All applications are fully isolated by strict partitioning mechanisms and error containment [Rus00] that enables safe sharing of the processing resource, the memory and all communication means.

On the communication side the platforms are common digital modules with standard input/output interfaces, where data communication takes place via bus based networks like Avionics Full Duplex Switched Ethernet (AFDX). All the data from sensors and other equipments are translated from/to the standard data network. The network is configured to route the information anywhere within the architecture, which eases the system integration, and allows to mix components from different suppliers.

Additionally, this also allows the application developers to focus on the application layer, thus reducing the risk of hardware integration issues. As applications often share a major part of their underlying hardware and lower-level software architecture (e.g., drivers), maintenance of the platform is easier than with previous specific architectures for each application. Finally, as a unique feature applications can be reconfigured on spare resources if the resource that supports them is detected faulty during operations, increasing the overall robustness of the avionics functions.

To sum up, the main advantages of the IMA architecture are:

- **Common processing subsystems** allow multiple applications to share and re-use the same computing resources. This facilitates a reduction in the number of deployed subsystems which are not fully utilized and provides a more efficient use of system resources, leaving space for future systems and extensions.

- **Software abstraction** isolates the application not only from the underlying bus architecture but also from the underlying hardware architecture. This enhances portability of applications between different platforms and also enables the introduction of new hardware to replace obsolete architectures.

- IMA architecture reduces the cost of change, since it lowers re-certification costs by strict partitioning of applications and platform components for simplified impact analysis and thus facilitates reuse of application.

**Example 23** Figure 9.2 shows the implementation of the air conditioning unit using an IMA architecture, which has an optimized set of shared computing resources. As its main advantage it uses less physical resources (only one CPU and one communication bus) when compared to the federated system depicted in Figure 9.1 hosting the same functions.
ARINC 653 [ARIb] is a key technology in the development of applications for Integrated Modular Avionics. In many ways it represents a conceptual shift in avionics development as it recognizes the real-time operating system as key component of an IMA system. It specifies the interface boundary between avionics software applications and the core executive software that is tightly integrated with the underlying platform. The current section follows the introduction of the ARINC 653 standard as described in [Pri08].

The aerospace industry developed ARINC 653 as a standardized RTOS interface definition with the following three main specific needs for avionics applications:

- **Real-time** - Reactions and responses of the system must be within prescribed time period, where missing of a deadline can lead to catastrophic failures.
- **Safety-critical** - Must comply with operative regulation rules (e.g., DO-178B) for the highest safety criticality levels
- **Deterministic** - Results provided by the system are predictable and repeatable.

Up to now, ARINC 653 is the only RTOS interface definition that supports these needs.

**Overview of the ARINC 653 Architecture**  
ARINC 653 defines support for robust partitioning in avionics systems, meaning that one processing unit is able to (i) host one or more avionics applications and (ii) execute them completely independently. An overview of the core ARINC 653 Architecture is depicted in Figure 9.3. The main components of the system are:

- The **Hardware Board** that provides the computational resources and physical communication means for the hosted avionics application. For example like the Motorola MVME5100 or the Wind River SBC8641D.

- The **Board Support Package** that contains all necessary drivers and kernel components for the integration of the Module Operating System over the Hardware Board. This is usually provided by the Module operating system provider.
9.1. INTEGRATED MODULAR AVIONICS

The Module (Real-Time) Operating System (MOS) that implements and provides the ARINC 653 APEX services and API as defined by the standard (e.g., the Wind River VxWorks 653 or the SYSGO PikeOS). The MOS is responsible for allocating processor time and memory regions to each avionics application and provide fault containment, such that a failure in one application cannot cause a failure in another application [Rus00]. The configuration of the MOS is handled by XML Configuration Tables that represents primary software elements both in the development and certification process.

Partitions are hosted by the Module OS. A partition is similar to that of a multitasking application within a general purpose computer. Partitioning separates applications in two dimensions: space and time [Rus00]. Spatial separation means that the memory of a partition is protected. No application can access memory out of the scope of its own partition. Temporal separation means that only one partition at a time has access to system resources (including the processor). Therefore only one application is executing at one point in time resulting that there is no competition for system resources between partitioned applications.

ARINC 653 uses a static configuration (all defined in the XML Configuration Tables) where each partition is assigned a set of execution windows. An application in the partition associated with the current execution window gains access to the hardware board. When the execution window terminates, the program is preempted, and continues its execution from the point it was previously preempted when its next execution window starts. An application within a partition does not in any way know if it has been preempted by the MOS.

All user defined applications are running in normal Application Partitions, while certain services (e.g., Health Monitor) of the Module OS are running in dedicated System Partitions. These system partitions also have dedicated execution windows and are part of the overall static con-
figuration. However, some severe errors can override the predefined static configuration and the OS allows immediate execution of predefined countermeasure operations with access to the complete available hardware resources.

- Finally, Processes within the scope of a partition are scheduled by a priority-based preemptive scheduler with first-in-first-out (FIFO) order for processes with the same priority. This second level scheduler is invoked whenever an execution window assigned to its partition starts and the partition gains access to the hardware resources. The process scheduler is preempted by the first level partition scheduler when the execution window terminates.

To enable application portability, communication between partitions (inter-partition communication) is independent of the location of both the source and destination partition. It is always handled by messages. An application sending a message to, or receiving a message from, another application will not contain explicit information regarding the location of its own host partition or that of its communications partner. The information required to enable a message to be correctly routed from source to destination is contained in the XML configuration tables that are usually developed in collaboration with the individual application developers and maintained by the system integrator. The system integrator configures the environment to ensure the correct routing of messages between partitions hosted on an IMA platform.

As for intra-partition communications the ARINC 653 standard defines well known structures like buffers, blackboards, events and semaphores that work similarly as in other multi-process systems (e.g., POSIX).

**Health Monitor** One unique feature of the ARINC 653 standard is the definition of Health Monitor (HM) functions. The Health Monitor resides with the MOS and interfaces to a recovery strategy table (all defined in the XML configuration tables) defined by the system integrator. The Health Monitor is responsible for monitoring both all hardware faults within the IMA system and the faults within the MOS. Usually, hardware faults are detected by the Built-In Test (BIT) and any hardware reconfiguration is hardware implementation specific and thus driven by the board support package. However, this is hidden from all other parts of the IMA system.

Faults can be detected at various levels. The main objective is to contain faults before they propagate across their interface boundary. If an application detects a fault in its operation, it is able to report this to the MOS, which then invokes the Health Monitoring function. A recovery table of faults is used to specify the action to be taken in response to the particular fault. These operations include (i) reporting the fault, (ii) restarting or terminating the faulty process or (iii) terminating the complete partition and starting an alternative one. Usually these actions are largely depend on the actual IMA system and the level of safety required for the concrete application.

Within ARINC 653 three types of recovery tables are present:

- The optional *process level* recovery table defines a kind of fault handler process that runs along the other processes within partition. Faults handled at this level are usually specific to the concrete application and may occur in normal execution.

- *Partition level* recovery tables are executed within a system partition and scheduled within its own time window, however its memory and context belongs to the Module OS. They define actions for errors that cannot be handled by a single process and may require partition level interaction (e.g., restart, termination).
9.2. CERTIFICATION OF AIRWORTHINESS: REGULATORY BODIES AND ORGANIZATIONS

- **System recovery** tables (sometimes called Module level) defines actions to be taken in the present of severe failures that could result in catastrophic events (e.g., power failure, system crash, etc.). Actions taken on this level has higher priority than any other partition and immediately scheduled for arbitrary long time window.

There is a strict hierarchy between the tables. In the event of a failure, first, the process level (if present) recovery table is checked if it contains any actions defined to be taken in the present of that failure. If there is such an action than the failure is handled at that level. In case there is no definition than the failure is propagated to the partition level and finally if it cannot be handled on the partition level it is propagated up to the system level.

**Summary**  ARINC 653 is a key technology in the development of safety critical applications over Integrated Modular Avionics architecture. As its main advantage it recognizes the operating system as a key element in the architecture and abstracts away the underlying hardware by defining a standard interface and execution semantics for the complete real time operating system.

9.2 Certification of Airworthiness: Regulatory Bodies and Organizations

The purpose of this section is to give an overview of Airworthiness Certification of aeronautical products in civil aviation. It gives a general view of current regulatory bodies and organisations involved in the aeronautical environment. Additionally, it also introduces some of the specific regulations that an aeronautical product has to comply with, providing a brief overview of certification standards used in the development of products.

9.2.1 Regulatory Bodies and Organizations for Civil Aeronautical Products

The first widely accepted international agreement for the aeronautical word was accepted in 1944 at Chicago and gave birth to the International Civil Aviation Organization (ICAO) [ICA]. ICAO currently works as a specialized agency of the United Nations with the following purpose:

- Ensuring the orderly and safe development of civil aviation all over the world by the adaptation and codification of standards and international procedures for airworthiness.
- Planning and development of international air transport to ensure its safe and orderly growth.
- Defining the protocols and procedures for air accident investigation followed by transport safety authorities of member states

However, it is not the ICAO directly that performs the various certification activities for different avionics equipments and aircrafts. ICAO only defines a set of minimum rules to which all member countries are mandated to comply. Based on these rules each country is than able to develop its own regulations compliant to ICAO. Some of the largest regulation bodies are the following: European Aviation Safety Agency (EASA) in Europe [EAS], Federal Aviation Administration (FAA) in the USA [FAA], Transport Canada Civil Aviation (TCCA) in Canada [TCC], etc.

Focusing on details for Europe the EASA was established in 2006 as a EU Agency and it is tied directly to the European Commission. The EASA has direct authority over aircraft manufacturers, service and equipment providers, repair stations and operators all over the European Union. Its aim
is to have a consistent and uniform set of rules on the safety and interoperability of the European aviation system, with a single organization responsible for preparing, implementing and monitoring them. However, similarly to ICAO the EASA mainly works as a monitoring and codification organization and concrete implementation of the European regulation rules and individual airworthiness certification are issued by the National Aviation Authorities.

Certification Rules and Regulations  The certification in civil aviation can be described along with three aspects:

- It is the formal recognition and Legal statement (as a written certificate issued by the national authority) that an aeronautical product complies with the applicable regulations.

- It is the procedure, i.e. the certification process by which it is given the documented assurance that a product, process or service conforms to specified requirements.

- The output of such a process helps achieve acceptable documented answers to the three questions: Does the system meet regulations? Is the system fit for flight? Is the system safe for flight?

Certification requirements derive from legal duties and associated rules and regulations. The Regulations include the technical codes on the basis of which are conducted the certification processes of aircraft (e.g., airplanes: FAR 25/EASA CS-25, helicopters: FAR 27/CS-27 and FAR 29/CS-29, etc.). Usually, these regulatory materials contain very high-level requirements and serves as the root element for the hierarchical refinement of the requirements that in finally the avionics system must fulfil.

Example 24  As an example a high-level, safety requirement for large aeroplanes defined in EASA CS 25.1309 [Eurb] are expressed as follows:

The airplane systems and associated components, considered separately and in relation to other systems, must be designed so that-

1. Any catastrophic failure condition
   
   a) is extremely improbable; and
   
   b) does not result from a single failure; and

2. Any hazardous failure condition is extremely remote; and

3. Any major failure condition is remote.

The aeronautical product has to comply with its technical needs (intended operational function) and with the safety requirements, considering the function supported. Safety is an overall ability designed and built into a system. Safety encompasses properties such as reliability, integrity, availability, continuity, and/or any other characteristics such as performance and/or human factors when they relate to safety (e.g. landing performance is obviously a safety issue, the flight crew response to cockpit warning is a safety issue as well).
Organizations  In compliance with certification and safety regulations there are several organizations that develop standards for authorities, which may adopt those as acceptable means of compliance with their rules and regulations. Usually these standards are either defined by the industry as already used methods for the development of aircrafts and avionics system or adapted field-proven technology from other safety critical (e.g., control of power plants) or military domains.

Standards give means to (i) develop certifiable systems, software and hardware, (ii) conduct activities to produce certification artifacts or (iii) contribute in systems certification and safety processes.

The four most widely recognized organizations are the following:

- EUROCAE - European Organization for Civil Aviation Electronics [EURa] is a European agency that works closely with RTCA to define standards on all levels of aircraft manufacturing, like ED12 / DO-178B (Software), ED-14 / DO-160 (Environmental Conditions), ED-80 / DO-254 (Hardware), ED-124 / DO-297 (IMA), etc.

- RTCA - Radio Technical Commission for Aeronautics [RTCa] is a nonprofit organization funded by the FAA. Its main focus is on the definition of standards for aircraft manufacturing.

- SAE International [SAEa] is a society of engineering professional with more than 120,000 members world wide. It is a general standards development organization for all kind of powered vehicle. In the aeronautical domain they are most notable by their Aerospace Recommended Practices guidelines like the ARP 4754 (Systems), ARP 4761 (Safety), etc.

- ARINC - Aeronautical Radio [ARIa] is a privately held company and a major provider of communications and system engineering solutions for different industries (e.g., avionics, healthcare, networks, security, etc.). Within the avionics industry it has developed several standards from line-replaceable electronics units to complete real-time operating systems.

9.3 Certification of Airworthiness: Processes and Standards in Civil Aeronautics

This section provides an overview of the main standards that are used as internationally recognized means of compliance with regulatory certification requirements. Standards for Systems, Safety, Software and Hardware development and certification, together with Integrated Modular Avionics development and certification. As the current thesis focuses on software related issues in avionics the current section gives a deeper insight to software related issues than to hardware ones.

Certification Process  The objective of the Certification Process is to demonstrate that the aeronautical product meets regulations and its functional and safety requirements. The certification process extends from Proof of Concept through program development and tests, to in-service operations, including continued airworthiness (maintenance).

The airworthiness is the ability of an aeronautical product including all of its subcomponents to (i) satisfy applicable rules and regulations, (ii) meet intended functions and (iii) be operated in safe conditions for people. Usually the certification process can be separated into two tasks:

- The Design Assurance Specification Process aims to define the Development Assurance Level (DAL, see in Section 9.3.1 for more details) to each aircraft functions, systems and components based on the criticality of the consequences of their failure.
Figure 9.4: Overview on Aeronautical System Certification

- The Development Assurance Process is, as all planned and systematic actions used to demonstrate (see in Section 9.3.2) – to an adequate level of confidence based on the DAL levels –, that errors in requirements, design, and implementation have been identified and corrected in order to ensure that the system satisfies the applicable certification basis.

9.3.1 ARP-4754: Certification Considerations for Highly Integrated or Complex Aircraft Systems

ARP-4754 [SAEc] discusses the certification aspects of highly-integrated – refers to systems that perform or contribute to multiple aircraft-level functions – and complex – refers to systems whose safety cannot be shown solely by test and whose logic is difficult to comprehend without the aid of analytical tools – systems installed on aircraft, taking into account the overall aircraft operating environment and functions. It was developed in the context of the EASA CS-25 certification regulations as a mean to comply with the regulation rules.

ARP-4754 addresses the total life cycle for systems that implement aircraft-level functions. It excludes specific coverage of detailed systems, software and hardware design processes beyond those of significance in establishing the safety of the implemented system. These specific aspects are detailed in separate standards but always executed in parallel as design considerations in one process may have significant effect on the others:

- Methodologies for safety assessment processes are outlined in ARP-4761 (see in Section 9.3.2).

- Coverage of complex hardware aspects of design are dealt with in RTCA document DO-254 and its EUROCAE counterpart, ED-80 (see in Section 9.3.3).
More detailed coverage of the software aspects of design are dealt within RTCA document DO-178B and its EUROCAE counterpart, ED-12B (see in Section 9.3.4).

Figure 9.4 outlines the relationships between the various documents which provide guidance for system development, safety assessment, and the hardware and software life-cycle processes. ARP-4754 is intended to be a guide for both the certification authorities and applicants for certification of highly-integrated or complex systems, particularly those with significant software elements. As such, the focus is toward ensuring that safety is adequately assured through the development process and substantiating the safety of the implemented system.

**System Development Assurance Level** One key element of the ARP-4754 standard is the introduction of *System Development Assurance* as a means of certification. It is based on the following principles:

- Highly integrated and complex systems present greater opportunities for development errors and undesirable unintended effects. Since these errors are generally not deterministic and as suitable numerical methods for characterising them are not available, other qualitative means should be used to establish that the system can satisfy safety objectives.

- Development assurance establishes confidence that the system development has been accomplished in a sufficiently disciplined manner to limit the likelihood of development errors that could impact aircraft safety.

Development assurance is a process involving specific planned and systematic actions that together provide confidence that errors or omissions in requirements or design have been identified and corrected to the degree that the system, as implemented, satisfies applicable certification requirements.

Systems and components are assigned “development assurance levels” (DAL) based on failure condition classifications associated with aircraft-level functions implemented in the systems and components. The rigor and discipline needed in performing the supporting processes will vary corresponding to the assigned development assurance level. The system DAL is assigned based on the most severe failure condition classification associated with the applicable aircraft-level function(s).

The main guidelines behind the definition of DAL for the system and its components are the following:

- For complex systems a primary development assurance level is based on the overall system architecture through the allocation of risk.

---

<table>
<thead>
<tr>
<th>Failure Condition Classification</th>
<th>System Development Assurance Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catastrophic</td>
<td>A</td>
</tr>
<tr>
<td>Hazardous/Severe Major</td>
<td>B</td>
</tr>
<tr>
<td>Major</td>
<td>C</td>
</tr>
<tr>
<td>Minor</td>
<td>D</td>
</tr>
<tr>
<td>No safety effect</td>
<td>E</td>
</tr>
</tbody>
</table>

Table 9.1: Design Assurance Levels
• For components that supports multiple aircraft functions, the applicable safety requirement should be based on the most severe of the effects resulting from failure or malfunction of any supported function or its combination.

• If it is proved that the system architecture provides containment for the effects of design errors, development assurance activities can be conducted at a reduced level of process rigor for the system components within the architectural containment boundary.

• If a system has multiple categories of failure conditions associated with its different functions, architectural means may be used to limit the interaction between items. This may allow the separate items to be developed at different assurance levels.

• System architectural features, such as redundancy, monitoring or partitioning, may be used to eliminate or contain the degree to which an item contributes to a specific failure condition, allowing simplification or reduction of the necessary assurance activity.

Finally, ARP-4754 defines how development assurance levels are associated to the recommended activities contained within the supporting processes: (i) Software level assignment – as defined in DO-178B – are directly related to their corresponding failure condition classification and (ii) hardware level assignment is treated in a similar way if the safety of the hardware component cannot be demonstrated through deterministic techniques due to its complexity.

9.3.2 ARP-4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment

ARP-4761 [SAEd] provides general guidance in evaluating the safety aspects of a design. For this purpose, it describes guidelines and methods of performing the safety assessment for certification of civil aircraft. This standard is a collection of all safety analysis methods that can be used as part of the functions, systems and equipment assessment for safety. The intent of this document is to identify typical activities, methods, and documentation that may be used in the performance of safety assessments for civil aircraft and their associated systems and equipment.

The guidelines and methods provided in ARP-4761 are intended to be used in conjunction with other applicable guidance materials, including ARP-4754, DO-178B and DO-248B, and with the advisory material associated with FAR/CS-25.1309.

Safety Assessment Process The System Safety Assessment Process is the complete process applied during the design of the system to establish safety objectives and to demonstrate compliance with FAR/CS-25.1309 [Eurb] and other safety related requirements.

It provides a methodology to evaluate aircraft functions and the design of systems performing these functions to determine that the associated hazards have been properly addressed. The safety assessment process is usually qualitative and only in certain scenarios contains quantitative aspects.

The main goal of the process is to provide the necessary assurance that all relevant failure conditions have been identified and that all significant combinations of failures, which could cause those failure conditions have been considered. The ARP 4761 document presents guidelines for conducting an industry accepted safety assessment consisting of:

• Functional Hazard Assessment (FHA) consists in identifying and classifying the failure condition(s) associated with the aircraft functions and combinations of aircraft functions. These
failure condition classifications establish the safety objectives. The output of the FHA is used as the starting point for conducting the PSSA

- **Preliminary System Safety Assessment (PSSA)** is a system evaluation of the proposed architecture(s) and implementation(s) based on the Functional Hazard Assessment (FHA) failure condition classifications to determine safety requirements of the system. It establishes the safety requirements of the system and determines that the proposed system architecture can reasonably be expected to meet the safety objectives identified by the FHA (i.e. how failures can cause the functional hazards of the FHA).

- Finally, the **System Safety Assessment (SSA)** is a comprehensive evaluation of the implemented system to be certificated to show that the qualitative (and quantitative) safety requirements as defined in the FHA and PSSA have been met. It evaluates the implemented system to show that safety objectives from the FHA and derived safety requirements from the PSSA are met.

**Safety Analysis Methods** Next to the overall process definition ARP-4761 also presents information on the safety analysis methods needed to conduct the safety assessment. Without going into details these include:

- **Fault Tree Analysis, Dependence Diagram and Markov Analysis**: All these approaches are top-down analysis techniques. After identifying the failure conditions in the FHA, these techniques can be applied as part of the PSSA to determine what single failures or combinations of failures can exist (if any) at the lower levels that might cause each failure condition.

- **Failure Modes and Effect Analysis and Summary**: It is a systematic, bottom-up inductive analysis method for identifying the failure modes of a system, component, or function and determining the effects on the next higher level of design. Its purpose is to identify the effects of each failure on the system and support the other analysis methods.

- **Common Cause Analysis**: It evaluates the overall architecture sensitivity to common cause events (in particular individual failure modes or external events) which can lead to a catastrophic or hazardous/severe-major failure condition. Common cause is defined as an event, which bypasses or invalidates redundancy.

**9.3.3 Design Assurance Guidance for Airborne Electronic Hardware: DO-254**

The purpose of RTCA/DO-254 [RTCb] is to provide guidance for design assurance during the development from conception through initial certification and subsequent post certification product improvements to ensure continued airworthiness of airborne electronic hardware such that the hardware performs its intended function in a specified environment.

One key element of DO-254 is that it defines the difference between simple and complex hardware. A hardware component is considered **Simple** if a comprehensive combination of deterministic tests and analyses can ensure correct functional performance under all foreseeable operating conditions with no anomalous behaviour. All items that are not simple are considered to be **Complex**.

The main importance of this definition is that if a hardware component is considered as simple than its certification is straightforward using extensive testing and there is no need for the application of complex certification approaches as detailed in the DO-254.

The procedure defined in the standards are applicable, but not limited to (i) line replaceable units (LRUs), (ii) circuit board assemblies, (iii) custom micro-coded components (i.e., application specific integrated circuits (ASICs) and programmable logic devices (PLDs)), (iv) integrated technology components (i.e., hybrids and multi-chip modules) and (v) commercial-off-the-shelf (COTS) components.

### 9.3.4 Software Considerations in Airborne Systems and Equipment Certification: DO-178B

The purpose of RTCA/DO-178B is to provide guidelines for the production of software for airborne systems that performs its intended function with a level of confidence in safety that complies with airworthiness requirements. These guidelines are in the form of: (i) objectives for software life cycle processes, (ii) descriptions of activities and design considerations for achieving those objectives and (iii) descriptions of the evidence that indicate that the objectives have been satisfied. An example objective from the standard is the following: Executable Object Code compiles with low-level requirements.

However, software is never certified as a standalone artifact and is always treated parallel to the hardware and system development procedures as they all depend on each other. How these dependencies and interactions are executed between the software and system development procedures are defined in DO-178. It focuses on two aspects: (i) define how to keep track of requirements allocated to the software components in the system design, especially those that contribute to the system safety and (ii) specifies the direct mapping of failure conditions –as defined in ARP-4754 – to specific software levels (see in Table 9.2). For each software level DO-178B defines a set objectives that must be met in order to be accepted by the regulation bodies. These objectives are mainly related to the software lifecycle process and the integral process.
9.3. CERTIFICATION OF AIRWORTHINESS: PROCESSES AND STANDARDS IN CIVIL AERONAUTICS

9.3.4.1 Software Lifecycle Process

For the complete software lifecycle process, DO-178B defines objectives for each of the activities taken during the planning and development and outlines guidelines for meeting these objectives:

Software Planning

The software plans include consideration of methods, languages, standards, and tools to be used during the development. One key goal of the planning is the precise definition of transition criteria, which specify if a process may be entered (or reentered). However, as different development models (waterfall, V-model, iterative, etc.) require different criteria to be satisfied for moving from one activity to another, specific transition criteria are not defined in DO-178B.

The Software Development Process

The Software development process includes requirements, design, coding and integration as separate activities.

In DO-178B requirements are allowed to be developed to the level that detail the functionalities of the system on two levels, referred as high-level and low-level requirements. High-Level Requirements (HLR) are typically derived from system level requirements and their definition implies a black-box view of the software. Low-Level Requirements (LLR) are software requirements derived from high-level requirements and design constraints from which source code can be directly implemented without further information. LLR are usually specified as part of the requirements decomposition process. Additionally, next to the LLR the Software Architecture (SA) refers to the structure of the software selected to implement the software requirements. It defines (i) what software components are exist, (ii) the interfaces to those components, (iii) how components are scheduled and invoked, (iv) and how information flows between the components. Certain low-level requirements may be directly derived from the design, architecture or implementation of the software (and hardware). These requirements called Drived Requirements (DR) do not have direct traceability to the system requirements, however, they must also be verified and considered for safety related issues in the system safety assessment process.

Ultimately, where, how or on what level requirements are defined is less important than ensuring that all of them are accounted in the designed and implemented software component and that traceability up to the system requirements is maintained to support verification.

As for design, coding and integration processes DO-178B provides only a brief description since these may vary between various development methodologies. It separates two conceptual artifacts: (i) Source Code (SC) is the code written in source languages, such as assembly language and/or high level language, in a machine readable form for input to an assembler or compiler and (ii) the final Executable Object Code (EOC) is obtained by traditional compilers and the resulting code can be integrated onto and executed without further problems on the Target Computer.

Additionally, on one aspect the DO-178B is quite strict as it specifies the outputs of each processes, which can be summarized as follows: (i) the design process produces the low-level requirements and the software architecture, (ii) the coding process is responsible for the source code and finally, (iii) the result of the integration is the executable assembly on the target system with all configuration and link files. Each of these outputs is verified, configured and assured as part of the integral process.
9.3.4.2 Integral Process

The Integral Process defines all the software verification, configuration management, certification liaison and software quality related issues that are needed to be performed parallel to the development process to be compliant with the designated software level.

Software Verification

Software verification objectives outnumber all others in DO-178B and represent over sixty percent of the overall objectives. There are specific verification objectives for all development activities and for the verification process itself. In all cases, focus is placed to assure that there is traceability from the high-level requirements down to the final integrated executable assembly on the target platform.

More specifically, (i) LLRs should be consistent with HLRs, (ii) the selected software architecture (SA) should be in conformant with HLRs, (iii) the software architecture is aligned with LLRs, (iv) source code (SC) complies with both the LLRs and software architecture (SA) and finally (v) for Level A software it must also be shown that the executable object code (EOC) is the equivalent of the source code.

In DO-178B verification is defined as a combination of reviews, analysis and testing.

Reviews are performed to provide qualitative assessment of a process or product. Common types of reviews are requirements, design and test procedure reviews. However, how to perform these reviews are not defined in DO-178B but industry best practices suggest to use checklist as they provide objective evidence and is a practical traceable means that the activity has been taken.

The aim of analyses is to provide repeatable evidence of correctness. Usually these are performed as specific algorithms or formal verification techniques. Typical types of analyses used include data flow, timing, memory leakage, control flow and race condition analyses. Due to their algorithmic nature these analyses are often performed using third party tools for which DO-178B tool qualification must be followed (see in Section 9.3.4.2).
Finally, testing is performed to demonstrate that (i) the software performs its intended function and (ii) does not show any unintended actions. DO-178B specifies a requirement-based testing approach, where the ultimate goal is to demonstrate that all requirements are tested. Additionally, next to this requirements coverage analysis a structural coverage analysis is also performed to determine the extent to which the test cases exercised the code. This structural coverage analysis is used as an indicator for complete test completion. How much testing is required for compliance at the various software levels are mainly driven by the requirement on source code coverage. A brief comparison on requirements for structural coverage for each software level in DO-178B is highlighted in Table 9.3. In general DO-178B structural coverage requirements are the primary cost driver on avionics software development projects and requires thoughtful test case planning that maintains a constant focus on requirement coverage.

### Software Configuration Management, Quality Assurance and Certification Liaison

The aim of the software configuration management is to ensure that changes are accomplished in a controlled manner. It is defined in six objectives that must be met for all software levels. This includes all activities for establishing configuration identification, baselines and traceability definition, problem report specification and management, change control and archival of data. It is important to note that in the aeronautical word support for certain systems can be easily measured in 10-years, thus configuration management is vital for effective support.

Software quality assurance process (SQA) objectives provide a general oversight on the entire DO-178B processes and always require complete independence on all software levels. The main goal of the SQA is to ensure that any deviations during the development process from plans and standards are detected, tracked and finally, resolved.

The certification liaison process is designed to guide the certification process by defining in the beginning what you intend to do and at the end provide all evidence that what you did actually. The personnel responsible for this process is the interface between the regulation bodies and the development team and do the negotiation and presentation for all deliverables including legal issues.

### Tool Qualification

As an additional consideration, DO-178B requires qualification of tools, which are used to reduce or automate processes noted by the standard when their output is not being verified and used in the development. Tools are classified as development and verification tools:

- **Development tools** produce outputs that becomes part of the avionics system and thus can introduce errors. In this case the tool is required to satisfy the objectives at the same level as

<table>
<thead>
<tr>
<th>SW Level</th>
<th>Coverage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>MCDC</td>
<td>Level B + 100% Modified Condition Decision Coverage</td>
</tr>
<tr>
<td>B</td>
<td>DC</td>
<td>Level C + 100% Decision Coverage + Independent Designer and Verifier</td>
</tr>
<tr>
<td>C</td>
<td>SC</td>
<td>Level D + 100% Low-Level Requirement Coverage +100% Statement Coverage</td>
</tr>
<tr>
<td>D</td>
<td>-</td>
<td>100% High-Level Requirement Coverage</td>
</tr>
<tr>
<td>E</td>
<td>-</td>
<td>No coverage requirements</td>
</tr>
</tbody>
</table>

Table 9.3: DO-178B Coverage Requirements
the software it produces. Due to this very strict regulation, complex development tools for the aeronautical word are very rare and expensive.

- Verification tools on the other hand cannot introduce errors but may fail to detect them. Qualification objectives for verification tools requires the demonstration of its requirements under normal operational conditions. These demonstration requirements are far less restrictive than in case of development tools.

Which tool belongs to which classification is usually specified by the regulatory bodies, however, in certain cases as it is not clear a tool belongs to which category the developer can negotiate over its classification with the regulatory body.

9.3.4.3 Summary

DO-178B does not describe how to assess safety risk. Instead it uses the results of a system safety assessment process performed according to ARP-4761 to determine what level of rigor is required to achieve confidence in the behaviour of the software and maintain a direct traceability between the system requirements down to the software to make sure that all requirements are properly addressed. Additionally, it is important to mention that DO-248B [RTCc] is an extension to the original standard and addresses (i) issues that were not clear in the original document and (ii) technologies that were not available at that time.

9.3.5 Future of Aeronautical Software Certification: DO-178C

As the future implementation of the Software Considerations in Airborne Systems and Equipment Certification DO-178C [RTCf] retains the core software development process from DO-178B, updating it where DO-178B was not completely precise or lack certain regulations. However, what is most significant about DO-178C is the addition of four novel technology-specific supplements that define how to earn certification credits that ease certain certification requirements for the core development process. These new technology concepts are:

- The Model-based Development and Verification supplement (DO-331) supports the development at higher levels of abstraction than source code providing a mean to handle the complexity of software components. This way DO-178C allows to use models as a means to capture requirements for the software and generate source code from these models. However, the two key questions [Cla09] here are (i) the ability to maintain traceability between the models capturing requirements and the generated source code and (ii) the challenges of integrating generated code with manually written ones as these separate code segments cannot be verified until the integration phase, which is problematic for an early complete low-level requirement coverage. In any case, how these issues are handled depends on the characteristics of the modeling language and the mechanisms supported by the MBD tool auto-generating the source code.

- The Software Tool Qualification supplement (DO-331) defines a new classification method for tools used in the development process. There are three categories, where two of them are the ones already introduced in DO-178B as Development and Verification tools (see in Section 9.3.4.2 and the new category is called Certification tools. It is a tool that automates a verification process and whose output is used to justify the elimination or reduction of (i) a verification process other than that automated by the tool, or (ii) a development process which could have an impact on the embedded software. Additionally, the supplement defines 5 tool
qualification levels TQL (similar to the DALs), which specify what kind of certification evidences are required for the various tools in order to achieve a certain TQL. Typically, the TQL level of a tool is based on its category and the DAL level of the embedded software under development (e.g., a certification tool for Level B software has a TQL level of 4, where TQL 1 is the most rigorous).

- The *Object-Oriented and Related Technologies* supplement (DO-332) focuses on currently available OO languages used today such as C++, Safety Critical Java and Ada 2005. Its aim is to address the issues of determinism in OO languages. One of the most problematic part is subtyping. It is the ability to create new types or subtypes in an OO language, although powerful, introduces the challenges of maintaining type consistency and subtype verification. To handle this problem the supplement recommends to use an exhaustive, flattened class approach to ease verification and lower nondeterminism.

- The *Formal Methods* supplement (DO-333) allows to use mathematical proofs as an additional means of verification. As formal methods were already used for decades in the development of avionics software the aim of this supplement to precisely specify how to obtain certification credits using such techniques. However, DO-178C advocates target testing to ensure that the code works correctly on the target platform.

The growing complexity of current avionics system is related not only to their safety-critical aspects but also from the many combination of options offered by the various suppliers. The emerging DO-178C aims to bring novel software development techniques to handle this complexity growth and achieve cost-effective certification with backward compatibility to already used methodologies that are accepted in DO-178B. In its current form DO-178C was ratified in 2012, however, up to know we do not know about any project that has been certified using this version of the standard. For this reason, we will mainly refer to the older DO-178B version in the current thesis.

### 9.3.6 Integrated Modular Avionics (IMA) development guidance and certification considerations: DO-297

The DO-297 [RTCd] standard provides design assurance guidance for the development and certification of modular avionics by IMA developers, integrators, applicants, and those involved in the approval and continued airworthiness of IMA systems. It provides specific guidance for the assurance of IMA systems as differentiated from traditional federated avionics.

The key advantage of DO-297 is the introduction of *Incremental Certification*. It applies at acceptance of different parts of the system (Module, Applications, System, Aircraft) to facilitate change and reuse of these parts. Six tasks define the incremental acceptance of IMA systems in the certification process, where a task is associated to the acceptance of the different parts of the system in the context of DO-297 incremental certification process. An overview how these task are related to each other are depicted in Figure 9.6.

**Task 1 - Module Acceptance**  The purpose of module acceptance within the overall certification process is to demonstrate the module characteristics, performance, and interfaces to obtain incremental acceptance of the module. This is accomplished by providing documented evidence (acceptance and/or compliance data) for the benefit of the other IMA system acceptance tasks and for potential reuse. If reuse is a desired outcome for the module development, reuse should be addressed during the development of the module (see Task 6).
The key task for reusability is the precise definition of the *usage domain*. This document defines the assumptions made by the developer regarding the subsequent use, installation configuration, and V&V activities to be conducted by the system integrator.

**Task 2 - Application Acceptance**  An application is software and/or application-specific hardware with a defined set of logical interfaces that, when integrated with a platform, performs an aircraft function or part thereof.

The main goal of application acceptance within the overall IMA system acceptance process is to demonstrate that the application complies with the applicable regulations and requirements allocated by IMA system design, performs within the module limitations, and provides the characteristics and performance as specified. Another goal is to provide acceptance data and compliance evidence for the benefit of the integration of the application in the IMA system and its potential reuse on subsequent projects.

Software and/or hardware applications intended for future reuse should be developed using available guidance such as DO-178 and DO-254, respectively.

**Task 3 - IMA System Acceptance**  The main goal of IMA system acceptance is to demonstrate that the integrated modules, hosted applications, and the platform continue to perform their intended functions and do not adversely affect other hosted applications or modules. The activities may be performed on or off the aircraft. For off-aircraft activities, a major goal is to perform V&V activities that can be applied toward the overall aircraft certification effort.

The delineation between Tasks 3 and 4 will vary significantly by project. Therefore, the life cycle data for Tasks 3 and 4 may be combined or allocated as appropriate. Any life cycle data not addressed in Task 3 should be completed in Task 4.

**Task 4 - Aircraft Integration of IMA System**  The final IMA system installation, integration, and Verification & Validation activities are similar to those that would be conducted on a federated system architecture, demonstrating that each aircraft function and hosted application functions as intended, supports the aircraft safety objectives, and complies with the applicable regulations.

However, during the installation activities, the interactions between hosted applications relative to the provided aircraft functions should be verified and validated during aircraft ground and flight
testing. Also, the interactions, interfaces, and connections between the IMA system and other aircraft systems should be verified and validated. Any IMA system life cycle data that were not addressed in Task 3 should be completed as part of Task 4.

**Task 5 - Change** A primary objective of the IMA system development and acceptance process is to minimize the impacts of an IMA system component change on the IMA system and aircraft certification. Only the changed module(s) and/or application(s) could require re-acceptance or re-approval when considering installation, safety, operational, functional, and performance issues. The main goal of the change process within the IMA system is to bound changes in such a way that their effects are known and can be fully verified and validated.

The major activity is to conduct and document the Change Impact Analysis (CIA). The CIA has to deal with the usage domain of the module as defined in Task 1.

CIA requires to ensure that the change in the module or application has no adverse impact on affected, but unchanged modules and applications. However, it does not prevent to perform all necessary verification, validation, and integration activities (including regression analysis and testing) to obtain acceptance of the modified module or application.

**Task 6 - Reuse** The main goal of reuse process is to be able to use module or application life cycle data that has been previously assured and accepted, with minimal need for oversight by the certification authority and of course for costs reduction. However, it should be noticed that any subsequent changes in the module or the application life cycle data can be done. In fact, practical experience shows that changes should be small enough to be economically satisfactory – as the CIA defines –, because in any other case, the entire certification process has to be rerun from scratch using program’s certification needed level. Reuse should be planned during the initial development process. Modules are accepted with the intent of being reused in multiple systems.

**9.3.6.1 Summary**

DO-297 sets the basis for incremental certification that holds the key to battle the complexity of future avionics system. This is done with the proper definition of boundaries between modules and applications, which is the key for effective incremental certification.

**9.4 Summary**

The current chapter gave an overview on the actual state on certification of airworthiness in the civil aeronautical domain. It focused on the introduction of standards for avionics software development. Additionally, it gave an overview on incremental certification as defined for IMA systems.
10.1 Introduction

10.1.1 Motivation: Development of ARINC 653 Configuration Artifacts

Unquestionably, the ARINC 653 standard [ARIb] has taken a leading role within the aeronautical industry in the development of safety-critical systems based on the Integrated Modular Avionics concept. One of the main promises of IMA is cost saving in reduced development, integration and verification and validation effort.

In case of ARINC 653 compliant platforms many deployment and implementation details are defined in configuration tables. Typically, these configurations are defined directly by the system architect with limited tool support that only ease (i) the manipulation of its XML representation, (ii) their validation to the ARINC 653 schema definition and some basic consistency checks.

Unfortunately, despite the inherent complexity of ARINC 653 configurations, current tools supporting configuration design offer very low-level support directly on the XML representation level. Existing tools lack of support for (1) capturing the development process for configurations, (2) validating design constraints for configurations on-the-fly and (3) providing traceability between high-level requirements and the configuration tables, which require hand-crafted traceability lists. As a result, design and verification of configuration tables is a tedious and error prone activity.

Model-driven engineering (MDE) has become a key technique in system and software engineering [MS08]. It facilitates on systematic use of models from a very early phase of the design process. However, as MDE is attracting increasing attention in safety-critical system development, it needs to be adapted to be in-line with the rigid certification requirements (see DO-178B in Section 9.3.4) imposed by regulation bodies.
10.1.2 Contribution

In the current chapter, we present a model-driven approach for systematically designing standard ARINC 653 configuration tables by supporting model-based validation of design decision consequences. It is able to configure (i) the Wind River VxWorks 653 Safety Critical RTOS [Wil07] and (ii) the GMV SIMA ARINC 653 simulation platform [SRS08]. Additionally, it can also generate configuration artifacts for the AIDA (Architecture for Independent Distributed Avionics) middleware developed in the DIANA project as an abstraction layer over ARINC 653 (see in 10.4.3.3).

Our approach is based on Process-Driven MDE [6] that uses model transformation services organized into complex model transformation chains. These transformation chains are closely aligned with the designated development process as driven by precise workflow models, where workflow activities comprise of individual development steps carried out by either internal (e.g., a form for defining attributes of model elements) or third party external tools (e.g., an optimization framework like VIATRA2 DSE).

Our realization of the Process-Driven MDE approach followed the strict separation of modeling layers as defined in MDA [Obj01] and provides a clear distinction between Platform Independent Models (PIM) and Platform Specific Models (PSM).

Additionally, in parallel to the development process, our approach generates end-to-end traceability information starting from the high-level engineering models down to the generated configuration artifacts. Moreover, individual development steps are guarded by design contracts describing certain functionalities carried out by each step, allowing a light-weight automated model-level validation for early error detection and localization.

10.1.2.1 The DIANA project

DIANA, Distributed, equipment Independent environment for Advanced avioNics Applications [DIA], was an aeronautical research and development project funded through the European Commission’s 6th Framework Programme and led by GMV, Portugal with leading avionics experts and airframers including GMV, AleniaSia, Atego, Dassault, Embraer, NLR, THALES, and academic partners of Budapest University of Technology and Economics and Karlsruhe Institute of Technology.

The project aimed at the definition of an advanced avionics middleware, called AIDA (Architecture for Independent Distributed Avionics), supporting (i) the execution of object-oriented applications over virtual machines, (ii) high-level publish-subscribe like communication abstraction, and (iii) the applicability of model-driven engineering techniques in the development process.

Our main contribution to the project was the specification and implementation of an MDE based framework for the systematic design of configuration artifacts for ARINC 653 and partly AIDA based applications. Our primary contribution lies in the adaptation of MDE based techniques in the context of avionics system configuration development with a special interest in integration of certification means to the complete development process.

10.1.3 Structure

The rest of the chapter is structured as follows: Section 10.2 presents an air-conditioning case study from the avionics domain, in Section 10.3 we introduce our development process based on model-driven techniques as defined within the DIANA project, Section 10.6 highlights how we adapted two separate model-driven techniques to support certification activities for DO-178B. Section 10.7 intro-
10.2 Case Study: Air Conditioning

In the current chapter, we use a generic air conditioning system (installed on an aircraft) as our running example to demonstrate our approach. An overview of the case study is depicted in Figure 10.1.

An air conditioning system aims to regulate the temperature and pressure in the aircraft. This is carried out in the following way. The air conditioning pack is regulated by the pack controller to supply the mixing unit with a sufficient flow of cool fresh air. This air is supplied to an arbitrary number of zones (in Figure 10.1 we depicted two zones Aft and Forward). In order to regulate the temperature of this airflow, the zone controller regulates the amount of hot air added to the flow of cool air, which is set on the air conditioning panel and monitored on the system display. Additionally, as air-conditioning is a critical task all components have a redundant equivalent for better reliability. An overview of the air conditioning system is depicted in Figure 10.1. It is a simplified version of the NLR demonstrator in the DIANA project [DIA].

10.3 Overview of the Approach

The complete development process from requirements to deployable configuration artifacts is composed of four phases and depicted in Figure 10.2.

**Specification** The main goal of the specification phase is the generation of the platform independent models of the individual ARINC modules with respect to specification of the Architectural requirements like functionality (e.g., applications and messages with their types), dependability (e.g., redundancy degrees), etc. These models are either derived from already available COTS modeling languages – like Matlab Simulink [Mat] – or defined by the system integrator or architect using a dedicated IDE. The models used to specify the platform independent architectural description of the modules are discussed in Section 10.4.1.1.

Additionally, the specification phase encapsulates the definition of the Platform Description that captures the resources available for the system under development. In our case these
are mainly predefined ARINC 653 resources (partitions, channels, etc.) and available AIDA middleware services.

**Design** The aim of the design phase is to synthesize the Platform Specific Model (PSM) from the PIMs defined in the specification phase. The mapping process consists of several refinement steps and usually requires an interactive process. It involves refinement of abstract information into concrete platform specific definitions like mapping data type names used in PIM to platform specific types, denoting messages defined in the PIM by ICD specific information, or associating AIDA middleware services to applications (e.g., logbooks). It is followed by the allocation process where (i) applications are allocated to ARINC 653 partitions, and (ii) ARINC 653 ports are allocated to the partitions based on the communication schema defined in the PIM. How these steps are organized and executed is defined in the *PIM-PSM mapping process* and discussed in Section 10.5.

**Implementation** The PSM is then used to derive (i) the AIDA middleware configuration XMLs, (ii) the ARINC 653 deployment configuration XML (as the input for the SIMA simulation platform) describing the structure of the allocated partitions over a module along with the communication resources used and (iii) the VxWorks specific auxiliary configuration tables. How the configuration files are defined and generated is discussed in Section 10.7.3.3.

**Deployment** Finally, all configuration that needs to be deployed on the actual hardware or fed into the simulator is compiled and linked together with the other components to form the complete executable system. For this phase our approach does not provide any additional feature.

**V&V** Additionally, we tightly integrated two verification and validation extensions to the development process to support certification:

- As an essential requirement of DO-178B certification, end-to-end traceability is carried out from the high-level models to the generated configuration files. In case of the design phase we proposed a model-based traceability approach that keeps track of all manipulations done during the PIM-PSM mapping process. As in the implementation phase separate traceability files are generated that links the model elements in the Integration model with their corresponding configuration element. This effectively complements the (Level A) certified XML to Binary compiler for the VxWorks 653 platform as our approach provides the necessary traceability for the complete development process down to the configuration XML definition, while the certified compiler generates the required binary (and traceability tables) for the actual deployment; thus eliminating the need for testing them.

- Analogously to the DO-297 tight interfacing schema for incremental certification, we used *design contracts* - defined by graph patterns - to guard the input and output models of each separate development steps, thus specifying the intended behavior of the development step without any restrictions on its internal implementation. Additionally, these contracts can also be used for early error detection and localization by using on-the-fly contract evaluation techniques as realized by our EMF-IncQuery [13] framework.

Both techniques are discussed in Section 10.6.1 and Section 10.6.2, respectively.
10.4. MODELING ARCHITECTURE

Transforming high-level models into low-level configuration artifacts is a complex task, which needs to bridge a large abstraction gap. Consequently, a multitude of transformation steps (both manual and automated) are done in the course of the PSM generation. Moreover, critical design decisions are also made during this mapping process. For this reason, we define intermediate models to subdivide the mapping problem into well-defined subproblems (e.g., mapping, allocation, etc.). The overview of our modeling architecture is depicted in Figure 10.3.

The aim of the PIM is to capture the high-level architectural view of the system along with the definition of the underlying implementation platform, while the PSM focuses on the communication details and service descriptions. Finally, the target platform specific configuration artifacts are derived from these PSM models along with the required traceability information.

10.4.1 Platform Independent Models

In order to support already existing modeling tools and languages – in our case, Matlab Simulink – we use a common architecture description language called Platform Independent Architecture Description Language (PIADL) for the description of architectural details by extracting relevant information from common off-the-shelf modeling tools. As for capturing the underlying ARINC 653 platform we use a Platform Description model (PDM) capable of describing common resource elements defined in the platform (e.g., module). Finally, functional requirements can be incorporated into the PIADL and the
10.4.1.1 Platform Independent Architecture Description Language

The goal of the Platform Independent Architecture Description Language (PIADL) is to provide a cross-domain formal framework for the discussion of hardware-software integration of embedded systems. The language contains the common, most important concepts of the various domain-specific standard languages (e.g., component, port, connections, etc.), but do not contain any specific element of the various application domains (like aerospace, industrial control or automotive). The languages can be treated as a common ancestor of the domain-specific languages that inherit and specialize the core concepts to specific needs (like SysML). The language was developed in the context of the European DECOS FP 6 research project [DECa] and formally specified by András Balogh in his PhD thesis [Bal].

Without going into details, we highlight the important elements of the metamodel through our running example. Figure 10.4 identifies a small extract of the complete model describing how a Pack Flow and Zone Controller are connected with the Air conditioning panel (for easier readability only a small number of messages are displayed). An application (Job) represents a stand-alone, schedulable software entity that can communicate with each other. Actuators and Sensors are connected to these jobs representing interfaces with the environment and finally, all of these elements communicate via messages (highlighted as envelopes) like the PackFlowValveSetting and the TempSelection. In case of jobs, messages are sent through ports that are part of interfaces to give a better organization of communication channels between various elements. These interfaces and ports are depicted by rectangles attached to the jobs containing dots

Figure 10.3: Overview of the Modeling Architecture

PDM by the system architect.
10.4.1.2 The Platform Description Model

The Platform Description Model (PDM) describes the resource building blocks, which are available in an ARINC 653 Module to assemble the overall resources of an application. This mainly includes standard ARINC 653 element such as partitions, communication channels, ports. The metamodel mainly focuses on the Partitions and the communication Channels between them with a possible extension to capture the system, module and partition HM tables. The metamodel strictly follows the structure of the ARINC 653 configuration schema [ARIb] with some clarifications and extensions like, abstract elements that are introduced to use generalization for separation of concerns.

An extract of the PD metamodel is depicted in Figure 10.5. It captures how all major ARINC 653 components are generalized from the ModelElement class.

10.4.2 Platform Specific Models

The platform specific models contain all relevant low-level details of the modelled system.
10.4.2.1 The Integrated System Model

As an intermediate model the PIM-PSM mapping editor creates a so called Integrated System Model (ISM), that contains all information from the PIADL and the PD along with the decisions made during the mapping process to generate the appropriate configuration artifacts.

The ISM contains more than a 150 elements and thus is the most complicated metamodel used in our approach. It consists of three separate main domains as depicted in Figure 10.6:

**Platform Definition (PDD)** The PDD defines the complete platform resources used by the applications. It mainly depends on the Platform Description Model but it also provides the necessary auxiliary model elements (e.g., mapping relations for the applications, compatibility relations, etc) for capturing the relations between the software and hardware elements.

Additionally, it defines all VxWorks and SIMA specific modeling elements that are needed for their specific configuration artifacts (e.g., page size, virtual/physical memory limits, etc.).

**Application Component Descriptor (ACD)** The ACD captures the complete communication architecture between the various software components. It is based on the architecture defined in the PIADL but specifies all additional elements that are specific to the underlying platforms (e.g., periodic or aperiodic messages, deadlines, etc.). This part shows many similarities with the 2nd edition of the AADL standard [SAEb], how it handles virtual and physical buses and routers.

Additionally, the ACD specifies the communication interfaces of the applications. In our case these are captured by the ICD (see in Section 10.4.2.2) as it describes the internal structure of the various messages and interfaces used by the applications.

**Service Definitions (SDD)** SDD specifies the compound AIDA services required by the applications.

In our case this is demonstrated by the AIDA Logbook service defined in [37]. The tool generates the AIDA Logbook Descriptor, which defines the allocation of instances of the logbook to different modules, the structure of the logbook in terms of message size, logbook size and buffer sizes and the location of client applications, using the logbook.

Finally, the PSMRoot is the root element of the integration model and it holds references between the elements of the PIADL, PD and the PSM models.
10.4. MODELING ARCHITECTURE

10.4.2.2 AIDA Interface Control Document - ICD

The AIDA ICD is used to define the interface of a system, component, service, application, etc. in order to ease integration of the described element with other parts of a system. The focus is on the possible interactions of the considered system or application with its environment, not on its internals.

The AIDA ICD is intentionally compatible with ARINC 653 and, to a certain extent, with ARINC 825. It describes rules and recommendations that should be followed to describe a service or application and to encode messages.

Description of an ICD can be done at different levels of abstraction, starting from abstract ones (closer to designer preoccupations) to more concrete ones (e.g., making reference to physics in case of information encoding). An overview on the different levels defining an ICD is depicted in Figure 10.7.

- A high-level description (logical view or symbolic data type) is related to the description of the information the different data structures represent like dimensions, symbolic data types, units, etc.

- While the low-level (physical level) description focuses on the encoding (like data alignment, padding, byte order, etc.) of the defined data structures.

A third level is also considered, which is the language level. It describes how low-level encoding schemas can be effectively mapped to different implementation languages.

The ICD is considered to be part of the Integrated System Model, however, it was defined separately and can be used as stand alone language for data structure definition. A detailed definition of the ICD is available in [41].

For a detailed introduction to the Integrated System Model and the DIANA ICD please consult with [38] and [41], respectively.

10.4.3 Target Platforms

As mentioned in Section 10.3 our framework has three target platforms the Wind River VxWorks 653 real time operating system, the GMV Simulated Integrated Modular Avionics and the Architecture for Independent Distributed Avionics middleware.

10.4.3.1 Wind River VxWorks 653 RTOS

VxWorks 653 [Riv07b] is Wind River’s platform for safety-critical applications certifiable according to DO-178B. It is an IMA operating system with proven compliance to ARINC 653 [Sch09].

---

1The definition of the ICD is mainly the work of Damien Carbonne from Dassault Aviation
VxWorks 653 implements IMA by means of virtualization technology. There is a hypervisor monitoring and controlling a set of guests. Each guest uses its own local executive, the Partition Operating System (POS). Several types of POS are supported by the platform, such as the ARINC 653 APEX (APplication EXecutive), the classic VxWorks RTOS or a general purpose OS like Linux. Note that there is only one code instance per POS physically present in the system that is linked to the virtual address space of the partitions that actually use this particular POS.

The hypervisor is called the Module Operating System (MOS). It implements time- and space partitioning, the ARINC 653 inter-partition communication channels and the Health-Monitoring system. The MOS is the only component that runs in privileged mode. Guest systems run in user space and are not allowed to execute privileged instructions that may impact the proper function of the system. Figure 10.8 (based on [Riv07b] illustrates the architecture of the VxWorks 653 platform.

Finally, the hypervisor is running on the selected CPU via the Board Support Package (BSP). Note that the components of the system are not linked together to one image; instead individual binaries are created for the MOS, for the POSees and for the applications. The boot loader is responsible to locate the different components on the boot medium and to load them into memory according to a configuration derived from system configurations.

The configuration the system integrator has to provide in order to link, load and execute the system follows the VxWorks component structure. There is a configuration file for the MOS that defines fundamental architecture-related settings, such as processor frequency, page size and virtual and physical memory; there are configuration files for the POSees, defining their memory layout and how they are loaded into memory; there are configuration files for the applications, defining memory sizes and ports; there are Health Monitor tables that define the health monitoring on partition and module level; there is, finally, a configuration for the module bringing the single configuration files together and adding time partitioning-related information.

10.4.3.2 GMV SIMA Simulator

Simulated Integrated Modular Avionics (SIMA) [SRS+08] is an execution environment, providing the ARINC 653 Application Programming Interface (API) and robust partitioning to operating systems that do not support these features by themselves. SIMA is designed to run on all POSIX-compliant operating systems and optimised for the Native POSIX Thread Library (NPTL), available on Linux since kernel version 2.6.
In SIMA, ARINC 653 partitions are mapped to POSIX processes, and ARINC 653 processes are mapped to POSIX threads. Each SIMA application is, hence, linked to a single POSIX program, containing user code and data, the APEX code and data and, finally, the platform execution environment, i.e. the NPTL for Linux. The Module Operating System (MOS), controlling the different POSIX processes, belonging to the same simulated module, is likewise linked to one POSIX process. The architecture of the SIMA platform is depicted in Figure 10.9.

The APEX services are implemented by a static library, called POS. The POS implements the APEX process scheduler on top of the POSIX FIFO scheduler (sched_fifo). POSIX features are encapsulated within a portability layer; this way main parts of the APEX code do not rely directly on POSIX, but on scheduling policies implemented by the POS itself. The advantage of this approach is enhanced portability - there is even an implementation of the SIMA POS, running on bare hardware - and the fact that scheduler features that introduce subtle differences between different POSIX implementations are handled in the portability layer and hidden from the APEX implementation.

The MOS implements the APEX partition scheduler. To be able to suspend and resume partitions, commands are exchanged with the POS in the partitions using signals and shared memory segments. Obviously, this approach does not answer safety and security threats caused by random errors in the partitioned code. The POS has to respond correctly to given commands which may not be true in the case where faulty or malicious application code corrupts the state of the POS. In fact, the MOS does only simulate the behaviour of an ARINC 653 compliant OS on top of non-safety aware systems like standard Linux.

Since SIMA main purpose is simulation, it aims at full conformity with the standard. The SIMA configuration is therefore strictly compliant to the schema defined in today’s ARINC 653 part 1 and 2. Additional information that is needed by the system is added by means of a separate configuration file. This file defines the mapping of certain elements of the ARINC 653 configuration to the Linux OS; APEX ports, for instance, can be mapped to UDP ports.
10.4.3.3 AIDA Middleware

AIDA (Architecture for Independent Distributed Avionics) is an IMA-based middleware, backward compatible with the ARINC 653 standard. This means that AIDA is compatible with ARINC 653 COTS RTOS. It enhances aspects of ARINC 653 and the current state-of-the-art in IMA, namely it improves the neutrality of the IMA execution environment regarding the underlying hardware and operating system and it enhances the location transparency for easier communication definition. Figure 10.10 gives an overview on the AIDA architecture.

The basic building blocks in the AIDA platform are partitions as defined in the ARINC 653 standards. Partitions are fault and change containment units and as such relevant for incremental certification of applications and services as well as for application deployment and reuse. It supports two kinds of partitions based on their implementation language: C and Safety Critical Java. However, applications rely on the ARINC 653 API and all language specific features needs to be implemented over ARINC 653. Additionally, they can use services defined by the AIDA middleware to invoke local or remote services and to exchange data, based on the AIDA middleware communication means. The services of the AIDA middleware are defined by configurations given as an XML descriptor. The API level of the middleware is based on ARINC 653 and – logically - hosted as a layer in the partitions.

For a detailed introduction of the AIDA middleware consult with [37] and [DIA07b].

10.4.4 Alignment of DO-178B artifacts

In order to be compatible with DO-178B, we need to align our development artifacts used in the mapping process to the artifacts defined in DO-178B. For this alignment, we followed the considerations highlighted by Steven P. Miller in [Ste]. The mapping along with the objectives that it aims to fulfill is depicted in Figure 10.11:

- **System Requirements** are treated the same way as in case of traditional software development meaning that they are defined in “shall” statements in a natural language and all have a unique ID.

- **High-Level Requirements** are defined as the combination of the PIADL and the PDM models. They represent all the software and hardware specific information that is required to capture the functional requirements against the possible ARINC 653 configurations.

- **Low-Level Requirements** are captured at the PSM level by the Integrated System Model as it completely fulfills the original DO-178B definition of the LLR: “…requirements derived from
10.5. STEPS OF THE DIANA PIM-PSM MAPPING PROCESS

Mapping the PIM to the PSM is handled by a complex, iterative and interactive model transformation process based on the work of András Balogh [Bal] and extended in [6]. This needs to bridge a large abstraction gap, where critical design decisions made by the system architect, which cannot be automated. Therefore, we support the system architect by subdividing the mapping process.
into well-defined design steps and precisely define the contracts, interactions and interfaces of each step. Individual design steps are then organized into complex workflow-driven model transformation chains, which are closely aligned with the designated development process followed by the function/system provider or airframer. In order to assist the system architect, our framework guarantees that a certain design step can only be started if all prerequisite steps are successfully completed. This fine grain definition capability allows to easily incorporate additional design steps, if required.

The process starts with the definition of a complete PIADL model as the task of the system architect. It can be either manually defined using the (i) external PIADL editor or (ii) derived from a Simulink model, which is a near one-to-one derivation from a limited set of elements. However, as some ARINC and AIDA specific parameters cannot be directly derived from Simulink, the resulting model requires additional clarification from the system architect, which can be defined using the external PIADL editor. For example, a subsystem block in the Simulink model is mapped to a Job in the PIADL, but its modular redundancy value (how many instances of the job are required) is not present in the Simulink model.

The high-level workflow of the PIM-PSM mapping process is depicted in Figure 10.17. It consists of 28 steps organized into 6 separate groups.

### 10.5.1 Application Group

The group consists of steps to define the resource requirements of the applications and partitions used in a module and create a viable mapping that is compatible with the available resources and dependability requirements.

First, the PIADL and the PD models are imported into the framework. This step also resolves certain dependability attributes defined in the PIADL like redundancy degree of applications and messages (e.g., triple or double modular redundancy for critical applications). As the platform description does not include all information needed for the allocation process and configuration generation, the system architect needs to (i) define the memory requirements and compatibility mapping of the applications and (ii) define new partitions or modify existing ones and define their explicit memory requirements.

To demonstrate how these steps are captured on model level, Figure 10.12 illustrates the low level model elements created for a partition (partitions creation step). Model Elements in orange and dashed lines are newly created, while elements in green (and solid references) are already existing in the model. The tags «Integration» and «PD» represent the package of the model element. Partitions are defined/stored in the Platform Description model with separate model elements describing their corresponding memory requirements capturing the size, access (type) and type attributes. For easier readability (i) attribute types are excluded from the figure and (ii) references and association are depicted by simple lines.

As the final step in the group, all allocations of applications-to-partitions conforming to the defined constraints and requirements are computed.

This is done by translating the complete allocation task to a CSP(M) constraint problem as introduced in Chapter 8. As all possible solutions are calculated within the requirements, the system
10.5. STEPS OF THE DIANA PIM-PSM MAPPING PROCESS

10.5.2 Communication Group

The group involves steps in the PIM-PSM Mapping editor that carry out the allocation of inter-partition communication channels and the specification of ports residing on each end of these channels. The allocation is based on the architecture defined in the PIADL model (derived from a Simulink model), the selected application to partition mapping and the redundancy requirements of the applications. Based on this information the allocation algorithm creates the required ARINC 653 ports and connects them.

For example, in case of triple modular redundancy this can result in a large number of additional channels and ports as all input and output communication channels to and from an application needs to be generated for all its redundant instances.

Additionally, the system architect needs to define the ARINC 653 specific parts like (i) the queue length of queuing ports as they cannot be derived directly from the PIADL and the VxWorks specific queuing port protocol mapping (e.g., sender block or receiver discard) to be able to generate the configuration files. Finally, as the ARINC 653 standard has a strict naming convention the framework validates that all naming conventions are kept for the port and channel names. It also support a simple automated naming algorithm to help the system architect generate valid names.

Figure 10.13 depicts a simple example how the allocated channels are visualised. In this case the Data Monitoring application allocated over the I/O Processing partitions uses the Temp. channel to send the temperature value to the Refresh GUI application.

10.5.3 Service Inclusion Group

The Service Inclusion group involves steps to define services for the applications. First, the system architect defines, which services can be used by which applications. Within the DIANA project the logbook service was selected to exemplify how middleware services can be defined in a model-driven...
manner; thus the tool itself developed within the project gives support only for the definition of the AIDA logbook service.

Once the services are selected the system architect define the different logbooks to be used within the system. This includes the specification of their criticality level and thus the number of replicas that must be present in the system, their dedicated size and unique identifier. As the following step the created logbook are mapped to the applications, which uses them and finally the different replicas are allocated to separate modules to meet the required criticality requirements. For example, on one hand for Level D applications it is allowed to host the logbook on the same module as the applications, however, on the other hand for Level A three separate replicas on different modules (even on separate physical hardwares with unique power supplies) are required.

Again as these steps are mainly defined by the system architect the framework gives support mainly for the early validation of the definitions based on the middleware specifications.

Following the notations introduced in Section 10.5.1, Figure 10.14(a) captures how an AIDA logbook with its three separate replicas are instantiated and mapped to the Proc Input Temp application at the model level.

10.5.4 Health Monitoring Group

The group consists of steps to define the Health Monitoring recovery tables for module, partition and application levels (see in Section 9.1.1.1) along with the different error entities and actions to be carried out. It supports both the standard ARINC 653 specific error code and action declarations and also gives support for the VxWorks 653 RTOS action specifications and mode mappings.

All these definitions are done by the system architect by hand. The framework gives support for early validation (e.g., naming conventions, required action definitions etc.) based on the specification of the different tables and the system-specific requirements for health monitoring tables. The defined tables are saved in the PDD as part of the integration model.

For example, Figure 10.14(b) highlights how the different Health Monitors tables are defined and interrelated on the model level for the WxWorks RTOS. The WxWorksHM is the root of the HM Table definition and it contains the System, Module and separate Partition HMTables.
10.5.5 AIDA ICD Group

The AIDA ICD group consists of steps related to the description of messages provided and required by the different applications. First, the quantities used within the system are imported from an already defined set (e.g., velocity measured in kilometer per hour or miles per hour) captured in a specific XML file. This is followed by the definition of the platform specific data types. These types can be either elementary types like integer, float, double or composite types such as variableArray or fixedArray. Usually, types are derived from already available (or defined) ICD types using constraint on their domain.

Parallel to this step, the structure of the messages are defined. This is done in two steps, first the system architect refines the PIM messages into platform specific messages by defining the head and body parts (names) of the message. When the parts are defined their type is defined in the final step of the group by specifying the data types’ of the specified parts of the messages.

Figure 10.15 describes how the Temp PIM type is refined into the Int1_100 PSM representing an integer value with a domain of 1-100. The Int1_100 type is based on the predefined 16 bit unsigned integer type from the ICD with additional constraints over its domain. Based on these PSM types, complex messages are defined following a similar way, where the ICD provides the basic structures like arrays, buffer, etc and the system architect can construct any further message types based on these basic building blocks.

10.5.6 Artifact Generation Group

Finally, when the prerequisite steps for a certain code generator is finished the actual textual representation is synthesized by separate dedicated code generators. In our case the ICD generator simply serializes the model into its XML representation using the built in support of the Eclipse Modeling Framework. As for the other artifacts we hand-coded the generators using a template based code generator based on graph transformation rules to derive the required formats defined by the two platforms and the middleware (described in detail in Section 10.7).

A sample fragment of the generated configuration tables capturing the definition of a communication channel is captured in Figure 10.16.

Additional generated artifacts are depicted in Appendix D describing different configuration descriptors for the module operating system, module level health monitoring and the AIDA logbook service.
Figure 10.17: DIANA PIM-PSM Mapping Process
10.6 Verification and Validation Support

As already highlighted by DO-178B (see in Section 9.3.4) the two most problematic questions with the use of MDE based techniques in the context of verification are: (i) the ability to maintain traceability between the models capturing requirements and the generated artifacts and (ii) how generated and handwritten code segments can be verified before the integration phase to be able to provide early low-level requirement coverage.

In the current section, we aim to tackle these questions in the context of the ARINC 653 configuration generation by (i) introducing a model-level traceability schema that is in line with the certification consideration defined in DO-178B (and C) and (ii) applying a design-by-contract [Mey92] approach to guard the different design steps of the development process; thus allowing better feedback and error detection early in the development process.

- **Traceability**: As stated by DO-178B (see in Section 9.3.4), traceability up to the system requirements must be maintained in order to demonstrate that all requirements are accounted in the final developed system.

  In Section 10.6.1 first, we demonstrate how we aligned our modeling artifacts to the DO-178B certification artifacts, then introduce the separate models used to capture both the inter-model and intra-model traceability links and finally, demonstrate how this approach can be integrated into an already used development chain for VxWorks 653 configuration development.

- **Design Candidate V&V**: To support early candidate verification and validation of design decision, we applied the concept of contracts to guard the input and output interaction points between the different design steps. This tight synchronization with the development process allows better error localization and early detection.

  How these contracts are defined and how they interact with the steps of the development workflow is described in Section 10.6.2.

10.6.1 Traceability

In order to show that our proposed approach is in-line with DO-178B, we need to demonstrate that it fulfills the three specific objectives regarding traceability between requirements and source code (see in Figure 10.18):

- **Objective A-3.6 - High-Level Requirements are Traceable to System Requirements**, where the purpose of this objective is

  ... is to ensure that the functional, performance, and safety-related requirements of the system are allocated to software were developed into the software high-level requirements

- **Objective A-4.6 - Low-Level Requirements are Traceable to High-Level Requirements**, where the purpose of this objective is

  ... is to ensure that the high-level and the derived requirements were developed into the low-level level requirements
Objective A-5.5 - Source Code is Traceable to Low-Level Requirements, where the purpose of this objective

... is to ensure that the software low-level requirements were developed into the Source Code

Traceability Model

Based on our proposed mapping for the DO-178B artifacts (see in Section 10.4.4), we designed the traceability links between the various elements to be an integral part of the various models. This allowed to have a relatively small specific traceability model that has direct links to specific parts of the modeling and configuration elements. A high-level overview – based on an application instance to partition allocation example – on how traceability is implemented between the various artifacts is depicted in Figure 10.19. For easier readability the references between model elements are depicted by simple arrows without type information.

From a design viewpoint, we separated the definition of traceability links from the System Requirements to the generated XML configuration tables into three levels:

System Req. to HRL. For this part, we applied a simple Key-Key map between the unique key of the System Requirements and its corresponding PIADL or PDM model elements. These map elements are saved next to the System Requirement descriptions and can be serialized into simple traceability matrices. Within the DIANA project these traceability were maintained by hand and we did not implement any support. For easier readability these links on Figure 10.19 are depicted by simple arrows between the Req1 requirement and the Zone_Controller application and its Zone_ControllerPar partition.
10.6. VERIFICATION AND VALIDATION SUPPORT

However, it is important to mention that there is already available off-the-shell integration [Sod] between DOORS – one of the most widely used requirements management platform – and Rational Software Architect – a widely used IBM modeling front end built atop of the Eclipse platform – that allows easy management of traceability between modeling artifacts and natural language written requirements.

**HLR to LLR** The main idea in the design of traceability between the HLR and LLR level is that the Integration System Model is considered as a prime artifact for maintaining trace information between the PIADL, PDM and the elements that are created in the Integrated System Model. This allows the traceability model to be small (compared to the ISM) and it only needs to hold references to the elements of the ISM. However, as a consequence the ISM needs be designed taking into account that

- It must hold all references (trace links) to the PIADL and PDM elements
- It should be structured to be in-line with the mapping process and especially the different design groups (e.g., Application group).
- For easier traceability, it should define separate modeling elements for each design step in the mapping process; thus easing the identification, which step created, which element.

Following these consideration many of the traceability links become integral part of the PIM and PSM models; thus generating the additional *Trace model* elements at each design step presents only a small development overhead for the mapping framework. However, we followed a conservative way how the traceability model was designed and defined separate trace elements for each design step. This allowed an easier identification of traceability information when searching for specific trace links within the model and ease the generation of traceability matrices if required.

Figure 10.19 shows, how the *AppAllocationElement* as the relative root element of the particular allocation of the *Zone_Controller* application to the *Zone_ControllerPar* partition holds direct (e.g., to the *Ins1* application instance) and indirect (e.g., to the *Zone_Controller* element through the *Ins1* element) references to each model element involved in the description of the allocation. Additionally, when the allocation is selected at the *Application Allocation* step its corresponding traceability element(s) in our example the *ApptoPartitionTrace* are automatically created and maintained through the mapping process.

**LLR to SC** Finally, for the XML configuration files we used simple XML path descriptions to pinpoint the corresponding part of the configuration files. In Figure 10.19 it is shown by the simple arrows from the *APTI ApptoPartitionTrace* element to the highlighted (with red) lines of the Target Platform configuration description.

Additionally, in case of simple text based VxWorks configuration files (e.g., usually referred as the MOS and POS files) the trace information is saved as the line number of the corresponding configuration element.

The links are automatically generated when the configuration files are synthesized from the Integrated System Model by the design steps in the Artifact generation group.
Using this approach, we achieved to have traceability from the System Requirements down to the generated XML configuration files. Additionally, as our choice for the RTOS was the VxWorks 653 from Wind River, we could even go further as Wind River has an XML-to-Binary compiler [Riv07a] for its configuration files certified to DO-178B Level A, enabling traceability down to the Executable Binary Code. Please note that in case a separate documentation is required to describe the traceability link by the certification authority it can be easily generated from the proposed approach as all information is available and it only requires a simple model traversal along the defined traceability links.

However, as already mentioned the current approach has some limitations that requires future work, more specifically:

- Our approach does not aim to fulfill the compliance objectives as defined in six separate objectives in DO-178B (also in DO-178C). However, we believe that even in its current form our approach is capable of fulfilling some of the goals defined in these objectives. For example, **Objective A-5.1 - Source Code Complies with Low-Level Requirements**, requires to demonstrate that the Source Code does not implement any undocumented function, which can be interpreted in our case that the configuration files do not hold any additional element that are not generated from the Integrated System Model. This statement holds for the Health Monitoring tables as they are completely generated from the model, but not for the module descriptor as scheduling of partitions are not supported by our approach.

- The trace information generation needs to be explicitly defined and implemented for each step
10.6. VERIFICATION AND VALIDATION SUPPORT

during the development of the various design steps. This includes the initial generation of the traceability links and their maintenance in case of any change initiated within the framework on any of the PIM or PSM models.

While there are techniques that do not require this additional effort as they support automated maintenance of traceability links [MGP09, GRVV10, SM07] and in [7]. It is important to mention that due to the strict tool qualification policy of DO-178B and its revisited version DO-178C (see in Section 9.3.4.2 and Section 9.3.5, respectively) these automated model-level traceability maintenance approaches could not be effectively applied due to their probably enormous qualification effort that is not in-balance with their provided advantages but provides a possible future research direction.

Finally, within the DIANA project traceability was implemented only for a selected subset (two-third of the PIADL model) of the complete modeling domain to exemplify how traceability within an MDE framework can be implemented. However, industry experts within the project all agreed that the concept is viable [40] and is a step forward compared to currently used approaches and may be approved by regulatory bodies.

10.6.2 Design Contracts for Design Steps by Graph Patterns

During a development process certain steps require external COTS tools (e.g., Matlab, SAL, etc.) or user interaction to perform their task. In order to guarantee that the result of these steps is acceptable and the process can continue, the definition of design contracts is an already well-known paradigm [Mey92]. The idea is to guard both the input and output of a step by specific constraints. Thus, a contract is composed of a precondition and a postcondition. In our interpretation a precondition defines constraints that need to be fulfilled by the input of the step in order to allow its execution, while the postcondition guarantees that the process can only continue (e.g., the step is considered completed) if its constraints are satisfied by the output.

In our approach we used graph patterns to define the design contracts [Baa05, AH08]

Example 25 To demonstrate how these design contracts can be formulated using graph patterns consider the simplified job allocation step of the Application Allocation category (See in Section 10.5.1).

In this step the task is to allocate an IMA system defined by its jobs and partitions over a pre-defined cabinet structure and to minimize the number of modules used (as the motivating example described in Section 8.2.1). An integrated modular avionics (IMA) system is composed of jobs (also referred as applications), Partitions, Modules and Cabinets. Jobs are the atomic software blocks of the system defined by their memory requirement. Based on their criticality level, jobs are separated into two sets: critical and simple (non-critical). For critical jobs, double or triple modular redundancy is applied while for simple ones only one instance is allowed. Partitions are complex software components composed of jobs with a predefined free memory space. Jobs can be allocated to the partition as long as they fit into its memory space. Modules are SW components capable of hosting partitions. Finally, Cabinets are HW storages for maximum (in our example) two modules used to physically distribute elements of the system. Additionally a certain number of safety related requirements will also have to be satisfied: (i) a partition can only host jobs of one criticality level and (ii) instances of a certain critical job can not be allocated to the same partition. An excerpt of the Platform Description metamodel describing the detailed IMA system is depicted in Figure 8.1.

Based on this metamodel we defined the pre- and the postcondition of this step as depicted in Figure 10.20 and Figure 10.21, respectively. All subconditions in the pre- and the postcondition are defined as positive and negative conditions depicted with + and - markings, respectively.
Precondition  For the definition of the precondition we rely only on that the model has at least one cabinet, one partition with its free memory defined and one job with an instance. These simple requirements are captured by the cabinet, partition and job graph patterns depicted in Figure 10.20.

Postcondition  The jobInstancewithoutPartition, partitionwithoutModule and modulewithoutCabinet subconditions describe that in a solution model each JobInstance, Partition and Module is allocated to a corresponding Partition, Module and Cabinet, respectively. For example, the jobInstancewithoutPartition subgoal captures its requirement using a double negation (NAC and negative constraint) stating that there is no unallocated job instance \( J_I \) in the solution model. As the declarative graph pattern formalism has an implicit existential quantification, nested (double) negation is required to quantify elements universally. Similar double negation is used in case of the other two subgoals.

The rest formulates safety and memory requirements. The partitionMemoryHigherThan0 pattern captures the simple memory constraint that all partitions must have higher than zero free memory. The safety requirement stating that a partition can only host jobs of one criticality level is captured by the partitionCriticalityLevelSimilar pattern. As it is a negative constraint it describes the (positive) case where the \( P_1 \) partition holds two job instances \( J_1 \) and \( J_2 \) of a simple and a critical job \( Job1 \) and \( Job2 \), respectively. The criticalInstanceonSamePartition and criticalInstanceonSameModule patterns restrict in a similar way that no job instances \( J_1 \) and \( J_2 \) of a critical job \( Job \) can be allocated to the same partition \( P_1 \) or module \( M_1 \).
Guarding the application allocation using this contract ensures that all applications are properly allocated and safety requirements are fulfilled by the output model. Allowing further steps to build on these ensured guarantees.

Note that these contracts can also be used for the automated allocation process developed for the DIANA PIM-PSM mapping editor using the ViATRA2 DSE framework as described in Section 8.2.1.

### Concepts of Design Contracts

The current section gives an overview on the concepts of design contracts defined by graph patterns as used in our PIM-PSM mapping process.

**Definition 27** A subcondition $SC$ is a graph pattern $GP$ capturing the structural condition and a validity criteria $VC$ defining when the subcondition is accepted. Formally, $SC = (GP, VC)$, where $VC$ is either one of from the following set \{positive, negative, cardinality($X$)\}

- A **negative** subcondition is satisfied by a model $M$ if it does not have a match in $m$. Formally, $\not\exists m : GP \rightarrow M$.
- A **positive** subcondition is satisfied if its representing graph pattern has a match in the model $M$. Formally, $\exists m : GP \rightarrow M$.
- Finally, a **cardinality($X$)** subcondition is satisfied iff their representing graph pattern has (at least) a predefined positive number $X$ of matches in the underlying model $M$. Formally, $|\{m | m : GP \rightarrow M\}| \geq X$.

**Definition 28** A Design Contract $Con$ is composed of a pre- $Pre$ and a postcondition $Post$, where both conditions are the conjunction of subconditions $SC$ (and a graph pattern is a disjunction of pattern bodies). Formally, $Con = (Pre = \wedge_{i \in I} SC_i, Post = \wedge_{j \in J} SC_j)$.

The $Pre$ and $Post$ conditions are satisfied iff all their subconditions are satisfied by the underlying model.

Finally, the execution of the development process is needed to be slightly modified to take into consideration of the pre- and postconditions of the contracts. A step within the development process can be in one of the following states:

- A step can be considered **completed** if its postcondition is satisfied. However, note that it may require to satisfy additional requirements that cannot be specified by graph patterns (e.g., max allowed weight for a cabinet that requires the calculation of all contained HW elements).
- A step is in **active** (or in-progress) state – meaning that its execution can be started or already in progress – iff (i) all of its prerequisite steps are **completed** and (ii) the precondition defined in its design contract is satisfied by the underlying model.
- Finally, a step is **upcoming** – can only be executed if all of its preceding steps will become completed – iff it is not **completed** or **active**.

It is important to see that this definition requires that the result of a design step is always available in the underlying model and all steps can query those parts of the underlying model that contains elements captured in their contracts.
Conclusion

Using the described design contract approach tightly synchronized to the development process on a step-by-step basis, we achieved a design time approach for early, candidate validation of design decisions. Based on the feedback from the project partners the advantages of our approach are the following:

- The use of *direct model-level validation* enabled earlier, design time error detection that prevented certain errors to propagate into later phases of the development process.

  Additionally, the tight integration of the design contracts to the development process – compared to a batch like model-based validation defined as a set of invariants – enabled better error localization that helped with the fixing of the error.

- Additionally, as a promising advantage it may decrease (re-)certification cost in case a change is introduced into the implementation of a certain development step. This can be achieved when the activity implemented within the step can be completely defined using a design contract. This way, whenever the realization of the step is changed within the development process (for example, automation is introduced to the step or the graphical interface changes), then the re-certification only needs to prove that the new implementation is conform to the contracts and thus the complete process does not need to be re-certificated.

- Finally, – one function provider project partner also mentioned that – certain *low-level requirements can be directly defined by contracts* on the model level; thus allowing easier approval of requirement coverage. This can be a huge advantage as it can ease traceability from the requirements to the designed system.

Although, the presented approach is a step forward for the application of design contracts defined as graph patterns for the model-driven validation of avionics configuration descriptors in its current form and realization it holds certain limitations that is subject to future work:

- *Lack of precise formal semantics for rich graph pattern languages*. For the realization of the automated validation of design contracts we used the EMF-INQUERY framework that provides a rich graph pattern language [BURV11] that is expressive enough to define practical contracts for complex steps. However, due to its expressiveness the precise formalization of its semantics is an ongoing work and until it is not completed it cannot be accepted as certification artifacts as stated in the DO-333 Formal Methods supplement for DO-178C.

  Additionally, we believe that after defining the precise formal semantics for EMF-INQUERY language may allow property preservation analysis along the complete workflow that could also provide certification evidences and thus reduce testing or manual validation.

- *Better error localization*. The step-by-step level definition of contracts helps to capture errors introduced into the model at the actual design step and gives a better error localization compared to batch like validation at the end of the design phase.

  However, in many cases complex design decisions are realized into the underlying model along several steps and errors detected by a contract of a single step may not be introduced solely in the step it is defined and thus may mislead the system architect when trying to fix the problem. One possible solution for solving this problem is the introduction of hierarchical contracts that would allow to define contracts for a set of steps or groups. This way contracts could be defined
10.7. MDE BASED IMPLEMENTATION OF THE DIANA PIM-PSM MAPPING PROCESS

on a higher abstraction level closer to the conceptual level of design decisions rather than the actual realization steps.

Note that it would be possible to define these contracts by OCL and transform a large subset directly to the graph pattern formalism [WTEK08b]. However, we selected to use graph patterns for two implementation specific reasons: (i) at the development of the DIANA tooling there were no OCL execution engine available capable of providing on-the-fly performance for constraint validation and (ii) as a requirement from the project partners, we tried to keep the number of used modeling tools low to simulate the case of a possible tool certification process.

10.7 MDE based Implementation of the DIANA PIM-PSM Mapping Process

The current section gives an overview on how the DIANA PIM-PSM mapping process were realized in the project with a special focus on how model-driven techniques were applied. First, we give an overview on the overall architecture of the DIANA tooling as realized over Eclipse framework in Section 10.7.1. Then in Section 10.7.3 we highlight the interesting parts of the DIANA tooling with respect to the application of model-driven techniques; this includes design space exploration for allocation generation (see in Section 10.7.3.1), on-the-fly design contract validation (see in Section 10.7.3.2) and automated configuration generation (see in Section 10.7.3.3).

Additionally, as certain model-driven techniques applied within the DIANA tooling are based on our EMF-INCQUERY framework we also give a short overview on its capabilities and usage in Section 10.7.2.

10.7.1 Overview on the Implementation of the DIANA PIM-PSM Mapping Process

A high-level overview of the DIANA tooling architecture is depicted in Figure 10.22. It mainly follows a model (data) centric approach as the different modules of the systems are mainly communicating through the central model store.

The developed editors and tools are all based on the Eclipse framework that is currently the de facto tool development platform on Java. The complete system is composed of the following major modules:

- As one of the most important feature all models within (except for the VIATRA2 DSE models) the DIANA tool chain are represented using EMF (Model Store). This includes the models described in Section 10.4 and also any auxiliary model that is temporarily used in the system for certain generation steps.

- As a key feature of the DIANA approach the system architect can import already defined Matlab Simulink systems as PIADL models. This import is implemented as a separate plugin for the tool that first serializes the Simulink representation to a direct EMF representation and from this representation a simple one-way transformation is done from the Simulink EMF representation to its PIADL instance model representation.

- The PIADL editor is an Eclipse GEF (Graphical Editing Framework) based graphical editor. It provides a full-featured editor with drag-and-drop capabilities for model creation and manipulation. Most of the steps in the mapping editor are small, user driven model manipulations as design decisions can rarely be fully automated in the avionics domain.
The original version of the editor was built in the DECOS [DECb] project and was later adapted to the specific needs of the DIANA project. An example model as visualized by the editor is depicted in Figure 10.4.

- The **PIM-PSM Mapping editor** is a custom plug-in built to execute the proposed DIANA development workflow (see in Section 10.5). It consists of two main parts: (i) a set of forms that define the design steps with all GUI elements and (ii) a simple workflow execution engine that checks the state conditions for the design steps and implements all integration interface between the various tools used within the system.

- The automated allocation of applications to partitions are based on the ViATRA2 DSE framework. As both the mapping editor and the ViATRA2 DSE framework are based on the Eclipse platform, we used the platforms extension mechanism for their integration. This way the mapping editor can directly translate the required part of the Integrated Model and the defined compatibility mapping into a CSP(M) problem.

- The contract validation module is based on the EMF-INQUERY framework (see in Section 10.7.2) that provides on-the-fly contract evaluation based on the incremental pattern matching mechanism described in Section 6.1.

- Finally, EMF-INQUERY is also used for the automated generation of configuration artifacts using model templates specified by graph patterns.

### 10.7.2 Overview on the EMF-INQUERY framework

The aim of the EMF-INQUERY [13] approach is to bring the benefits of graph pattern based declarative queries and incremental pattern matching to the EMF domain. The advantage of declarative query specification is that it achieves (efficient) pattern matching without time-consuming, manual
coding effort associated to ad-hoc model traversal. While EMF-INCQUERY is not the only technology for defining declarative queries over EMF (e.g. EMF Query or MDT-OCL), its distinctive feature is *incremental pattern matching*, with special performance characteristics suitable for scenarios such as on-the-fly well-formedness validation.

Additionally, some shortcomings of EMF are mitigated by the capabilities of EMF-INCQUERY, such as cheap enumeration of all instances of a certain type, regardless of where they are located in the resource tree. Another such use is the fast navigation of EReferences in the reverse direction, without having to augment the metamodel with an EOpposite (which is problematic if the metamodel is fixed, or beyond the control of the developer).

Note that this work is the main research contribution of Gábor Bergmann’s PhD thesis.

### 10.7.2.1 Usage

EMF-INCQUERY provides an interface for each declared pattern for (i) retrieving all matches of the pattern, or (ii) retrieving only a restricted set of matches, by binding (a-priori fixing) the value of one or more pattern elements (parameters).

In both cases, the query can be considered instantaneous, since the set of matches of the queried patterns (and certain subpatterns) are automatically cached, and remain available for immediate retrieval throughout the lifetime of the EMF ResourceSet. Even when the EMF model is modified, these caches are continuously and automatically kept up-to-date using the EMF Notification API. This maintenance happens without additional coding, and works regardless how the model was modified (graphical editor, programmatic manipulation, loading a new EMF resource, etc.).

EMF-INCQUERY achieves incremental pattern matching by adapting the RETE algorithm discussed in Section 6.1.

### 10.7.2.2 Architectural Overview of EMF-INCQUERY

Both the query language and the implementation of EMF-INCQUERY are adapted from the VIATRA2 [VB07] framework. However, the role of VIATRA2 is limited to the development phase, as the runtime module of EMF-INCQUERY is not dependent on it. Queries in EMF-INCQUERY can be defined by graph patterns in the transformation language of VIATRA2 (see in Section 3.1.1). A *generator component* can be invoked to translate them to the EMF-specific query form that serves as the input for the EMF-based Pattern Matcher Engine. The latter is responsible for evaluating queries over EMF ResourceSets, and is intended to be invoked from any Java program.

Graph patterns suitable for the EMF conversion have to refer to the metamodel elements of the relevant EMF format. Therefore VIATRA2 first needs to be aware of the EMF metamodel (the Ecore model), which can be ensured by importing it into VIATRA2’s (meta-)model representation, the VPM model space (see in Section 2.3.3). As an additional benefit, the development of graph pattern based queries can be eased by taking advantage of the VIATRA2 framework. The VIATRA2 transformation interpreter shares identical functional behavior with EMF-INCQUERY. Therefore VIATRA2 serves as a faithful prototyping environment for graph patterns, capable of experimenting on EMF instance models imported into its model space. As part of the debugging process in VIATRA2, EMF instance models conforming to the metamodel can be imported, and the patterns can be matched against their VPM representations. Patterns can also be embedded into simple transformation programs to observe induced behaviour. See Figure 10.23 for a graphical overview of the various artifacts, software modules and their relations.
Note that the tool chain relies on the pre 0.4 version of the EMF-INCQUERY framework developed in 2010. Its newest version (0.6.7 as of Q3 2012) have been developed from scratch and does not rely on the VIATRA2 framework. For more details about this newer version consult with the framework’s official webpage [fra12].

10.7.3 Application of Model-Driven Techniques in the DIANA Tool Chain

This section gives an overview how we successfully applied model transformation based technology (i.e., see ur research contribution of Part II. and III.) in various parts of the tool chain. Section 10.7.3.2 introduces a graph pattern based contract notation used to define conditions for steps. Section 10.7.3.3 highlights how graph transformation like approach is used for ARINC 653 configuration generation.

10.7.3.1 Automated SW Allocation based on VIATRA2 DSE

As mentioned in Section 10.5.1 we applied our VIATRA2 DSE framework for finding all possible allocation candidate for the application to partition mapping. How the general labeling rules and constraints were defined for the automation is discussed in Section 8.3.2. However, in order to achieve better performance on larger allocation problems we applied the following domain specific optimization hints:

- The concrete rules generated for the constraint problem are similar to the ones introduced in Section 8.3.2, however, additional constraints are generated into the labeling rules based on
the compatibility mapping between the applications and partitions to reduce the traversed state space. The idea is to generate for each application-partition pair specific labeling rules that only allow the allocation between the paired elements. We experienced that this optimization reduced the average execution time by more than 40%.

- For each application that can only be allocated to a single partition due to its compatibility mapping, we modify the starting state by allocating it to its defined partition. This way these applications are eliminated from the constraint problem.

**Example 26** An example labeling rule generated for the `SendFlowDemand` Job and the `I/OProcessing` partition is depicted in Listing 10.1. The rule itself is very similar to labeling rule `allocateJobInstance` depicted in Figure 8.7. The only difference lies in its check condition that restricts its usage except on the `SendFlowDemand` job and the `I/OProcessing` partition.

```java
@LabelingLiteral(executionmode='choose')
grule addJobToPartition_SendFlowDemand_2I/OProcessing(out Job) = {
  precondition pattern lhs(Job,P1,M1,M2) = {
    'Partition'(P1);
    'Job'(Job);
    'JobInstance'(JIns)
    'Memory'(M1); 'Memory'(M2);
    'Partition'. 'freeMemory'(PM,P1,M1);
    'Job'. 'memoryNeeded'(JM,Job,M2);
    'Job'. 'instances'(JI,Job,JIns)
  neg pattern noJob(Job) = {
    'Partition'(P2);
    'Job'(Job);
    'Job'. 'instances'(JIns);
    'Partition'. 'jobs'(PB2,P2,JIns);
  }
  check(name(JIns) == "SendFlowDemand_2" && name(P1)== "I/OProcessing"
    && toInteger(value(M2)) <= toInteger(value(M1)))
  }
  action
  { let PA = undefined, PJ = undefined in seq
    { new 'Partition'. 'jobs'(PA,P1,JIns));
      setValue(M1,toInteger(value(M1)).toInteger(value(M2)));
    }
  }
}
```

Listing 10.1: Labeling rule for allocating the `SendFlowDemand` Job on the `I/OProcessing` partition

### 10.7.3.2 On-the-fly Design Contract Validation

In order to have an almost instantaneous feedback on the satisfaction of the design contracts, we applied the EMF-INCQUERY framework for the evaluation of the subcontracts. However, to support the different satisfaction criteria of the design contracts (see in Section 10.6.2) we had to develop a custom module for the PIM-PSM mapping. This custom module is responsible for both (i) the evaluation of the contracts and (ii) the generation of feedbacks for the system architect.

Based on the experience gained during the DIANA project we realized that EMF-INCQUERY should support the automated generation of validation plugins for a majority of Eclipse based editors. We believe that declarative definition of validation rules can be defined in early stages of the development and provides a powerful mechanism for the validation of avionics models.

The latest 0.6.7 version of the EMF-INCQUERY framework supports the automated generation of such validation plugins. The system architect only needs to define the graph pattern for the validation rule – along with some auxiliary information – and EMF-INCQUERY is able to generate all
the necessary Java classes and error marking extensions (for the Eclipse Problem Views). Note that such a validation rule is only equivalent to a subcontract in our contract definition, although, support for contract definition is on the development plan for the 1.0 version of EMF-INCQUERY.

**Example 27** A sample validation rule defined using the newest version of the EMF-INCQUERY framework is depicted in Listing 10.2. The validation rule defines the \texttt{partitionCriticalityLevelSimilar} validation rule that checks that instances of critical and non-critical jobs are not allocated to the same partition (see in Figure 10.21).

The \texttt{@Constraint} annotation is used to mark a graph pattern as a validation rule. It contains four parameters: (i) the \texttt{location} defines the element to which the error marker will be attached in case the validation rule is violated, in our case it is the partition \texttt{P}, (ii) the \texttt{message} defines the textual error message that will be displayed. It can use the parameters from the graph pattern head and may refer to the EAttributes of the parameters (\$\texttt{Job1.name}\$), (iii) the \texttt{severity} is either error or warning as defined by Eclipse for its problem view, and finally, (iv) the user can define to which editor he/she would like to attach its validation rule using the \texttt{targetEditorID} parameter.

The definition of graph patterns is similar as in the VIATRA2 transformation language, the only difference is that in EMF EReferences between EObjects cannot be directly referred to and thus a reference between two EObjects is defined as a pair, where the first parameter is the source object while the second one is the target. For example, \texttt{Job.instances(Job1,J1)} represent that there is an \texttt{instances} EReference from the \texttt{Job1} EObject (that is of type \texttt{Job}) to the \texttt{J1} EObject. The pattern definition language of the EMF-INCQUERY framework is detailed \cite{BURV11}.

```java
@Constraint{
    location = "P"; // location of the error marker
    message = "The $\texttt{Job1.name}\$ critical and the $\texttt{Job2.name}\$ non-critical applications are allocated to the same $\texttt{P.name}\$ partition." //error message
    severity = "error" // severity of the validation rule
    targetEditorID = "hu.bme.mit.diana.pimpsmeditor" //the editor to which the validation rule is attached
}

pattern partitionCriticalityLevelSimilar(P : Partition,
                                      Job1 : CriticalJob, Job2 : SimpleJob) = {
    'CriticalJob'('Job1');
    'SimpleJob'('Job2');
    'JobInstance'('J1');
    'JobInstance'('J2');
    'Partition'('P');

    'Job'.'instances'('Job1,J1') //Cannot define a direct reference to an EReference
    'Job'.'instances'('Job2,J2')
    'Partition'.jobs('P,J1');
    'Partition'.jobs('P,J2');
}
```

Listing 10.2: Validation rule for

### 10.7.3.3 Configuration Generation by Graph Transformation

Model transformation based automatic code or configuration generation is one of the main driving forces \cite{HKV08, EEHT05} of model-driven system development. It offers many advantages including the rapid development of high quality code, reduced number of errors injected during development and the consistency between the design and the code is retained, in comparison with a purely manual approach.

An example ARINC653 configuration snippet is depicted in Figure 10.24. It captures the details of the flight management non-system partition, which has the highest \texttt{Level A} criticality as defined
in [RTCe], one queueing and four sampling ports and separate memory blocks for code and data. A port is defined with its direction, maximum message size and name, where the sampling and the queueing ports have additionally refresh rate or maximum number of messages parameters, respectively. Finally, a memory block is defined by its access mode (e.g., read or write), type (code or data) and size.

![Figure 10.24: Example ARINC653 descriptor](image)

To generate the required XML format we based our code generator on the EMF-INQUERY framework, where configuration file templates can be defined using a special kind of graph transformation rules called template rules. These template rules define both the GT pattern that matches to the required elements in the model and generate code as a side effect. Note that these GT rules do not modify the underlying model during their execution.

**Example 28** A fraction of the code generator responsible for the generation of the Partition_Memory XML subtree is depicted in Figure 10.25 and Listing 10.3.

The `partitionMemory` GT rule defines the template for the `Memory_Requirements` XML element. Its pattern matches to the partition `P` that has a memory `M` as its memory block. A memory has three attribute defined as `memorySize MS`, `memoryType MT` and `memoryAccess MA`. The `print` block defines the template that is printed out when the rule is applied. All three parameters value are retrieved using the `value` keyword.

The `partitionID` is an auxiliary pattern used to get the ID `PID` of the partition `P`.

![Figure 10.25: Example GT patterns used for configuration generation](image)

Although, for the DIANA project we used Java as the control language for the query executions, in the definition of the code generators we followed the ASM control structures as provided in ViATRA2.
For this reason, we demonstrate the control structure of the example using ASM. The example code shown in Listing 10.3 demonstrates how we defined our code generator using the partitionMemory template rule and the partitionID pattern. The outer forall rule is used to find all partitions P with their id PID in the model as defined by the partitionID pattern, and then execute its inner sequence on all matches separately. For each partition separate Partition_Memory XML elements are emitted out with their additional PartitionIdentifier and PartitionName parameters. As for the Memory_Requirements XML elements a forall rule invoking the partitionMemory GT rule is defined. The rule is invoked for all memory blocks M of partition P, where P is (at that point) already bound to a concrete partition by the outer forall. The whole code generator is built up using similar snippets and execution ideas.

```
... // memory block generation
forall P, PID with find partitionID(P, PID) do seq{
    println("<Partition_Memory PartitionIdentifier="+value(PID)
              "PartitionName="+name(P)+">");
    forall M with apply partitionMemory(P, M); // GT rule as template
    println("</Partition_Memory>");
}
...
```

Listing 10.3: Partition_Memory code generator snipplet

10.8 Related Work on MDE for Safety Critical Development

Relevant EU research projects There are numerous approaches in the literature introducing various model based techniques for the development of embedded system. Here we give a brief summary of some current and previous international and EU research projects with significant relevance to design and verification of embedded systems involving model-based techniques.

The clear separation of PIMs and PSMs, which specify tool integration processes with different levels of details were first introduced in the DECOS [DECb] project and as it follow-up generalized in the GENESYS [GEN] project, which proposed a cross-domain architecture for embedded systems, and a fitting model-driven engineering process for their development. As distinctive features, GENESYS supports (i) different modeling languages including UML and many of its profiles, (ii) a service-oriented development of subsystems, (iii) both uni- and bidirectional model transformations with manual, semi-automatic, automatic execution. Their direct continuation is the INDEXYS [IND] project that aims to realize industrial level implementation of the cross-domain architectural concepts developed in the GENESYS.

TopCased ("The Open source toolkit for Critical Systems") [TOP] is a software environment primarily dedicated to the realization of critical embedded systems including hardware and/or software. Topcased promotes model-driven engineering and formal methods as key technologies, such as a model bus-based architecture supporting standard modeling technologies such as EMF, AADL, UML-MARTE, and SysML. For model transformations, TopCased uses ATL [CCF+06].

The recent EU projects of ModelWare [Modb] and MODELPLEX [MODa] outline techniques that are based on similar approach. ModelWare aimed at defining and developing the complete infrastructure required for large-scale deployment of MDE strategies and validating it in several business domains. It can (i) provide transparent integration across model, tool, platform, machine boundaries; (ii) support the creation of distributed, multi-user tool chains; (iii) handle many metamodels and artifacts; (iv) integrate interactive and non-interactive tools; and (v) use different technologies for communication. ModelWare offers a process modeling framework, and a model bus for exchanging high-level data that are either Java-based or described by Web Services. On the other hand, it lacks
model transformation support, which has only been added in its successor MODELPLEX project. MODELPLEX has a SPEM2 based tool set for supporting the enactment and execution of processes and is integratable with workflow and project management tools as well.

COCONUT [COC] focuses on the definition of a formal framework [BCG+10] based on a tight integration of design and verification through refinement steps of an embedded platform design flow, from specifications to logic synthesis and software compilation.

INTERESTED [INTb] is an industry driven project aiming to define and implement a reference design and rapid prototyping platform for interoperable embedded systems. Its main focus is on the development of the tool-chain based on model-driven paradigms rather than research on novel application in the domain and will built on the results of the TOPCASE and other open source model-based development platforms.

CHESS [CHE] sought to improve model-driven development practices and technologies to (i) better address safety, reliability and robustness functionalities as required by the aeronautical and railway industry and (ii) develop techniques to guarantee the correctness of assembled component embedded systems by reusing certification artifacts of the components used for the complete system.

FRESCOR [FRE] aimed to integrate advanced flexible scheduling techniques directly into an embedded systems design methodology, covering all the levels involved in the implementation, from the OS primitives, through the middleware, up to the application level using contracts to define the application requirements.

CHARTER [CHA] focuses on cost-reduction of certification of critical embedded systems by integrating real-time Java, model-driven development, rule-based compilation, and formal verification into a novel development process called Quality-Embedded Development (QED).

**Model transformations in tool integration** In the followings, tool integration solutions with model transformation support are presented.

As our own model transformation framework, VIATRA2, is positioned as a dedicated model transformer, has been successfully applied both in scenarios where the abstraction gap (between source and target languages) was relatively small (such as code generation from MDA-style platform-specific models [GÁV06, GDV09, RVV09], or abstract-concrete syntax synchronization in domain-specific languages [RÖV10]), as well as mappings with strong abstractions (e.g., the generation of mathematical analysis models from design artifacts, for formal analysis purposes).

The IPSEN approach [KNS99] outlined probably the first integration related scenario, where model transformation techniques played a key role. The aim of IPSEN was to construct an integrated software development environment (SDE) tool, which helped capturing both context-free (i.e., syntactic) and context-sensitive (i.e., graph-based) aspects of languages by textual and graphical editors, respectively. The technique of graph transformation has been heavily used for the development of the tool especially for specifying constraints and translations in the context-sensitive domain.

ModelCVS [Eli06] employs (i) semantic technologies in forms of ontologies to partly automate the integration process, and (ii) QVT transformations, which are generated from these ontology descriptions. As distinctive features, ModelCVS uses Subversion for versioning, EMF and MOF-based metamodels for model representation, and a generic workflow ontology for defining processes. In contrast to our approach, ModelCVS prepares adapters for tools and not for models as these latters are stored in a central repository. Additionally, model transformations are used in ModelCVS for the synchronization of models, and not for the integration of different tools.

Giese et al. [GW09] applied triple graph grammars to achieve bidirectional model synchronization between overlapping models. As a unique feature, their approach facilitates an incremental solution
based on acyclic dependencies within the correspondence graphs, thus it can scale up to industrial size problems. The approach was applied to integrate SysML and AUTOSAR models [GNH10] in a consistent way during an interplaying design process, where SysML was responsible for the system level design, while AUTOSAR models captured the software related decisions.

From the model transformation point of view, a similar setup can be found in MOFLON [Fel08, Car08]. Transformations are again used for model synchronization, but in this case, they are defined by triple graph grammars. MOFLON operates on JMI and MOF 2.0 based models.

**Model queries for contract evaluation** OCL [Obj06] is a standardized navigation-based query language, applicable over a range of modeling formalisms. Taking advantage of the expressive features and wide-spread adoption of OCL, the project Eclipse OCL through its *Essential OCL* language provides a powerful query interface that evaluates OCL expressions over EMF models. Additionally, it also supports the definition of invariants and operations to enrich the Ecore metamodel using either the *Complete OCL* [Wil11] or the *OCLinEcore* [Ecl12] languages. Balsters [Bal03] presents an approach for defining database views in UML models as derived classes using OCL. The derived classes in this case are the result set of queries, which is similar to the match sets provided by EMF-INCQUERY.

There are several technologies for providing declarative model queries over EMF, e.g. EMF Model Query 2 [Ecla] and EMF Search [Eclb]. Other graph pattern based techniques like [BET08, GHS09] have been successfully applied in an EMF context.

Cabot et al. [CT09] present an algorithm for incremental runtime validation of OCL constraints and uses promising optimizations, however, it works only on boolean constraints. An interesting model validator over UML models [GRE10] incrementally re-evaluates constraint instances whenever they are affected, but relies on environments that support the recording of read-only access to the model, unlike EMF. Additionally, general-purpose model querying is not viable.

These approaches provide possible alternatives to implement the design contract validation approach. However, many of them lack incremental evaluation support or require more integration effort to enable their effective use for complex, multi resource based EMF models. Additionally, the compiled version of EMF-INCQUERY provides source code only for the defined queries – meaning a smaller code complexity for the implementation – and thus enables potential future certification.

### 10.9 Summary on MDE in Avionics Configuration Development

In the context of the DIANA EU FP6 project [DIA], we designed and participated in the development of a PIM-PSM mapping framework [14] for systematically designing standard ARINC 653 configuration tables. The framework is based on a platform independent architectural modeling language (PIM) [DECa] that allows the integration of industry leading architectural languages like AADL [SAEb] or Matlab Simulink. The precise low-level details of a specific configuration for the ARINC 653 platform are captured by a Platform Specific Model (PSM). Mapping the PIM to the PSM is handled by a complex interactive model transformation process that needs to bridge a large abstraction gap, where critical design decisions made by the system architect; thus it cannot be fully automated. Therefore, the mapping process is subdivided into well-defined design steps and precisely defined the contracts, interactions and interfaces of each step. Individual design steps are then organized into complex workflow-driven transformation chains, which are closely aligned with the designated development process followed by the airframer or function provider. Finally, configu-
ration tables for the standard ARINC 653 and VxWorks specific Module descriptions are generated based on the PSM models.

To support certification, end-to-end traceability links from the PIM to the synthesized configuration files are generated using both (i) inter-model traceability based on an integration model and (ii) model-to-configuration traceability with XMI files connecting generated configuration elements to their corresponding model elements. All traceability link follows the guidelines defined in DO-178B.

Finally, for early model-based validation, we adopted the technique of contract based design to the mapping process that is analogous to the strong separation of components as defined in DO-297 for IMA systems. Combined with our EMF-IncQuery framework we were able to provide on-the-fly validation results to the system architect; thus providing better error localization and containment.

The results of this chapter are formulated as thesis contributions as follows:

1. Design contracts defined by graph patterns for model-driven development steps in avionics.
   Following the concept of strong separation of components through precise interface descriptions as defined by DO-297 [RTCd], I elaborated a contract language [14, 6, 26] for model-driven development of avionics system configuration artifacts (see in Section 10.6.2).

2. On-the-fly validation of design contracts in avionics systems defined by graph patterns.
   Aligned with the recommendation as proposed in DO-178B for early error detection, I defined an on-the-fly design contract validation approach based on incremental pattern matching [14, 13] to support the model-driven development process for avionics system configuration artifacts (presented in Section 10.6.1).

3. End-to-end traceability in model-based design process for avionics configuration design.
   I proposed an approach to support end-to-end traceability [14, 23] from high-level architectural models to XML based artifacts following the certification requirements defined by DO-178B [26,38].

Within the DIANA EU FP6 research program [DIA] I designed and participated in the development of a complete mapping framework that applies model-driven engineering techniques for the systematic design of ARINC 653 configuration tables. The DIANA framework is built upon the foundations of systematic model-based design for critical embedded systems laid down by wide international collaboration in the DEcos project [DECb] with the Fault Tolerant Systems Research Group as key contributor, and used in the PhD thesis of András Balogh [Bal]. However, the target domain of the DEcos project was mainly time-triggered architecture for the automotive domain (AUTOSAR) that compared to the ARINC 653 platform and its DO-178B certification guidelines rise different design and certification requirements.

The mapping framework was developed in collaboration with Dénes Monostori, who was an MSc student under my supervision. Finally, the on-the-fly contract validation module is based on the EMF-IncQuery framework [13], which was developed as a cooperative work between the members of the Fault Tolerant Systems Research Group and OptXWare Ltd.

My primary contribution lies here in the adaptation of general modeling and model transformation techniques in the context of avionics systems for systematic configuration design for ARINC 653 architectures with support for the automated generation of certification artifact as required by DO-178B.
Part V

Conclusions and Appendix
Conclusion

As a final conclusion, my results described in the thesis are compared with the main objectives defined in Section 1.5. Additionally, I summarize the practical applications of my results and outline some future research directions and application domains.

11.1 Fulfillment of Objectives

Challenge 3: Combined graph pattern matching strategies

As contributions towards the performance optimization of graph pattern matching, I defined a hybrid pattern matching approach (Chapter 6) that is able to uniformly handle and apply different pattern matching strategies – within a transformation program – on graph pattern basis. The approach is based on the generalized search graph concept defined for uniformly representing graph pattern matching operations (Chapter 5). I verified the performance characteristics of the approach by benchmarks (Appendix C and B).

I integrated these results into the ViATRA2 framework that served as the underlying model transformation system for several research and industrial projects.

Challenge 2: Graph transformation based structural constraint solver for solving evolutionary design space exploration problems

I specified a novel approach for defining constraint satisfaction problems directly over models using graph transformation rules and graph patterns (Chapter 8). Additionally, I introduced two extensions to the original approach to support both flexible constraint specification and dynamic manipulation of constraints and labeling rules. Furthermore, I have also built a prototype solver implementation on top of the ViATRA2 model transformation framework (ViATRA2 DSE) using incremental pattern
matching for efficient constraint propagation. Finally, I verified the performance of the constraint solver system by systematic benchmarks and comparison with other solvers.

**Challenge 1: Model-driven systematic configuration design for civil avionics systems**

I adapted several model-driven engineering techniques for the systematic design of civil avionics configuration artifact (Chapter 10). Additionally, I proposed an end-to-end traceability approach from high-level models down to the generated artifacts using both inter-model and intra-model trace links. Finally, for model-based validation and early error detection analogously to the design-by-contract methodology, I adapted a graph pattern based contract language to specify the input and output requirements on the steps of the development process.

The approach has been validated by leading the members of the DIANA consortium, leading avionics system providers and aircraft manufacturer and they provided feedback for further improvements especially on tool certification issues.

### 11.2 Applications of new Results

In order to demonstrate the practical relevance of the approaches and methods outlined in the current thesis the current section highlights the applications of the result of my thesis.

#### 11.2.1 Pattern Matching Algorithms in VIATRA2

The results of Thesis 1 have been provided the theoretical basis for implementing the local-search based graph pattern matching engine of the VIATRA2 framework. Additionally, they have also served as the backbone of the graph transformation module, which have utilized the search graph to generate the appropriate manipulation operations. Moreover, the hybrid pattern matching strategy has been integrated into the VIATRA2 framework and has provided a fine-grained optimization capability between memory consumption and runtime performance for transformation designers. All modules are part of the current official release of the open source VIATRA2 framework hosted by the Eclipse Foundation.

As a key part of VIATRA2 it has been applied in many research projects from various tool integration tasks (DECOS FP6 [DECb], DIANA FP6 [DIA], MOGENTES FP7 [MOG] EU projects) to early model-based verification and validation (HIDENETS FP6 EU project) and source code and configuration generation (SENSORIA FP6 [SEN], E-Freight [E-F] FP7 EU projects).

#### 11.2.2 VIATRA2 Design Space Exploration framework

The results of Thesis 2 have been implemented in the VIATRA2 DESIGN SPACE EXPLORATION (VIATRA2 DSE) framework, an add-on to the VIATRA2 DSE release. Its implementation has been evaluated and compared with several state-of-the-art constraint solvers like KORAT [BKM02], GROOVE [Ren04a] and the industry leading SICStus Prolog [Theb] CLP(FD) library. As a result, the VIATRA2 DSE framework has provided comparable results and especially in case of dynamic problems outperformed all other approaches.

Moreover, it has been effectively used in the DIANA project for the allocation of safety-critical software components over airborne ready ARINC 653 compatible real-time operating system.
11.3. FUTURE RESEARCH DIRECTION

Finally, it serves as the basis for a follow-up PhD research by Ábel Hegedüs, who investigates further guided traversal optimization algorithms based on rule dependency analysis [3] and various selection criteria [12].

The framework is available from the official ViATRA2 site at http://viatra.inf.mit.bme.hu

11.2.3 Model-Driven Development of Integrated Modular Avionics Systems

In the context of the DIANA EU FP6 project, I participated in the development of a complete model-driven mapping framework from high-level platform independent models to configuration artifacts for the underlying ARINC 653 RTOS. I was responsible for the design and development of the complete mapping process carried out by the framework consisting more than 25 separate design steps. I also developed the necessary integration module to the ViATRA2 DSE solver to support the mapping of avionics software payload to the underlying implementation platform consisting of partitions and modules.

The developed framework has been evaluated by leading industry partners like Embraer the 3rd largest civil airframer on the world, the Dutch National Aerospace Laboratory and GMV Aeronautics the largest avionics company in Portugal. Based on their feedback we fine tuned the implementation and introduced the results (i) at the 2008 Farnborough Air Show as part of the DIANA tutorial on future 3rd generation IMA platform [35] and (ii) in a joint publication with GMV at a premier industrial avionics conference: the 29th IEEE/AIAA Digital Avionics Systems Conference [14].

Another major follow-up of our approach is that Embraer initiated a cooperative research project with our group on a related topic.

11.2.4 EMF-INQUERY

In order to apply our technology to a broader industrial domain, incremental pattern matching technology has been adapted to EMF, one of the most widely used modeling environment as of today. EMF-INQUERY [13,20,9,7] provides an effective query API for EMF models with additional support for automated validation and change analysis.

Apart from its application for the validation of design constraints Section 10.7.2. A collaborative work with Ábel Hegedüs and Tamás Szabó (an MSc student partly under my supervision) has started to adapt the ViATRA2 DSE framework to EMF using EMF-INQUERY.

The EMF-INQUERY framework is a major research contribution of Gábor Bergmann’s PhD thesis and lead by István Ráth a colleague of mine.

11.3 Future Research Direction

Combining Structural and Finite-Domain Constraint Solvers

As it have been highlighted in Chapter 8, constraint satisfaction programming can be adapted to graph based models by using a combination of graph patterns and transformation rules. Additionally, model-driven techniques like incremental pattern matching provides an effective way to solve structural constraint defined using our proposed approach. Although the current approach allows the definition of simple constraints over the attribute values of the model but it does not provide any mean – other than simple checks on the result candidates – to use them for finding a solution.

Current research in our group is focused on combining structural and finite domain constraint solvers to provide means and techniques to (i) represent states in the state space in a unified way
allowing the combination of structural and finite domain constraints for the definition of the problem and (ii) provide a solver architecture that can handle such combined constraint set.

Adaptive Hybrid Pattern Matching

The hybrid pattern matching approach described in Chapter 6 provides a mean to combine different pattern matching strategies. However, we believe that our current approach is only the first step to take full potential of hybrid pattern matching as in its current form it lacks the support for (semi) automated adaptive switch between matching strategies based on the underlying model and the transformation program. To overcome this limitation we aim to adapt well-known approaches to aid the transformation designer for code optimization and better adaptive strategy switching.

- **Pattern analysis** can be used to classify graph patterns according to complexity, size, and complex cost metrics statically.

- **Program analysis** aims to identify patterns and model manipulation steps that are frequently used, rarely used, or unused for a period of time by analyzing the transformation program, without actually running it (see in [33,34]).

- **Trace analysis** improves this knowledge of transformation behaviour by actually running the program on one or more provided typical models and gathering statistics on the type and amount of executed pattern queries and model manipulations (see in [10]).

- **Quantitative model analysis** a highly promising approach to estimate the match set cardinality of graph patterns based on statistics of the model (without actually running the pattern matching algorithm).

Additionally, we plan to investigate ways to achieve tighter integration between our two pattern matching engines. The aim is to provide means to allow different strategies to be responsible for matching different subpatterns within the same pattern without explicit user interaction.

Graph Pattern based Contracts as Tool Certification Artifacts

We believe that graph pattern based contracts can be used not only for early model-level error detection (as described in Chapter 10) but also as a mean to define precise interfacing between tools within a development process (as the original concept of design by contract [Mey92]). This way the implementation of the different steps can be changed within a development process without complete recertification analogously for software components in an IMA system as defined in DO-297 (see in Section 9.3.6).

Additionally, we also believe that contracts can be considered as auxiliary documentation for the provided functionalities of a certain step in the development process and used as certification artifacts as required in the Software Tool Qualification (DO-331) supplement of DO-178C.

However, we also know that in order to be able to reason on tool certification issues we need feedback and support from industrial partners or certification authorities.
The Complete Antworld Case study in VIATRA2

A.1 Antworld Metamodel

```plaintext
entity (AntWorld) {
entity (metamodel) {
  entity (food); supertypeOf (datatypes.'Integer', food);
  entity (foodBite);
  entity (ant);
  entity (carrierAnt) {
    relation (carries, ant, foodBite);
  } supertypeOf (ant, carrierAnt);
  entity (searcherAnt); supertypeOf (ant, searcherAnt);
  entity (pheromone); supertypeOf (datatypes.'Integer', pheromone);
  entity (field) {
    relation (hasAnt, field, ant);
    relation (hasCarrierAnt, field, carrierAnt); supertypeOf (hasAnt, hasCarrierAnt);
    relation (hasSearcherAnt, field, searcherAnt); supertypeOf (hasAnt, hasSearcherAnt);
    relation (hasFood, field, food);
    relation (hasPheromone, field, pheromone);
    relation (path, field, field); supertypeOf (path, circlePath);
    relation (returnPath, field, field); supertypeOf (path, returnPath);
  }
  entity (verticalSideField) {
    supertypeOf (field, verticalSideField);
    relation (verticalReturnPath, verticalSideField, field);
    supertypeOf (field, returnPath, verticalReturnPath);
  }
  entity (horizontalSideField) {
    supertypeOf (field, horizontalSideField);
  }
}
```
APPENDIX A. THE COMPLETE ANTWorld CASE STUDY IN VIATRA2

Listing A.1: VIATRA2 representation of the AntWorld metamodel

A.2 Antworld Simulation Transformation Program

```java
@incremental
import AntWorld;

machine antMachine_GT{

    asmfunction foodCounter/0;
    asmfunction foodTotal/0;
    asmfunction circlesTotal/0;
    asmfunction antsTotal/0;

    // >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    // @local search
    pattern along_ReturnPath(OuterNeighbor, InnerNeighbor) = {
        field(InnerNeighbor);
        field(OuterNeighbor);
        field.returnPath(ARP, OuterNeighbor, InnerNeighbor);
    }
    pattern boundaryBreached() = {
        field(Field);
        field.hasSearcherAnt(HasAnt, Field, Ant);
        searcherAnt(Ant);
        meg find alongReturnPath(OuterNeighbor, Field);
    }
    pattern boundaryVerticalSideField(VerticalSideField) = {
        verticalSideField(VerticalSideField);
        meg find alongReturnPath(OuterNeighbor, VerticalSideField);
    }
    pattern boundaryHorizontalSideField(HorizontalSideField) = {
        horizontalSideField(HorizontalSideField);
        meg find alongReturnPath(OuterNeighbor, HorizontalSideField);
    }
    pattern boundaryCornerField(CornerField) = {
        cornerField(CornerField);
        meg find alongReturnPath(OuterNeighbor, CornerField);
    }

    // >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    // @local search
    pattern circled(Field1, Field2) = {
        field(Field1);
        field(Field2);
        field.circlePath(CP, Field1, Field2);
    }
```
pattern missingCircleLink(Field1, Field2) = 
  find circled(InnerField1, InnerField2);
  field(InnerField1);
  field(InnerField2);
  verticalSideField(Field1);
  verticalSideField.Field1.verticalReturnPath(VRP1, Field1, InnerField1);
  verticalSideField(Field2);
  verticalSideField.Field2.verticalReturnPath(VRP2, Field2, InnerField2);
  neg find circled(Field1, Field2);
} or {
  find circled(InnerField1, InnerField2);
  field(InnerField1);
  field(InnerField2);
  horizontalSideField(Field1);
  horizontalSideField.Field1.horizontalReturnPath(HRP1, Field1, InnerField1);
  horizontalSideField(Field2);
  horizontalSideField.Field2.horizontalReturnPath(HRP2, Field2, InnerField2);
  neg find circled(Field1, Field2);
}

grule expandVerticalSide(out VerticalSideField) = {
  precondition find boundaryVerticalSideField(VerticalSideField)
  expandedVertical(VerticalSideField, OuterNeighbor) = {
    verticalSideField(VerticalSideField);
    verticalSideField(OuterNeighbor) in 'ants', 'model';
    verticalSideField.verticalReturnPath(VRP, OuterNeighbor, VerticalSideField);
  }
  action { call newField(OuterNeighbor); }
}

grule expandHorizontalSide(out HorizontalSideField) = {
  precondition find boundaryHorizontalSideField(HorizontalSideField)
  expandedHorizontal(HorizontalSideField, OuterNeighbor) = {
    horizontalSideField(HorizontalSideField);
    horizontalSideField(OuterNeighbor) in 'ants', 'model';
    horizontalSideField.horizontalReturnPath(HRP, OuterNeighbor, HorizontalSideField);
  }
  action { call newField(OuterNeighbor); }
}

grule expandCorner(out CornerField) = {
  precondition find boundaryCornerField(CornerField)
  expandedCorner(CornerField, HNeighbor, VNeighbor, CNeighbor) = {
    horizontalSideField(HNeighbor) in 'ants', 'model';
    horizontalSideField.horizontalReturnPath(HRP, HNeighbor, CornerField);
    verticalSideField(VNeighbor) in 'ants', 'model';
    verticalSideField.verticalReturnPath(VRP, VNeighbor, CornerField);
    cornerField(CNeighbor) in 'ants', 'model';
    cornerField.axisReturnPath(ARP, CNeighbor, CornerField);
    field.circlePath(CPH, HNeighbor, CNeighbor);
    field.circlePath(CPV, CNeighbor, VNeighbor);
  }
  action { call newField(HNeighbor);
    call newField(VNeighbor);
    call newField(CNeighbor); }
}

grule completeCircle(out Field1, out Field2) = {
  precondition find missingCircleLink(Field1, Field2)
  postcondition find circled(Field1, Field2)
}

rule newField(in Field) =
if (foodCounter() < 9) update foodCounter() = foodCounter() + 1;
else let Food = undef, HF = undef in seq {
    update foodCounter() = 0;
    update foodTotal() = foodTotal() + 1;
    new (food(Food) in Field);
    new (field.hasFood(HF, Field, Food));
    setValue(Food, 100);
}

rule growGrid() = seq {
    update circlesTotal() = circlesTotal() + 1;
    forall V with apply expandVerticalSide(V);
    forall H with apply expandHorizontalSide(H);
    forall C with apply expandCorner(C);
    for all F1, F2 with apply completeCircle(F1, F2);
    // println(""!! G rowing the grid "");
}

// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
// ANT ACTIONS
// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
pattern carrier(Ant, FoodBite) = {
    carrierAnt(Ant);
    ant.carries(CR, Ant, FoodBite);
    foodBite(FoodBite);
}

pattern searcher(Ant) = {
    searcherAnt(Ant);
}

pattern hasSearcherAnt(HA, Field, Ant) = {
    searcherAnt(Ant);
    field.hasSearcherAnt(HA, Field, Ant);
    field(Field);
}

pattern foodAvailable(Field, Food) = {
    field(Field);
    field.hasFood(HF, Field, Food);
    food(Food);
}

gRule grab(out Ant) = {
    precondition pattern canGrab(Ant, Food) = {
        find hasSearcherAnt(HA, Field, Ant);
        find foodAvailable(Field, Food);
    }
    postcondition find carrier(Ant, FoodBite)
    action {
        let Rest = tsInteger(value(Food)) - 1 in
        if (Rest > 0) setValue(Food, Rest);
        else delete(Food);
    }
}

gRule deposit(out Ant) = {
    precondition pattern canDeposit(Ant, CR, FoodBite, Hill) = {
        foodBite(FoodBite);
        ant.carries(CR, Ant, FoodBite);
        antHill(Hill);
        carrierAnt(Ant);
        field.hasCarrierAnt(HA, Hill, Ant);
    }
    /* CR to be deleted */
    postcondition pattern deposited(Ant, CR, FoodBite, Hill) = {
        searcherAnt(Ant);
        foodBite(FoodBite);
        antHill.piledUp(PU, Hill, FoodBite);
        antHill(Hill);
A.2. ANT WORLD SIMULATION TRANSFORMATION PROGRAM

    //pheromones to be applied on Field1 afterwards
    gtrule return (out Ant, out Field1) = {
        precondition pattern isReturning(Ant, Field1, HA1, Field2) = {
            field(Field1);
            field.returnPath(RP, Field1, Field2);
            field(Field2);
            field.hasCarrierAnt(HA1, Field1, Ant);
            find carrier(Ant, FoodBite);
        }
        /* oldHasAnt to be deleted */
        postcondition pattern rhs(Ant, oldHasAnt, Field2) = {
            field(NewField);
            field.hasCarrierAnt(NewHasAnt, NewField, Ant);
            carrierAnt(Ant);
        }
    }

    pattern hasPheromone(Field, Pheromone) = {
        field(Field);
        field.hasPheromone(HF, Field, Pheromone);
        pheromone(Pheromone);
    }

    rule leavePheromone(in Field) =
        try choose Pheromone with find hasPheromone(Field, Pheromone) do
            setValue(Pheromone, 1024 + toInteger(value(Pheromone)));
        else let Pheromone = undefined, HF = undefined in seq {
            new (pheromone(Pheromone) in Field);
            new (field.hasPheromone(HF, Field, Pheromone));
            setValue (Pheromone, 1024);
            /* println("Pheromone left at: ": name(Field));
        }

@Random
@locaSearch
    gtrule moveTowardsAttractingPheromone(in Ant, inout oldHasAnt, in Field1, out Field2) = {
        precondition pattern attractingOuterNeighbor(Field1, Field2) = {
            field(Field1);
            find hasPheromone(Field2, Pheromone);
            check(toInteger(value(Pheromone)) > 9);
        }
        /* oldHasAnt to be deleted */
        postcondition pattern rhs(Ant, oldHasAnt, Field2) = {
            field(Field2);
            field.hasSearcherAnt(NewHasAnt, Field2, Ant);
            searcherAnt(Ant);
        }
    }

    pattern home(Field) = {
        antHill(Field);
    }

@Random
@locaSearch
    gtrule moveAnywhereButHome(in Ant, inout oldHasAnt, in Field1, out Field2) = {
        precondition pattern anyNeighborButHome(Field1, Field2) = {
            field(Field1);
            field(Field2);
            field.path(P, Field1, Field2);
            neg find home(Field2);
        } or {
            field(Field1);
            field(Field2);
            field.path(P, Field2, Field1); // reverse direction
            neg find home(Field2);
        }
    }
/* OldHasAnt to be deleted */
postcondition pattern rhs(Ant, oldHasAnt, Field2) = {
    field(Field2);
    field.hasSearcherAnt(NewHasAnt, Field2, Ant);
    searcherAnt(Ant);
}
}

rule search(in Ant) =
    choose Field1, HA1 with find hasAnt(HA1, Field1, Ant) do
    try choose /random*/ Field2 with apply
    moveTowardsAttractingPheromone(Ant, HA1, Field1, Field2);
else
    choose /random*/ Field2 with
    apply moveAnywhereButHome(Ant, HA1, Field1, Field2);

// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
// WORLD MANAGEMENT
// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

pattern pheromone(P) = {
    pheromone(P);
}

rule evaporate(in Pheromone) =
    let Rest = (19*tointeger(value(Pheromone)))/20 in
    if (Rest > 0) setValue(Pheromone, Rest); else delete(Pheromone);

gtrule consume(out Hill) = {
    precondition pattern delivered(FoodBite, Hill) = {
        foodBite(FoodBite);
        antHill(Hill);
        antHill.piledUp(PU, Hill, FoodBite);
    }
    postcondition pattern consumed(FoodBite, Hill) = {
        antHill(Hill);
        searcherAnt(Ant) in Hill;
        field.hasSearcherAnt(HA, Hill, Ant);
        // delete(FoodBite); automatic
    }
    action {update antsTotal() = antsTotal() + 1; } // WEIRD
}

// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
// MAIN
// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

rule doRound(in RoundCounter, in StartTime, inout LastTime) = seq {
    iterate choose Ant with apply
    grab(Ant) do println("Ant grabs: " + name(Ant));
    forall Ant with apply
    deposit(Ant) do println("Ant deposits: " + name(Ant));
    forall Ant, FromField with
    apply return(Ant, FromField) do call leavePheromone(FromField);
    forall Ant with find searcher(Ant) do
        call search(Ant); // two kinds of search
    forall Pheromone with find pheromone(Pheromone) do
call evaporate(Pheromone);
   forall Hill with apply consume(Hill);
}

if (find boundaryBreached()) call growGrid();

let CurrentTime = systime() in seq {
    println("Finished round: " + RoundCounter);
    println("Elapsed time: " + (CurrentTime - StartTime));
    println("Total elapsed time: " + (CurrentTime - StartTime));
    println("# of circles: " + circlesTotal());
A.2. ANTWORLD SIMULATION TRANSFORMATION PROGRAM

Listing A.2: ViATRA2 source code for an iteration
Complexity Analysis of the AntWorld Case Study

In order to determine the polynomial order more precisely, we conducted the following analysis. In the followings, we follow the Landau notation [Don97] to describe asymptotical limiting behaviour of characteristic functions.

First, we split cumulative execution time into the cumulative time required to simulate the behaviour of the ants, the cumulative time required to grow the grid, and the cumulative time consumed by dropping and evaporating pheromone traces. The dropping of pheromone is included in the pheromone time, not in the ant management time. Formally,

\[ \text{Time} = \text{Time}_{\text{Ants}} + \text{Time}_{\text{Area}} + \text{Time}_{\text{Pheromones}} \]  

(B.1)

where the cumulative time spent on building the grid, \( \text{Time}_{\text{Area}} \), should be intuitively proportional to the grid size with any efficient implementation:

\[ \text{Time}_{\text{Area}} \sim \text{Area} \]  

(B.2)

The lower bound of the time consumed by pheromone management is approximated by the total number of times pheromones were dropped. This is also an upper bound of the total pheromone management time with an appropriate constant coefficient, because if pheromone is left on a new field, its evaporation will have to be simulated once each round, and there will be a constant number of rounds before it vanishes. Even if pheromone is dropped on the same field multiple times\(^1\), this evaporation cost will be sub-additive, as the pheromone trace containing the combined amount

\(^1\)this phenomenon is actually very common, as several thousand ants may retrieve food along the same path; our experiments suggest that the number of individual fields with pheromone traces tends to stay relatively low
APPENDIX B. COMPLEXITY ANALYSIS OF THE ANTWorld CASE STUDY

will still evaporate only once per round, and the exponential decay lends it a sub-additive lifespan. Consequently,

\[ Time_{\text{pheromone}} \in \Theta(\text{PheromoneDroppings}) \] (B.3)

In our experiments, we plotted the ant population increase against the size of the population in Figure B.1. The plot shows an approximately square root-type upper bound for the increase, in harmony with the following theoretical considerations. In order to give birth to the \( n \)th ant (excluding the initial 8), the colony needed to gather \( n \) food units. As food is distributed proportionally to the grid area, it follows that the discovered area defines an upper bound to the number of ants:

\[ \text{Ants} \in O(\text{Area}) \] (B.4)

The area is a quadratic function of the radius of the map, therefore the \( n \)th food unit, giving birth to the \( n \)th ant, needs to be delivered from a distance of at least \( \sqrt{n} \) with some constant multiplier (let us neglect the fact that the order of food units may actually vary). Pheromones are dropped on each step, therefore the spawning the \( n \)th ant involves at least \( \sqrt{n} \) pheromone droppings; formally, this can be expressed as follows:

\[ \frac{\delta \text{PheromoneDroppings}}{\delta \text{Ants}} \in \Omega(\text{Ants}^{0.5}) \] (B.5)

By integrating with respect to \( \delta \text{Ants} \), we obtain the following:

\[ \text{PheromoneDroppings} \in \Omega(\text{Ants}^{1.5}) \] (B.6)

As previously established, the birth of the \( n \)th ant requires retrieving food along a path having a length of at least \( \sqrt{n} \); since at most \( n \) ants are distributed along this retrieval path, and each ant can make one step each round, the birth rate per round can be approximated by an upper bound of \( \sqrt{n} \) (Figure B.1). This observation can be formalized as follows:

\[ \frac{\delta \text{Ants}}{\delta \text{Rounds}} \in O(\text{Ants}^{0.5}) \] (B.7)
Thus, we are looking for the expression for the cumulative time spent for ant management as the function of the size of the ant population. Its rate of change is expressed as follows:

$$\frac{\delta \text{Time}_{\text{Ants}}}{\delta \text{Ants}} = \frac{\delta \text{Rounds}}{\delta \text{Ants}} \times \frac{\delta \text{Time}_{\text{Ants}}}{\delta \text{Rounds}}$$  \hspace{1cm} (B.8)

As moving each ant in a round takes a constant-bounded time with an efficient implementation (and potentially more with an inefficient implementation),

$$\frac{\delta \text{Time}_{\text{Ants}}}{\delta \text{Rounds}} \in \Omega(\text{Ants})$$  \hspace{1cm} (B.9)

holds and by substituting (B.7) into (B.8), we get (B.10):

$$\frac{\delta \text{Time}_{\text{Ants}}}{\delta \text{Ants}} \in \Omega(\text{Ants}^{-0.5} \times \text{Ants}) = \Omega(\text{Ants}^{0.5})$$  \hspace{1cm} (B.10)

By integrating with respect to $\delta \text{Ants}$, we obtain (B.11):

$$\text{Time}_{\text{Ants}} \in \Omega(\text{Ants}^{1.5})$$  \hspace{1cm} (B.11)

The area management component of the total time can be approximated by combining (B.2) and (B.4):

$$\text{Time}_{\text{Area}} \in \Omega(\text{Ants})$$  \hspace{1cm} (B.12)

The following lower-bound approximation holds for the cumulative pheromone management time, as implied by (B.3) and (B.6):

$$\text{Time}_{\text{Pheromone}} \in \Omega(\text{Ants}^{1.5})$$  \hspace{1cm} (B.13)

Finally, from (B.1), (B.11), (B.12) and (B.13), we have an estimation of the time complexity of AntWorld simulation with respect to the number of ants:

$$\text{Time} \in \Omega(\text{Ants}^{1.5} + \text{Ants} + \text{Ants}^{1.5}) = \Omega(\text{Ants}^{1.5})$$  \hspace{1cm} (B.14)

In reality, the ants do not follow an optimal strategy for exhaustively retrieving all food available within the discovered radius, but rather they are diverted by pheromones towards the direction of previously found food bundles, distorting the circularity of explored area, and needlessly expanding the grid. Also, a number of ant steps are wasted during the search for new food sources. Consequently, the boundary expressed in (B.4) turns out to be weak; according to regression calculations performed on our measurements, the number of ants seem to be proportional to the area to the power of approximately 0.66. This also means that ants have a smaller birthrate than allowed in (B.7), and therefore (B.10) will not give a close approximation of the time spent on ant management. Finally, as even (B.12) gives a weak boundary, the total time may have a complexity higher than $\text{Ants}^{1.5}$. Our experiments confirm this assumption: for rounds 100-500, regression gave the approximation of $\text{Time} \sim \text{Ants}^{1.68}$ with a correlation over 99% for our solution (the first 100 rounds appeared more random and less characteristic). Nevertheless, this is still a low-order polynomial behaviour, see Figure 6.5 for the results; the complexity is visually confirmed by using logarithmic scales for both axes (Figure B.2).
Figure B.2: Cumulative Execution Time (double logarithmic scale)
The current appendix gives a general overview and analysis on the use of LS and INC graph pattern matching as well as the hybrid approach using the Object-to-Relation schema transformation case study.

C.1 Description of the Case study

C.1.1 Transformation overview.

The (simplified) Object-to-Relational schema mapping (ORM) case study was proposed as a performance benchmark of model synchronization transformations in both [VSV05] and [17]. The aim of the transformation is to produce corresponding relational database schemas from UML class diagrams according to the following mapping rules:

- First, a relational schema is created for the specified package below a given container by \textit{schemaRule} (Figure C.1). Transitive containment is represented by an edged tagged as “contains”.

- Then classes in the package are mapped into tables in the corresponding schema, each with an id column as a primary key (\textit{classRule}).

- Each association in the package is mapped into a table in the corresponding schema with a primary key column (\textit{associationRule});
- Each association end in the association is mapped into a foreign key of the corresponding table pointing at the table generated from the class that that the association end points to (assocEndRule);

- Attributes in the class are mapped into columns of the corresponding table (see attributeRule in Figure C.3).

\[
g_t rule\ schemaRule(P, Con) =
\{
    package(P) below Con;
    neg\ pattern\ mapped(P, SN, RN) =
    \{
        package(P);
        schema(SN);
        package.schemaRef(RN, P, SN);
    }
    new\ schema(SN);
    new\ package.schemaRef(RN, P, SN);
\}
\]

Figure C.1: GT rule for unmapped packages below a container

Figure C.2: Reference metamodel

In incremental synchronization, to avoid rebuilding target models in each pass, a reference model (also known as trace / correspondence model) is used to establish a mapping between source and corresponding target model elements (Figure C.2) and to trace changes in them. In this context, the reference model is often referenced in NACs to identify elements in the source model that have not yet been mapped into the target model. It is important to mention that we restrict our investigations to one-way synchronization.

\[
g_t rule\ attributeRule(C, A, T) =
\{
    class(C);
    class.attribute(A);
    class.col2attrs(CFI, C, A);
    table(T);
    class.tableRef(R1, C, T);
    neg\ pattern\ mapped(A, ColN, RN) =
    \{
        class.attribute(A);
        column(ColN);
        class.attribute.colRef(RN, A, ColN);
    }
    new\ column(ColN);
    new\ class.attribute.colRef(RN, A, ColN);
\}
\]

Figure C.3: GT rule for unmapped attributes
C.2. MOTIVATING SCENARIOS FOR HYBRID PATTERN MATCHING

C.1.2 Transformation scenario.

In the original benchmark example [17], the source model consists of two packages, both containing a (generated) set of classes with attributes. First, (i) the primary package will be mapped into a relational schema to create initial mappings. Then, the source models are modified, and, in an additional pass, (ii) the system has to synchronize the changes to the target model (i.e., find the changes in the source and alter the target accordingly). This scenario is now extended as follows.

Check phase First, we check as a precondition of the transformation that no generalization exists in the source UML model to ensure the applicability of the transformation. It is captured by a corresponding simple graph pattern (detectGeneralization in Figure C.4). We intentionally omitted support for inheritance from the model transformation itself to analyze a case where a transformation has to perform a preliminary applicability check; the practical consequences of this choice will be assessed later in Section C.2.2.

Initial transformation phase When there are no generalizations, the primary package is mapped into a relational schema by the transformation program.

Refactoring phase A refactoring operation modifies the package hierarchy of the source model, namely the secondary package is moved inside the primary package.

Synchronization phase Afterwards, synchronization propagates these changes into the target relational model so that it holds the mapping of the (changed) primary package once again. This involves creating the tables for classes that stem from the secondary package, and creating columns for attributes of these classes.

\[
\text{pattern } \text{detectGeneralization}(	ext{Sub}, \text{Sup}) = \\
\{ \\
\text{general(} \text{Gen} \text{);} \\
\text{class(} \text{Sup} \text{);} \\
\text{general.parent(} \text{PE}, \text{Gen}, \text{Sup}); \\
\text{class(} \text{Sub}); \\
\text{general.child(} \text{CE}, \text{Gen}, \text{Sub}); \\
\}
\]

Figure C.4: Graph pattern checking for generalizations

C.2 Motivating Scenarios for Hybrid Pattern Matching

Recent benchmarks evaluations [17] and tool contests [23] in the graph transformation community have shown that INC can easily be orders of magnitude faster than (most) LS approaches for certain problem classes. This section identifies three scenarios where, on the other hand, LS has a clear advantage, as demonstrated by our experiments\(^1\). For each scenario, we identify a hybrid pattern matching approach where some patterns and transformations should use LS, while the rest of the transformation relies upon INC to obtain a better performance than the two extremes (LS-only or INC-only).

\(^1\)Measurement environment: Intel Core Duo t2400@1,83 GHz processor, 3 GB RAM, Windows XP SP3, Sun HotSpot Java 1.6.0_02 and Viatra2 Release 3 build 2009.02.03.
APPENDIX C. SIMPLIFIED ORM MAPPING BENCHMARK FOR HYBRID GRAPH PATTERN MATCHING

PM Strategy | Memory limit [MB] | Used heap [MB] | Transform phase time [ms]
---|---|---|---
LS | 128 | 99 | 183729
INC | 128 | 128 | 21057
INC | 1024 | 128 | 4639
Hybrid | 128 | 105 | 5573

Table C.1: Match Set Memory and Performance

C.2.1 Scenario: match set cache does not fit into memory limit

This scenario demonstrates that the high memory consumption of incrementally maintained caches can be a bottleneck of INC. By choosing LS for patterns that are memory-intensive (i.e. with many matches) but not time-critical, the high memory consumption can be greatly reduced, while still retaining the short execution time comparable to INC.

Our experiments were performed on the Transform phase of the ORM case study (see in Section C.1), by measuring the heap commit size of the Java VM. In the followings, we model a frequently occurring development scenario. As the transformation designer is typically working with small toy models, scaling up to large model sizes might lead to unexpected results. For instance, while a toy model with 10 classes and 250 attributes and the corresponding INC cache easily fits in a few megabytes, a memory usage of 128MBs can be reached by increasing the model to 575 classes and 14375 attributes, as shown on Table C.1. With a memory limit of 128M, as the match set cache expands rapidly, the JVM begins to trash due to memory starvation shortly after the transformation is started. This leads to significant slowdown (to 21 seconds), and may even result in a failed execution because of heap exhaustion. If the amount of memory is suitably large (i.e. 1GB in our case), execution is very fast (4.6 seconds). LS is not an alternative here: while the memory consumption of the caches is spared, the execution time for this model size is very long (avg. 184 seconds).

Closely observing this transformation, we may identify LHS pattern attributeRule (see in Figure C.3) and its embedded negative application condition as patterns with high number of occurrences. By sacrificing execution time (runs in 5.0s with a 1G heap), we marked this pattern to be matched by the LS engine, despite using INC for the rest of the transformation. This reduced memory consumption to 105M, and allowed the transformation to run with approximately the same execution time (5.6s) even with a memory limit of 128M. Therefore the hybrid approach has the potential to efficiently scale up to higher model sizes given the same memory constraints.

C.2.2 Scenario: construction time penalty

This scenario emphasises that the time required to initialize the incrementally maintained caches might itself be too expensive. The construction time of the caches is not less than the time required to find all occurrences of the pattern, since the match set is directly available from this cache. If the transformation needs to find only one (or few) of many pattern occurrences altogether, there is no need for LS to continue the search and retrieve the entire match set, therefore it can be significantly faster than INC. This phenomenon only applies if the pattern is efficiently matchable by LS, unlike large patterns with high combinatorial complexity.

This behaviour was observed in the Check phase of the ORM case study (see in Section C.1). We measured the time it takes to find an arbitrary generalization edge if all 2500 generated classes inherit a common superclass, which is a single-occurrence query of a very simple graph pattern (see in Figure C.4) consisting of a single edge. The measurements show (in Table C.2) that constructing
C.2. MOTIVATING SCENARIOS FOR HYBRID PATTERN MATCHING

<table>
<thead>
<tr>
<th>PM Strategy</th>
<th>Used heap [MB]</th>
<th>Cache construction time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS</td>
<td>152</td>
<td>10</td>
</tr>
<tr>
<td>INC</td>
<td>159</td>
<td>143</td>
</tr>
</tbody>
</table>

Table C.2: Construction Time Performance

<table>
<thead>
<tr>
<th>PM Strategy</th>
<th>Used heap [MB]</th>
<th>Refactoring phase time [ms]</th>
<th>Synchronization phase time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS</td>
<td>-</td>
<td>0</td>
<td>&gt;2000000</td>
</tr>
<tr>
<td>INC</td>
<td>493</td>
<td>2109</td>
<td>13386</td>
</tr>
<tr>
<td>Hybrid</td>
<td>298</td>
<td>0</td>
<td>13570</td>
</tr>
</tbody>
</table>

Table C.3: Model Update Performance

the cache took 0.14s on average, while INC would gain only about 16 ms time (too small to be more accurately measured) compared to LS with each further query (if there were any). To complement these results, we also took the measurement on a source model without generalization, for which the transformation could be performed; we found that, in accordance with expectations, LS has no significant advantage in this case: constructing the cache took 16 ms, while LS needed 36 ms to complete query (both of them too small to be accurately measured).

C.2.3 Scenario: expensive model updates

This scenario happens if there is heavy model manipulation between infrequent pattern queries. In this case, the time overhead imposed on model manipulation by INC may outweigh its benefits. The cost of incrementally maintaining the match set caches for a long period of time with frequent model updates may be larger than the cost of applying LS and calculating the match set from scratch at each pattern query. In other terms, it may be superfluous to continuously maintain the match sets if they are not frequently used for model queries.

Expensive update overhead is observable in the Refactoring phase of the ORM case study (see in Section C.1). We measured the time it takes to move a package in the source model to a different package, while the INC maintains the caches of patterns that represent the location of classes in the namespace hierarchy of packages, classes and attributes (Table C.3). The transitive containment is a model feature of high combinatorial complexity, and moving a high-level element will cause drastic changes in this relationship, thereby forcing the INC to perform intensive cache updates. The measurements have shown that the cost of the single move operation can be as high as 2.1 seconds with INC. Using pure LS was not a feasible solution either, as the Synchronization phase did not terminate within half an hour. A hybrid pattern matcher assignment solved these problems: patterns using the transitive containment (see in Figure C.1) were matched by LS and the rest by INC, resulting in a fast move operation and an execution time of 14.5 ms for the entire Refactoring phase. These measurements were taken with both the primary and secondary packages consisting of 1000 classes and 25000 attributes.
C.2.4 Overall performance on the entire case study

Finally, we compare the overall performance of the three approaches on all three steps of the case study combined. Measurements were taken for various source model sizes, scaling up until the transformation became too slow (LS) or did not fit into memory (INC, hybrid). Figure C.5 indicates the total execution time versus the number of classes in the primary source package. For these measurements, the number of classes in the secondary package (N/4) was always one quarter of the number of classes initially in the primary package (N), and each class still had 25 attributes (25N, 25N/4); thus the largest case (N=2400) consisted of 2400 classes and 60000 attributes in the primary package, 600 classes and 15000 attributes in the secondary package, i.e. more than 150 000 source model elements altogether including edges. As the figure shows, INC scales up higher than LS, but the hybrid approach is even more efficient.

![Figure C.5: Overall Execution Time](image)

C.2.5 Hybrid Characteristic

By evaluating the qualitative metrics defined in Section 6.4.1 on the ORM case study, the observed behaviour in Scenarios C.2.1–C.2.3 can be explained in more detail.

- In Sec. C.2.1, we have identified the cause of the performance bottleneck to the `attributeRule` graph pattern with large match set. Since this pattern is used to filter for Attributes which have not yet been mapped to a Table column, it can be expected to have an initially large match set for class models with a large number of attributes. The match set size can be estimated a-priori by looking at `instance count` numbers for the Attribute type, or, by simply considering the general type composition characteristics of models the transformation is to be executed on.

- Sec. C.2.2 demonstrated the usage simple pattern for structural checking (i.e. executing only once). This case corresponds to low pattern complexity and low usage count which, especially when combined with a potentially high match count, indicates a good candidate for switching to LS.
Finally, Sec. C.2.3 uses a pattern with a transitive containment constraint which, when used for synchronization after a model move high in the containment hierarchy, caused a drastic overhead for the incremental pattern matcher. As the resolution suggests, such patterns should generally be matched with LS.
The current appendix presents configuration artifacts generated by the DIANA PIM-PSM mapping approach presented in Figure 10. However, due to the strict consortium restrictions on results of the project the presented artifacts are not the ones generated for our running air conditioning case study.

These configuration artifacts represent a simple aircraft that has only one module OS and all its functions are allocated to the partition of this module (see in Section D.1). It uses the logbook AIDA service (see its configuration in Section D.3 and has separate health monitoring tables for its modules (see in Section D.2) and partitions. Finally, generated traceability files for the logbook configuration artifacts is presented in Section D.4.

### D.1 Configuration Artifact for the Module Operating System

```xml
  <CoreSS>
    <xi:include href="mos.xml"/>
  </CoreSS>
  <Applications>
    <Application Name="flight controls">
      <xi:include href="flight controls.xml"/>
    </Application>
    <Application Name="flight management">
      <xi:include href="flight management.xml"/>
    </Application>
    <Application Name="ID processing">
      ...
    </Application>
  </Applications>
</Module>
```
<xi:include href="IO processing.xml"/>
</Application>
<Application Name="passenger information system">
<xii:include href="passenger information system.xml"/>
</Application>
<Application Name="Middleware_for_Channels">
<xii:include href="Middleware_for_Channels.xml"/>
</Application>
</Applications>
</Partitions>
<Connections>
<Source Partition NameRef="passenger information system" PortNameRef="ACP_1_T0_ZC_1__CO_S"/>
<Destination Partition NameRef="flight controls" PortNameRef="ACP_1_T0_ZC_1__CO_RO"/>
</Channel>
<Channel Id="1">
<Source Partition NameRef="passenger information system" PortNameRef="ACP_1_T0_PFC_1_AND_PFC_2__S"/>
<Destination Partition NameRef="flight controls" PortNameRef="ACP_1_T0_PFC_1_AND_PFC_2_R0"/>
</Channel>
<Channel Id="2">
<Source Partition NameRef="passenger information system" PortNameRef="ACP_1_T0_PFC_1_AND_PFC_2__S"/>
<Destination Partition NameRef="Middleware_for_Channels" PortNameRef="MW.ACP_1_T0_PFC_1_AND_PFC_2_R"/>
</Channel>
<Channel Id="3">
<Source Partition NameRef="Middleware_for_Channels" PortNameRef="MW.ACP_1_T0_PFC_1_AND_PFC_2_S"/>
</Channel>
D.1. CONFIGURATION ARTIFACT FOR THE MODULE OPERATING SYSTEM

```xml
<Connections>
  <Channel id="4">
    <Source PartitionNameRef="passenger information system" PortNameRef="ACP_1.T0.ZC_1..CS.S"/>
    <Destination PartitionNameRef="flight controls" PortNameRef="ACP_1.T0.ZC_1..CS.R0"/>
  </Channel>
  <Channel id="5">
    <Source PartitionNameRef="Io processing" PortNameRef="PP_1.T0.SD_1..CO_S"/>
    <Destination PartitionNameRef="Middleware_for_Channels" PortNameRef="MW_PP_1.T0.SD_1..CO_R"/>
  </Channel>
  <Channel id="6">
    <Source PartitionNameRef="Middleware_for_Channels" PortNameRef="MW_PP_1.T0.SD_1..CO_R"/>
    <Destination PartitionNameRef="flight management" PortNameRef="PP_1.T0.SD_1..CO_R"/>
  </Channel>
  <Channel id="7">
    <Source PartitionNameRef="flight controls" PortNameRef="ZC_1.T0.PTC_1..AND_PTC_2.S"/>
    <Destination PartitionNameRef="flight management" PortNameRef="ZC_1.T0.PTC_1..AND_PTC_2.R0"/>
    <Destination PartitionNameRef="Io processing" PortNameRef="ZC_1.T0.PTC_1..AND_PTC_2.R1"/>
  </Channel>
  <Channel id="8">
    <Source PartitionNameRef="Io processing" PortNameRef="ZTP_1.T0.ZC_1..AND_SD_1..CO_S"/>
    <Destination PartitionNameRef="Middleware_for_Channels" PortNameRef="MW_ZTP_1.T0.ZC_1..AND_SD_1..R"/>
  </Channel>
  <Channel id="9">
    <Source PartitionNameRef="Middleware_for_Channels" PortNameRef="MW_ZTP_1.T0.ZC_1..AND_SD_1..R"/>
    <Destination PartitionNameRef="flight controls" PortNameRef="ZTP_1.T0.ZC_1..AND_SD_1..S"/>
    <Destination PartitionNameRef="flight management" PortNameRef="ZTP_1.T0.ZC_1..AND_SD_1..R0"/>
  </Channel>
</Connections>
```

</%xml>
APPENDIX D. GENERATED CONFIGURATION ARTIFACTS FOR ARINC 653

D.2 Configuration Artifact for the Module Health Monitor Table

Listing D.1: XML configuration for the ARINC 653 Module operating system

D.3 Configuration Artifact for the AIDA Logbook

Listing D.2: XML configuration for the ARINC 653 module level health monitor table
D.4 Traceability Artifact for the AIDA logbook from the Integrated System Model

```
<?xml version="1.0" encoding="UTF-8"?>
<traceability:TraceRoot xmlns:traceability="http://www.dianaproject.com/traceability"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:aidalogbook="#schema:aidalogbook" standalone="no">
  <aidalogbook:Trace>
    <clientJobs href="Diana-07-08.psm#/Dias.O/@ownedJob.2"/>
    <clientJobs href="Diana-07-08.psm#/Dias.O/@ownedJob.3"/>
    <clientJobs href="Diana-07-08.psm#/Dias.O/@ownedJob.7"/>
    <clientJobs href="Diana-07-08.psm#/Dias.O/@ownedJob.6"/>
    <clientJobs href="Diana-07-08.psm#/Dias.O/@ownedJob.1"/>
    <allocatedToModules href="Diana-07-08.psm#/Dias.O"/>
    <replicas href="AIDALogbook.xml#/@system/AidaLogbook.O/@replica.0"/>
    <replicas href="AIDALogbook.xml#/@system/AidaLogbook.O/@replica.1"/>
    <replicas href="AIDALogbook.xml#/@system/AidaLogbook.O/@replica.2"/>
    <clients href="AIDALogbook.xml#/@system/AidaLogbook.O/@client.0"/>
    <clients href="AIDALogbook.xml#/@system/AidaLogbook.O/@client.1"/>
    <clients href="AIDALogbook.xml#/@system/AidaLogbook.O/@client.2"/>
    <clients href="AIDALogbook.xml#/@system/AidaLogbook.O/@client.3"/>
    <clients href="AIDALogbook.xml#/@system/AidaLogbook.O/@client.4"/>
    <clients href="AIDALogbook.xml#/@system/AidaLogbook.O/@client.5"/>
    <clients href="AIDALogbook.xml#/@system/AidaLogbook.O/@client.6"/>
    <logbook href="AIDALogbook.xml#/@system/AidaLogbook.O"/>
  </aidalogbook:Trace>
</traceability:TraceRoot>
```

Listing D.4: Traceability file for the logbook service
List of publications

Publication list of Ákos Horváth

Number of publications: 41
Number of peer-reviewed publications: 27
Number of independent citations: 51

International, peer-reviewed journal papers


National journal papers


Book chapter


International conferences


International workshops and tutorials


LIST OF PUBLICATIONS


Hungarian conferences


Technical reports and online content


Bibliography


Alex Wilson. The evolving ARINC 653 standard and it’s application to IMA, November 13th 2007. ARTIST2 meeting on Integrated Modular Avionics, Rome, Italy.


