Model Transformation-based Design of Dependable Systems

PhD thesis

András Balogh
MSc in Technical Informatics

Supervisor: Prof. András Pataricza, DSc
Tutor: Dániel Varró, PhD

Budapest 2009
Nyilatkozat

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


A dolgozat bírálatai és a védésről készült jegyzőkönyv a későbbiekben a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Inormaikai Karának Dékáni Hivatalában elérhetők.
Model Transformation-based Design of Dependable Systems
András Balogh
PhD thesis summary

Abstract. Design of dependable systems has to fulfill several different requirements including functionality, timeliness, dependability, and cost, which increases the design complexity dramatically and necessitates a strong algorithmic support. The thesis proposes a model-driven framework introducing automated and interactive solutions for the modeling, analysis, and synthesis of dependable systems in a dependability driven way. The integration of tools implementing the best industrial practice for the individual subtasks is facilitated by an advanced model-transformation based tool infrastructure.

The main trends in the development of dependable embedded computer systems are the growing complexity and spectrum of functionalities and the shortening of the life-cycle of systems, which demand novel, more productive development methods and tools. The built-in algorithmic intelligence supports the effective development and simultaneously maintains compliance to dependability and other non-functional requirements, and assures a proper design quality by integrated verification and validation techniques.

Model-driven development became a main trend in pure software development in a variety of domains (like enterprise information systems), but it faces significant difficulties in the embedded systems domain due to the large variety of non-functional aspects, strict design constraints.

Model-driven embedded systems development necessitates the definition of specific modeling languages in order to create compact models fitting well to the peculiarities of the particular application domain due to the huge variety embedded system application domains. The development of complex, model-driven tools and techniques require a solid foundation on the language definition (metamodeling) and model transformation environments level in order to avoid conceptual flaws in the modeling and synthesis process.

The thesis proposes such a mathematically well founded, iterative, interactive model-driven development environment and methodology for dependable embedded systems that is complemented by formal analysis and synthesis methods in order to guarantee a compliance to the specific requirements of the embedded systems domain. The main contributions of the thesis are the following ones:

**Model-driven tool infrastructure**: Several extensions are proposed to the existing model transformation approaches in order to increase the productivity of model transformation development by compacting the transformation programs and enable reuse of artifacts on different levels of abstraction. The integration of different modeling languages in a single, uniform, and precise formalism is supported by mapping the most widely used metamodeling approaches (MOF, ECore) to the VPM (Visual and Precise Metamodeling) paradigm, thus assuring a design environment independence to the main results achieved. This syntactic uniformization is complemented by the definition of a precise, operational semantics for these metamodeling approaches assuring a formally well founded integration of modeling tools and model transformations from heterogeneous environments.

**Iterative, interactive hardware-software integration**: Despite the necessary precise formulation of the design environment and target model human interaction cannot be avoided as in many cases some additional design constraints and objectives are rather specific to the particular application under design. The thesis proposes a novel, iterative, interactive framework for hardware-software integration of dependable embedded systems, which combines automatic and interactive steps in a seamless way by automating repetitive tasks but simultaneously granting the designer with taking key design decisions upon necessity.

Automated hardware-software integration is a time consuming process, therefore an early indication of potential design problems is extremely important in order to avoid unnecessary design iteration cycles originating in the infeasibility of a design candidate. The thesis proposes a novel, open and extensible design constraint and rule checker framework for domain-specific checks, which is tightly integrated into the hardware-software integration framework and assures the earliest possible detection of design flaws.

**Dependability-driven synthesis**: Mathematical optimization of resource optimization may grant additional computing power and resources for enforcing the dependability of the system by replication at virtually no cost.

A hardware cost minimization based optimization method is proposed in the thesis that synthesises the high-level architecture of the designated distributed system and simultaneously assures the compliance to dependability and performance requirements.

A multi-aspect (system cost, performance, robustness, node utilization) optimization based method for resource-task allocation and scheduling is proposed for the important category of time-triggered systems widely used in safety critical applications.
Megbízható rendszerek modelltranszformáció-alapú fejlesztése
Balogh András
Doktori diszertáció kivonata

Kivonat. Megbízható rendszerek tervezése során számos aspektust érintő követelményeket kell figyelembe venni, mint az időzítések, megbízhatóság, és költség, ami jelentősen megnöveli a tervezés bonyolultságát és erős algoritmikus támogatást tesz szükségessé.

Jelen disszertáció egy automatikus és interaktív megoldásokat tartalmazó modell-alapú tervező keretrendszer segítségével valósítjuk meg.

A megbízható beágyazott rendszerek fejlesztése területén a fő trend az elvált szolgáltatások komplexitásának és specifikus kényszerek termelésének növekedése, a rendszerek életciklusának rövidülése mellett, ami új, hatékonyabb fejlesztési módszerek és szoftverek bevezetését teszi szükségessé.

Az egyes megoldásokat implementáló eszközök integrációját egy korszerű, továbbfejlesztett modelltranszformáció alapú keretrendszer segítségével valósítjuk meg.

A modell-alapú tervezéshez a fő trend a különböző alkalmazási területeken (például a nagyvállalati rendszerek területén), de a beágyazott rendszerek esetén használata problémákba ütközik a különböző nem-funkcionális követelmények és tervezési kényszerek miatt.

A beágyazott rendszerek modell-alapú fejlesztéséhez specifikus modellépítő nyelveket kell definiálni, hogy kompakt, a tervezés speciális fogalmait ábrázolni képes formalizmus használhassunk. A modell-alapú tervezőeszközök és technikák fejlesztéséhez jól definiált nyelvet és metamodelleket kell használni.

A disszertáció egy matematikailag megalapozott, iteratív, interaktív modell-alapú fejlesztési keretrendszert és módszertant mutat be, amit formális analízis és szintézis módszerek egységeinek kivonásával és beépítésével valósítjuk meg.

Jelen disszertáció egy matematikailag megalapozott, iteratív, interaktív modell-alapú tervezési keretrendszert és módszertant mutat be, amit formális analízis és szintézis módszerek egységeinek kivonásával és beépítésével valósítjuk meg.

Modell-alapú eszköz infratrákturája: Több kiterjedt tervező környezetet és módszertant mutat be, amit formális analízis és szintézis módszerek egységeinek kivonásával és beépítésével valósítjuk meg.

Iteratív, interaktív hardver-szoftver integráció: Sok esetben az alkalmazás és a célpont determinált, de a beágyazott rendszerek esetén használata problémákba ütközik a különböző nem-funkcionális követelmények és tervezési kényszerek miatt.

Az automatikus hardver-szoftver integráció hosszú időt vesz igénybe, ezért különböző alkalmazás-specifikus, egyedi tervezési kényszereket és célokat is figyelembe kell venni. A disszertációban egy iteratív, interaktív keretrendszt javaslunk megbízható rendszerek hardver-szoftver integrációjának támogatására, mely kombinálja az automatikus, interaktív lépéseket, automatizálva az ismétlődő feladatokat, amikor lehetővé teszi a fejlesztőnek a tervezési döntéseket előállítását.

Az automatikus hardver-szoftver integráció hosszú időt vesz igénybe, ezért különböző alkalmazás-specifikus, egyedi tervezési kényszereket és célokat is figyelembe kell venni. A disszertációban egy iteratív, interaktív keretrendszt javaslunk megbízható rendszerek hardver-szoftver integrációjának támogatására, mely kombinálja az automatikus, interaktív lépéseket, automatizálva az ismétlődő feladatokat, amikor lehetővé teszi a fejlesztőnek a tervezési döntéseket előállítását.

Megbízhatóság-vezérelt szintézis: Az erőforrás-hozzárendelés matematikai optimalizálásának alapján megbízhatóság kedvezőbb tervezése a rendszereket és megfelelő tervezési kényszereket előállítja.

Az automatikus hardver-szoftver integráció hosszú időt vesz igénybe, ezért különböző alkalmazás-specifikus, egyedi tervezési kényszereket és célokat is figyelembe kell venni. A disszertációban egy iteratív, interaktív keretrendszt javaslunk megbízható rendszerek hardver-szoftver integrációjának támogatására, mely kombinálja az automatikus, interaktív lépéseket, automatizálva az ismétlődő feladatokat, amikor lehetővé teszi a fejlesztőnek a tervezési döntéseket előállítását.

Egy hardver költség minimálisával alapuló optimalizálási módszertant javaslunk a disszertációban, mely lehetővé teszi a tervezés alapú elősztott rendszer magas szintű architektúrájának előállítását, ki- előgett megfelelő tervezési követelményeket.

Több aspektust (költség, teljesítmény, robusztusság, megbízhatóság) optimalizálásának alapján alapuló módszert javaslunk az erőforrás-taszk allokáció és statikus ütemezés megvalósítására idővezérelt rendszerekben, melyeket széles körben használnak biztonságosként alkalmazásokban.
Preface

This thesis discusses the result of my research on the topic of model-driven development of dependable systems throughout many pages. But prior to that I would like to spend one page to thank all those people who gave significant assistance to me in writing this thesis in a rather informal way. As it is impossible to measure the significance of such assistance and support, the sequence of the following paragraphs is not trying to set up a priority list.

First of all, I should acknowledge András Pataricza, the supervisor of my thesis. He proposed the main direction of the research for me that triggered several parallel tracks and led to the results described in the current paper. He taught me a lot not only on dependable computing, formal analysis and synthesis methods, but also on non-technical skills in order to be able to properly present my ideas on project meetings, workshops, and conferences.

I should also acknowledge Dániel Varró, the tutor of my thesis who helped to understand the concepts related to model-driven methodologies, model transformations and domain-specific languages. I think we had a fruitful joint work that resulted in interesting results and several joint papers on model transformation related topics.

I got a lot of support from György Csertán, my colleague both at the University and OptXware. We worked together from the beginning of my PhD studies and had valuable conversations and joint work on different embedded systems related topics.

I am very grateful to my colleagues at the Fault Tolerant Systems Research Group (Department of Measurement an Information Systems, Budapest University of Technology and Economics) for the kind research environment. Discussions with many of them on various topics helped me a lot in my research work. I would like to acknowledge also the VIATRA development team for their work, and support on VIATRA related issues.

I wrote part of my thesis during 2009, and got a lot of support from my colleagues at OptXware Research and Development Ltd. They did everything to unload me from usual tasks to have enough time to complete the current thesis. Some of the tools based on my thesis have also been developed at the Company.

In the Summer of 2006, I spend two months with the group of Professor Hartmunt Ehrig, at the Technical University of Berlin, Germany in the framework of the Segravis project. I would like to acknowledge his group the interesting discussions about different topics that contributed significantly to my work.

I am very grateful to Professor Neeraj Suri and his group (Dependable, Embedded Systems and Software Group, Technical University of Darmstadt, Germany) for their support and kind hosting me on my summer visits. The joint work and publications on several projects were very interesting and valuable for me. The inspiring environment kept me productive enough to make significant progress in my work even during the short visits.

Part of my work has been supported by the European projects DECOS and GENESYS. During this projects I got valuable feedback from all the project partners having different background (academia and industry) that helped me to further improve my contributions.

I should also acknowledge many researchers including at least the entire staff of the Department of Professor Hermann Kopetz (TU Vienna), Wolfgang Herzner, Rupert Slick, Erwin Schoitsch (Austria Research Centers), Michalis Anastasopoulos (Fraunhofer Institute Kaiserslautern) for the fruitful discussions, joint work, and/or their encouragement and support.

Finally, I get to those people for acknowledgement who did everything to establish a stable background for me, and gave love, encouragement and support and did not let me to surrender even if the problems and time pressure seemed to make impossible the finishing of the thesis: Vera (my fiancee), my parents Zsuzsanna and Zoltán, and my grandparents.
Contents

1 Introduction ......................................................... 13
  1.1 Design Reusability ........................................... 13
    1.1.1 Design patterns / pattern based engineering ............ 13
    1.1.2 Component-based systems ................................ 14
    1.1.3 Distributed systems .................................... 16
  1.2 Model-driven development .................................. 16
    1.2.1 MDA .................................................. 16
    1.2.2 Metamodelling .......................................... 18
    1.2.3 UML .................................................. 18
    1.2.4 DSL .................................................. 18
    1.2.5 Visual Programming Languages ......................... 18
  1.3 Formal analysis methods ................................... 19
  1.4 Objectives of the research ................................ 19
  1.5 Structure of the paper .................................... 19

2 Model-driven Development ....................................... 21
  2.1 Development workflow ...................................... 21
  2.2 Metamodelling Environment ................................ 22
    2.2.1 Meta Object Facility .................................. 22
    2.2.2 ECore ................................................ 23
    2.2.3 VPM: Visual and Precise Metamodelling ............... 23
    2.2.4 VPM modeling concepts ................................ 24
    2.2.5 Algebraic representation of models .................... 24
    2.2.6 VPM and MOF .......................................... 25
    2.2.7 Mapping MOF and ECore to VPM ......................... 25
  2.3 Model transformations ...................................... 25
    2.3.1 Graph patterns ........................................ 26
    2.3.2 Additional pattern constructs ......................... 28
    2.3.3 Model Transformations ................................ 29
  2.4 Summary .................................................... 34

3 Model-driven tool infrastructure ................................ 35
  3.1 Introduction ................................................ 35
  3.2 Advanced GT language constructs .......................... 35
    3.2.1 Composition of simple patterns ....................... 35
    3.2.2 Composition of complex patterns ..................... 36
    3.2.3 Definition of design patterns ......................... 37
    3.2.4 Conclusions .......................................... 39
  3.3 Cascading Graph Transformations ............................ 39
    3.3.1 Overview of the Approach ................................ 39
    3.3.2 Metamodel conversion .................................. 40
    3.3.3 Model migration ....................................... 40
    3.3.4 Transformation cascading ................................ 41
    3.3.5 Performance evaluation of cascaded transformations .... 44
    3.3.6 Reuse of VPM importers on the EJB platform ........... 44
    3.3.7 Related work ......................................... 45
    3.3.8 Conclusions .......................................... 45
### 3.4 A formal semantics for ECore

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1 Related work</td>
<td>45</td>
</tr>
<tr>
<td>3.4.3 Operations on ECore models</td>
<td>47</td>
</tr>
<tr>
<td>3.4.4 GTASM definition of ECore operations</td>
<td>49</td>
</tr>
<tr>
<td>3.4.5 Live model sharing</td>
<td>53</td>
</tr>
<tr>
<td>3.4.6 Conclusions</td>
<td>53</td>
</tr>
</tbody>
</table>

### 3.5 Workflow-based transformation integration

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.1 High level architecture of MDD tools</td>
<td>53</td>
</tr>
<tr>
<td>3.5.2 Key components to be implemented</td>
<td>54</td>
</tr>
<tr>
<td>3.5.3 Generative Framework for Tool Synthesis</td>
<td>55</td>
</tr>
<tr>
<td>3.5.4 Conclusions</td>
<td>56</td>
</tr>
</tbody>
</table>

### 3.6 Applications

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6.1 VIATRA2 Framework</td>
<td>56</td>
</tr>
<tr>
<td>3.6.2 DECOS Toolchain</td>
<td>57</td>
</tr>
<tr>
<td>3.6.3 PIM-PSM mapping tool synthesis</td>
<td>57</td>
</tr>
<tr>
<td>3.7 Conclusions</td>
<td>57</td>
</tr>
</tbody>
</table>

### 4 Embedded Systems

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>59</td>
</tr>
<tr>
<td>4.2 ARTEMIS Challenges</td>
<td>60</td>
</tr>
<tr>
<td>4.3 Event and Time triggered paradigms</td>
<td>61</td>
</tr>
<tr>
<td>4.3.1 Timing model of a software component</td>
<td>61</td>
</tr>
<tr>
<td>4.3.2 Timing model of communication</td>
<td>61</td>
</tr>
<tr>
<td>4.3.3 Event Triggered Systems</td>
<td>62</td>
</tr>
<tr>
<td>4.3.4 Time-Triggered Systems</td>
<td>63</td>
</tr>
<tr>
<td>4.4 Host and network characteristics</td>
<td>63</td>
</tr>
<tr>
<td>4.4.1 Network protocols</td>
<td>63</td>
</tr>
<tr>
<td>4.4.2 ECU resources</td>
<td>63</td>
</tr>
<tr>
<td>4.4.3 ECU middleware</td>
<td>64</td>
</tr>
<tr>
<td>4.5 Domain specific Languages</td>
<td>64</td>
</tr>
<tr>
<td>4.5.1 SysML</td>
<td>64</td>
</tr>
<tr>
<td>4.5.2 AADL</td>
<td>64</td>
</tr>
<tr>
<td>4.5.3 Matlab Simulink and StateFlow</td>
<td>64</td>
</tr>
<tr>
<td>4.5.4 SCADE</td>
<td>64</td>
</tr>
<tr>
<td>4.5.5 LabView</td>
<td>64</td>
</tr>
<tr>
<td>4.5.6 UML MARTE</td>
<td>65</td>
</tr>
<tr>
<td>4.5.7 AutoSAR</td>
<td>66</td>
</tr>
<tr>
<td>4.5.8 DECOS PIM</td>
<td>66</td>
</tr>
<tr>
<td>4.5.9 Summary</td>
<td>69</td>
</tr>
</tbody>
</table>

### 5 MDSD with Quality Aspects

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Introduction</td>
<td>71</td>
</tr>
<tr>
<td>5.2 Motivating example: The DECOS tool chain</td>
<td>71</td>
</tr>
<tr>
<td>5.2.1 PIM modeling</td>
<td>72</td>
</tr>
<tr>
<td>5.2.2 CRD and its generation</td>
<td>72</td>
</tr>
<tr>
<td>5.2.3 PSM generation</td>
<td>72</td>
</tr>
<tr>
<td>5.2.4 Conclusions</td>
<td>73</td>
</tr>
<tr>
<td>5.3 Formal metamodels for MDD</td>
<td>73</td>
</tr>
<tr>
<td>5.3.1 Simplified Architecture Model</td>
<td>73</td>
</tr>
<tr>
<td>5.3.2 Execution Platform Modeling</td>
<td>81</td>
</tr>
<tr>
<td>5.4 Iterative, Transformation-based Hardware-Software Integration</td>
<td></td>
</tr>
<tr>
<td>5.4.1 Requirements for the integration process</td>
<td>86</td>
</tr>
<tr>
<td>5.4.2 Inputs and outputs of the process</td>
<td>86</td>
</tr>
<tr>
<td>5.4.3 Model import</td>
<td>87</td>
</tr>
<tr>
<td>5.4.4 Model marking</td>
<td>88</td>
</tr>
<tr>
<td>5.4.5 Mapping</td>
<td>88</td>
</tr>
<tr>
<td>5.4.6 Allocation</td>
<td>88</td>
</tr>
<tr>
<td>5.4.7 Transformation Steps</td>
<td>88</td>
</tr>
</tbody>
</table>
6 Dependability Driven Synthesis

6.1 Introduction ................................................. 103
6.2 High-level availability analysis ......................... 103
  6.2.1 Related Work ........................................... 103
  6.2.2 Availability calculation ............................... 104
  6.2.3 PIM level analysis .................................... 107
  6.2.4 PSM level analysis .................................... 108
  6.2.5 Conclusions ........................................... 109
6.3 Dependability driven architecture synthesis .......... 109
  6.3.1 Related work ........................................... 110
  6.3.2 Fault model ............................................ 110
  6.3.3 Redundancy patterns .................................. 110
  6.3.4 Application and resource model ....................... 110
  6.3.5 Deployment optimization .............................. 112
  6.3.6 Summary .............................................. 113
6.4 Synthesis of Time-triggered systems ..................... 113
  6.4.1 Introduction ........................................... 113
  6.4.2 Related work ........................................... 113
  6.4.3 Problem model ......................................... 114
  6.4.4 The scheduling process ................................. 114
  6.4.5 Non-functional properties ............................ 118
  6.4.6 Multi-aspect optimization ............................. 119
  6.4.7 Evaluation of the method ............................. 119
  6.4.8 Conclusions .......................................... 120
6.5 Synthesis of event-triggered systems .................... 120
6.6 Applications ............................................... 121
  6.6.1 Early analysis of system models ....................... 121
  6.6.2 Deployment optimization ............................... 121
  6.6.3 Allocation and scheduling of time-triggered systems .. 121
6.7 Summary .................................................. 122
# Contents

7 Conclusions  
7.1 Fulfillment of Research Objectives ........................................... 123  
7.2 Utilization of the new results .................................................... 124  
7.3 Open problems ................................................................. 124

8 References  
8.1 References to external publications ............................................. 127  
8.2 References to own publications ................................................. 136

Glossary ................................................................. 139

A Metamodels from Chapter 5  
A.1 NFP Metamodel ................................................................. 141  
A.2 Software Component Architecture Metamodel  
  A.2.1 NFP metamodel ............................................................... 142  
  A.2.2 PIM NFP models .............................................................. 142  
A.3 Platform Metamodel .............................................................. 143  
A.4 Platform Specific Metamodel .................................................... 144

B Algorithms from Chapter 5  
B.1 Connector validity specification ............................................... 147  
  B.1.1 Assembly connectors .......................................................... 147  
  B.1.2 Delegate connectors .......................................................... 147  
B.2 Hardware connector constraints ................................................. 148  
B.3 Data Type Size Calculation ..................................................... 149  
B.4 PIM Flattening ................................................................. 149  
B.5 PIM Component Instance Replication .......................................... 152  
B.6 Platform Model Flattening ...................................................... 155  
B.7 Data type mapping creation ..................................................... 158  
B.8 Job compatibility mapping creation ........................................... 158  
B.9 Communication Graph Synthesis ................................................ 159  
B.10 Data element to message mapping constraints  

C Algorithms and models from Chapter 3  
C.1 Design pattern definitions ....................................................... 163  
  C.1.1 Job with interfaces pattern ............................................... 163  
C.2 ECore metamodel VPM representation ......................................... 164  
C.3 MOF metamodel VPM representation ........................................... 168  
C.4 ECore semantics definition ..................................................... 171

D PIM-PSM mapping example  
D.1 Input models ................................................................. 179  
D.2 The PIM-PSM mapping .......................................................... 180
Chapter 1

Introduction

IT systems have become an essential part of our everyday life surrounding us at home, in the office, and even on different vehicles. The systems range from simple controls to distributed enterprise information systems. A common trend in each field of IT systems is the growing complexity due to the continuously appearing user and business requirements like service integration, network connectivity, and on-demand availability.

An other important aspect of these systems is that they often offer critical functionalities of different kinds. There are business critical functions among enterprise information systems that are required in order to let the primary businesses of the host enterprise work correctly. The malfunction of these systems can cause significant financial losses. Another group of systems performs safety-critical functions (like vehicle control) where not only financial losses can occur, but even humans can be damaged by erroneous system behavior. Critical systems must deliver their services in a dependable manner.

The two aspects (reusability and dependability) mentioned here are often contradictory. On one side, systems should be developed rapidly, even in case of changing requirements, on the other hand dependability is a key factor in many domains. There are several different system development approaches that address this issue by different techniques and methodology.

1.1 Design Reusability

Intellectual Property (IP) reuse is a key factor in achieving better productivity in system development. On one hand, the reuse of an existing solution directly lowers the design and/or implementation efforts, on the other hand flight-proven solutions can increase the overall quality of the system.

IP reuse can be achieved in different phases of the development process. In the design phase, design patterns are used to guide the design by already existing best practices. Code and component reuse can be facilitated during the implementation phase. Integration of different components can also be carried out at runtime.

1.1.1 Design patterns / pattern based engineering

While most of the engineering principles (architecture, civil engineering, mechanical engineering and so on) rely on standards and well formed best practices, software engineering has been a highly intuitive process in the past.

However, this approach was successful for the solution of relatively simple problems, with the growing complexity a more organized way of thinking and engineering was needed. A key tool that enables the increase of design, solution, or component reusability is the concept of patterns that have been used earlier in the field of architecture and civil engineering [AIS77b].

A (solution) pattern describes the core of a solution for a problem in a given context in such a way that it can be applied repeatedly without precise repetition ever [AIS77b]. Patterns can be used in different layers of software and systems engineering. We will focus on design patterns that are used in the design phase.

Design patterns usually describe how several elements of some system cooperate to accomplish a certain task. For instance, the Observer pattern (e.g., [GV95]) describes how objects (i.e., the Observers) can be informed and updated whenever some Subject changes in a flexible, object-oriented way. Essentially, this is achieved by providing a Subject-interface, where Observers can (de)register themselves, and an Observers-interface, through which the Subject can inform registered Observers about changes. Subject and Observer are called roles of the pattern, which on pattern application are taken over by specific objects.
Therefore, several constituents of a pattern can be distinguished:

- Roles: the structural or behavioral elements; often denoting code (fragments)
- Parameter: on application, are bound to roles or control other pattern aspects
- Rules: describe how the pattern has to be applied, and what has to be considered.

Since [GV95], for which the authors are still nicknamed as Gang of Four (GoF) due to its continuing influence, hundreds or thousands of patterns have been described in books like [BMR+96] or proceedings of pattern conferences. Already in 2000, the pattern almanach [Ris00] listed about thousand patterns and variants in over 68 categories.

Most of the patterns represent generic solutions, and are specified over standard Unified Modeling Language (UML). There are, however several Domain-specific languages (DSLs) that also need support for the design of system models. While most of the modern UML tool environments support the definition and application of design patterns, this is not the case for DSL tools.

1.1.2 Component-based systems

Component-based Software Engineering (CBSE) [HC01] is an answer to the new challenges of software development. This approach uses components as building blocks in order to assemble complex systems.

**Definition 1 (Component)** A component is a self contained, reusable entity that encapsulates a specific functionality (and/or data), and communicates with other components via explicitly defined interfaces.

Several industrial-grade component-oriented frameworks are available, for instance COM, COM+, .NET from Microsoft [Rog97, Mic], Enterprise Java Beans (EJB) [Pro, BMH06], OSGi [Allb], and Eclipse [Foue] in the Java community.

The above examples show that the component-oriented principle does not try to change the implementation paradigm, as there are component frameworks for object-oriented, module-based, and structured languages. Component orientation is a higher level concept that supports the partitioning of the solution and to achieve better separation of concepts by explicit definition of component boundaries and interactions.

Syntactical aspects of component interfaces and links are usually well-defined in the component frameworks, but there are limitations if it comes to expressing semantics, or Quality-of-Service (QoS) attributes. This limitation is a key issue during the development of dependable systems, where a) the semantic correctness should be evaluated, b) QoS values should be calculated based on component characteristics. In order to overcome these limitation, several frameworks have been developed. BIP [BBS06] is a formal component-oriented language for real-time systems design. Several research papers (like [KSS05, KSS07]) propose the utilization of semantic web technologies for the specification of the semantic aspect of components. There are promising results, but up to now, none of these got mainstream in the industry.

In our model, software components interact only using dedicated Linking Interfaces (LIFs). These interfaces represent synchronous or asynchronous ports. Besides the LIFs, components may also use non component-oriented elements as operating system or middleware services. From the software architecture modeling point of view, only the LIF interfaces are modeled, and the Local Interface (LF) interfaces are handled only at implementation level. Component implementations can also have control interfaces that are used by the execution environment to control the life-cycle of the components (start/stop/reset), or to configure them.

Figure 1.1: Stateful Software Component Model

Figure 1.1 illustrates the conceptual model of a software component, containing only LIFs, and an internal state. The internal state of the component represents the actual memory content of the component containing all state information about the component. Input LIFs receive inputs that (together with the internal state) control the behavior of the component that results in output messages. More formally:
Definition 2 (Stateful component) A stateful component \( \text{Comp} \) is a tuple: \( \text{Comp} = (\text{In}, \text{Out}, \text{State}, \text{f}_{\text{behavior}}) \), where

- \( \text{In} \) is the input LIF,
- \( \text{Out} \) is the output LIF,
- \( \text{State} \) is the internal state, and
- \( \text{f}_{\text{behavior}} : (\text{In}, \text{State}) \rightarrow (\text{Out}, \text{State}) \) is the data transformation executed by the component’s internal behavior.

Management of stateful components is a complex problem from the execution environment point of view and (as the internal state is hidden) it also hinders the diagnosis and monitoring of the system state. In order to overcome these problems, a modified component model is proposed by [Kop07] (illustrated by Figure 1.2).

![Figure 1.2: Software Component Model with Externalized State](image)

Definition 3 (Stateless component) A stateless component \( \text{Comp} \) is a tuple: \( \text{Comp} = (\text{In}, \text{Out}, \text{f}_{\text{behavior}}) \), where

- \( \text{In} \) is the input LIF,
- \( \text{Out} \) is the output LIF, and
- \( \text{f}_{\text{behavior}} : (\text{In}) \rightarrow (\text{Out}) \) is the data transformation executed by the component’s internal behavior.

The new model has no internal state, as all state information is externalized to the extra-component state store. The store is implemented as a stable storage that preserves the information between component executions. This solution has several advantages:

- The component can be restarted or moved between computational cores as its state is handled by the execution environment.
- The externalized state can be monitored for debugging or diagnosis purposes.
- The externalized state can be backed up in case of system failures and used at restart.
- The externalized state can also be propagated using messages. That way, multiple replicas of the component can vote on the current state. If a replica contains incorrect state information (for instance after replica restart), this can be replaced by the correct one.

Besides these advantages external state processing and propagation causes both processing and communication overhead so it is mainly used in safety-critical (sub)systems, while non safety-critical (sub)systems may use stateful components.
1.1.3 Distributed systems

With the spreading of networked computers, a new type of systems has been evolved. Distributed systems are composed of components running on different computing nodes that interact with each other via network links.

Distribution of system components can serve different goals:

- **Parallel execution** of different tasks can increase the throughput of the system.
- **Physical separation** of nodes (together with other dependability related techniques like replication) can result in better dependability in case of environmental problems (fires, floods, power outages).
- **Maintenance** of the system is easier, as if a unit fails, it can be usually replaced on field, and the system can be recover/restart with the new unit.
- In some cases (mainly in the embedded systems field) the controlled physical entity (industrial process, etc.) requires the distributed allocation of sensors and actuators, and thus also control nodes.

Each main component-oriented framework introduced earlier enables the development of distributed systems, by offering an inter-node communication infrastructure that hides the communication details from the components (and the component developers). Components can be deployed to separate hosts and the links will be built up runtime. In general, it can be stated that a well-designed communication middleware does not change the functional behaviour of the system.

In contrast to basic functional properties, QoS attributes of composite components and systems can significantly be affected by the fact that the components communicate via each other using an underlying network infrastructure. On one hand, distribution incorporates several factors like communication delays in the temporal domain and new error sources (communication infrastructure elements) in the dependability domain. On the other hand, if the middleware supports dependability related services like replication, state migration, voting, etc. the distributed implementation can also result in better dependability of the system. It can be stated, however, that these advantages only appear if the developer is aware of the fact that the system will be deployed as a distributed one. This results in a more strict, design time deployment planning, in contrast with runtime, ad-hoc allocation available in general frameworks.

1.2 Model-driven development

1.2.1 MDA

The Model-Driven Architecture (MDA) is a specification of the Object Management Group (OMG) [Grof]. MDA starts with the well-known and long established idea of separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform.

![Figure 1.3: Model Driven Architecture overview](image)

MDA provides an approach for:
specifying a system independently of the platform that supports it,

specifying platforms,

choosing the most suitable platform for the system under consideration, and

transforming the system specification into one for a particular platform.

The primary goals of MDA are:

- reduced time-to-market,
- productivity
- portability,
- interoperability,
- reusability.

The main concepts of MDA are (excerpts from the specification):

- **system** - A system may include anything that exists or is planned to exist: a program, a single computer system, some combination of parts of different systems, a federation of systems, each under separate control, people, an enterprise, a federation of enterprises.

- **model** - A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language.

- **application** - An application is a functionality being developed. A system is described in terms of one or more applications supported by one or more platforms.

- **platform** - A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.

- **platform-independent model** - A Platform-independent Model (PIM) is a view of a system from the platform independent viewpoint. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type.

- **platform-specific model** - A Platform-specific Model (PSM) is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform.

- **platform model** - A Platform Model (PM) provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform. It also provides, for use in a platform specific model, concepts representing the different kinds of elements to be used in specifying the use of the platform by an application. A platform model also specifies requirements on the connection and use of the parts of the platform, and the connections of an application to the platform.

- **model transformation** - Model transformation is the process of converting one model to another model of the same system.

- **implementation** - An implementation is a specification, which provides all the information needed to construct a system and to put it into operation.

MDA is said to be model-driven because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification. It suggests using different models at the different levels of abstraction and at different phases of development during system development. The process of development will consist of the composition and refinement of models. Transformations serve as means of progressive refinement of models from abstract, platform independent, requirement centric towards concrete, platform specific, implementation centric.
1.2.2 Metamodeling

Metamodeling is a methodology for the definition of modeling languages. A metamodel specifies the abstract syntax (structure) of a language. Metamodels are expressed using a metamodeling language that itself is a modeling language. The metamodel can also be interpreted as the object-oriented data model of the language under design.

There are several different metamodeling environments, like the Meta Object Facility (MOF) [Groe] from OMG, ECore that is a subset of MOF and is a component of the Eclipse Modeling Framework (EMF) [Foua], and Visual and Precise Metamodeling (VPM) [VP03] that has been developed at Budapest University of Technology and Economics. It should be noted, that even if these environments support the definition of metamodels, this should be complemented by additional elements in order to achieve a well defined language.

- Definition of language constraints is typically done using constraints languages.
- Definition of concrete syntax (graphical and/or textual) that will be the interface to the users of the language.
- Definition of model exchange format that enables the efficient serialization of the models.
- Definition of language semantics is a key issue in creating formal languages, but is often neglected in the practice.

It should be noted, that ontologies are important complementing technologies during language design as they can be used to define and analyze the concepts and relationships of the language. Ontology techniques help in finding potential language design problems like contradictory relations or invalid inheritance hierarchies.

1.2.3 UML

UML is a standard of the OMG [Grom]. UML is a visual language for specifying, constructing and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains and implementation platforms. During the last few years UML has emerged as the software industry’s dominant modeling language. It is widely accepted among system designers, analysts and programmers. The UML specification is defined using a metamodeling approach that adapts formal specification techniques. While this approach lacks some of the rigor of a formal specification method, it offers the advantages of being more intuitive and pragmatic for most implementors and practitioners.

An important aspect of UML is that besides being general-purpose, it also offers an extension facility that is called UML profile mechanism. Using this feature domain-specific modeling concepts can be introduced in the UML modeling environment that results in a more compact model that is easier to understand for domain experts. Several standard profiles are present that address specific domains like embedded systems, enterprise applications, and so on.

1.2.4 DSL

However, industrial practice has indicated, that general-purpose visual modeling languages, like the core UML are hard to use, as they enforce their own logic onto the designer independently off the traditions of the specific application domain. Domain-Specific Modeling (DSM) languages evolve rapidly in order to support a better expressiveness, familiarity and reuse of the application-specific notions and solutions and resulting in compact, easy-to-understand models for the designers.

DSM environments provide sophisticated graphical editors to domain experts during the design phase to capture precise models at a high level of abstraction. However, there exists no complete solution covering all the phases of the creation and use of DSM technologies.

1.2.5 Visual Programming Languages

A specific subset of domain-specific languages is called visual programming languages instead of modeling languages. These languages are low level but visual tools for specific target application domains and combine the advantages of visual representation with low level constructs. There are several languages used in embedded systems like FlowCode [Tecb] targeting single micro-controller systems, Dron [Rom] targeting space applications, and LabView [Ins] for measurement, data acquisition, and control.
1.3 Formal analysis methods

Formal analysis techniques are widely used in order to verify and validate system models or implementations. We distinguish two main categories of analysis methods: qualitative and quantitative approaches.

Quantitative methods are based on mathematical foundations and analyze the system using numerical metrics in order to express some property of the system. Typical investigation aspects include timing analysis, dependability-related metrics, or power consumption.

Qualitative methods specify systematic methods for analysis that help to identify potential design problems like single point of failures. It should be noted that there are several analysis modeling approaches that have both quantitative and qualitative evaluation possibilities.

1.4 Objectives of the research

In the current thesis, I present methods and techniques based on the model-driven paradigm that improve the productivity and quality of dependable systems development processes.

The objectives of my work were the following ones:

- **Objective 1** Improve the productivity of systems development by introducing interactive, semi-automated tools and methods in different phases of the development process.

- **Objective 2** Reduce cognitive complexity of system designs by separating the architecture and behaviour of applications from the details of the implementation platform (as facilitated by the principles of Model-Driven Architecture approach).

- **Objective 3** Improve the quality of system designs by the introduction of different consistency checks, analysis, and synthesis methods.

- **Objective 4** Define a model-driven hardware-software integration methodology that is capable of handling complex target architectures and several non-functional aspects like timeliness, dependability, and cost.

During the research work it became clear that the current model management and model transformation techniques need to be extended in order to achieve better productivity and reusability during the development of tools and techniques supporting the model-driven development principle. This led to the following objectives:

- **Objective 5** Provide advanced language constructs that support the reusability of model transformations.

- **Objective 6** Provide support for the executable specification of design patterns that allow the integration of best practices and common solution patterns into domain-specific modeling environments.

- **Objective 7** Provide support for the integration of different metamodeling environments using a precise semantical foundation.

Although I focused on the development of dependable (embedded) systems, the model-driven tool and methodology concepts are applicable in other domains as well.

The current thesis aims at the fulfillment of these objectives by a) proposing novel techniques for the model transformation and metamodeling environments that provide support for the efficient development and integration of model-driven tools, and b) by providing interactive, semi-automated, model-transformation based design environment for dependable embedded systems that features modeling, model integration, analysis, and optimization-based synthesis techniques.

1.5 Structure of the paper

The current thesis consists of seven main chapters (including this introduction) that contain new results and four appendices complementing the main parts with additional information.

- Chapter 2 gives a brief introduction of the main concepts of model-driven development including metamodeling environments and model transformation tools.
• In chapter 3, I extend the existing graph transformation concepts with several constructs that enable the reuse and integration of transformations in order to achieve better productivity during transformation development and better performance at transformation runtime.

• Chapter 4 introduces the current trends and challenges in the embedded systems domain.

• Chapter 5 presents a novel, quality-of-service aware, interactive approach for hardware-software integration complemented by an early verification and validation framework.

• Chapter 6 presents precise analysis and optimization-based synthesis methods that can be utilized during different phases of the model-driven development process.

• Finally, Chapter 7 concludes the main part of the thesis.

• Appendix A summarizes the metamodels introduced in Chapter 5.

• Appendix B contains the GTASM implementation of all algorithms discussed in Chapter 5.

• Appendix C contains algorithms and models from Chapter 3.

• Appendix D contains a step-by-step example of a PIM-PSM mapping tool usage scenario.
Chapter 2

Model-driven Development

In this chapter we will give more details on the foundations of model-driven development methodology and techniques that will be used throughout this paper.

2.1 Development workflow

The introduction of DSM technologies necessitates several language and supportive technologies design and implementation steps (see Figure 2.1). Language design serves for the creation of the modeling semantics and syntax corresponding to the particular application domain, typically by specializing an existing high-level language to the specifics of the target field. For instance, the host language is typically either native UML or some of its specific dialects covering a broad class of applications like OMG Systems Modeling Language (SysML) for embedded systems providing the general notion for the management of physical entities. Modern language and tool design technologies have the language metamodel as their core concept describing the individual modeling notions and their mutual relationship.

From the point of view of quality assurance of the language design process, the checking of the metamodel containment is a crucial step. In this step, the appropriateness of the specialization of the general language has to be checked for compliance and compatibility. There are approaches that use the fundamental relationship between ontologies (describing the hierarchies of concepts and their relationship) and metamodels (describing the notions and their mutual relationship). In order to check the metamodel - metamodel compliance between the high-level general purpose language and the derived DSM, these approaches transform both metamodels into ontologies [22] and use an off-the-shelves checking engine for compliance checking. It is worth noting that when performing such a formal check on a carefully designed DSM, which already passed several round of peer reviews done by experts, numerous minor flows like incompleteness or inconsistencies in the refinement notions and relationships were detected; moreover, missing refinements triggered new ideas on the content of the modeling language.

The modeling environment adaptation is nowadays widely supported by the industry. For instance,
Eclipse offers several packages for the introduction of user defined DSMs and their visual representations resulting in DSM customized visual editors. The user gets full support in the creation of his application model in the DSM environment. Finally, synthesis transformation design aims at the generation of the runtime code and configuration from the platform independent model as described by the user.

There are several cross-cutting techniques that are present in all phases of the design process. Traceability of requirements through various modeling artifacts to implementation is a requirement in most dependable system design context. This implies that the design environment should be able to track transformations, and establish traces, or reference models between the artifacts of the process.

Requirements management is closely correlated with traceability, and plays important role throughout the development process. Requirements are collected and managed using dedicated tools and methodologies. It should be noted, that several standard languages, like UML and SysML, have support for requirements elicitation and traceability, although this support is not always complemented by proper tool support. The key goal, from the DSM environment point of view, is the organization of requirements, the support of traceability, and support of verification and validation of the system models against the requirements.

An other important cross-cutting issue is the support of reuse in all phases of the design process. This can be achieved in a model-driven development environment by a) giving support for component libraries from where existing model fragments can be imported, and b) with the support for design pattern definition and application.

We can can state based on this overview that metamodeling (language design) and model transformation design play an important role in the development of domain-specific tool environments and methodologies. We will give a brief overview on the state-of-the-art technologies in the next Sections.

2.2 Metamodeling Environment

Metamodeling is the methodology for the definition of modeling languages. There are several approaches that are used widely by the industry and also by the academica. We will introduce the most important approaches in this section, and give a unified view of them using a formal model.

**Definition 4 (Metamodels and models)** Let $\mathcal{M}_{\text{Language}}$ denote the metamodel of Language, and $\mathcal{M}_m$ denote a model $m$. $\mathcal{M}_m \xrightarrow{\text{instanceOf}} \mathcal{M}_{\text{Language}}$ denotes that the model is an instance of the language.

### 2.2.1 Meta Object Facility

MOF [Groe] is a standard of OMG. Is serves as a basis for UML and the related languages. The intended goal of MOF development was the definition of a modern, platform-independent metamodeling framework that’s concepts can be easily mapped to different technologies.

MOF defined originally four meta-levels for modeling that were separated strongly from each other that was a main blocker in the utilization of MOF in several modeling contexts [VP03]. The new (2.0) version of the specification, however, states that “Suffice it to say MOF 2.0 with its reflection model can be used with as few as 2 levels and as many levels as users define” resulting in more freedom for the language developers.

MOF has a core sublanguage called Essential MOF (EMOF) that defines the minimal number of concepts that are required for the building of metamodels. This core set has been inspired by an open source project (EMF) [Foua] that aimed at the development of a metamodeling framework for the Eclipse tool platform. A subset of the MOF language has been used during the implementation of the ECore modeling core that proved to be sufficient for practical modeling scenarios.

Figure 2.2 illustrates the concepts of EMOF. The central element is the Class that can have Properties that can be typed by either primitive types (defining a simple value slot), or classes (defining an inter-class reference). References are uni-directional, but they can be paired using the opposite relation. There are several auxillary concepts that are not present in this diagram. Packages can be used to group elements and enumerations are special data types.

MOF is used widely in the standards of OMG and also by researchers and by the industry to define modeling languages of various scope. There is, however, a strong drawback of the approach: it lacks formal semantics. Although there are several papers (for instance [Sch06]) that suggest semantics for MOF, none of these has been accepted widely. OMG issued a Request for Proposal for the next generation of MOF-related standards, that on one hand includes a request for a semantic layer for MOF, and on the other hand also ontology mapping capabilities, that both transform MOF in a language that will be closer to the semantic technologies.
2.2.2 ECore

As mentioned earlier, ECore is the metamodeling language that is used by EMF. It has been developed in order to provide an approach – simpler than MOF – for metamodel definition that supports the direct implementation of models using a programming language. The main rationale in introducing ECore separately that it is the de facto standard metamodeling environment of the industry, and several domain-specific languages are defined using this formalism. ECore is nearly identical with the EMOF package of MOF, so the ECore elements will only covered shortly.

- **EClass** models classes (or concepts). EClasses are identified by name and can have several attributes and references. To support inheritance, a class can refer to a number of supertype classes.

- **EAttribute** models attributes, that contain data elements of a class. They are identified by name, and have a data type.

- **EDataType** is used to represent simple data types that are treated as atomic (their internal structure is not modeled). Data types are also identified by their name.

- **EReference** represents a unidirectional association between EClasses and as identified by a name. Lower and upper multiplicities can be specified. It is also possible to mark a reference as containment that represents composition relation between elements. If a bidirectional association is needed, it should be modeled as two EReference instances that are mutually connected via their opposite references.

The rest of the ECore metamodeling language contains utility elements, common supertypes that support the organization and hierarchisation of the models but the main metamodeling part has been covered here.

2.2.3 VPM: Visual and Precise Metamodelling

The VPM metamodeling approach ([Var04], [PV05]) will be introduced in this section based on the literature. VPM will be used throughout this paper in order to formalize metamodels, model queries, constraints, and transformations using this mathematically precise formalism.
2.2.4 VPM modeling concepts

Figure 2.3 illustrates the metamodel of the VPM approach. VPM uses different naming convention as MOF in order to avoid clashes between elements of both approaches.

A model element in VPM may be either an entity, a relation, or a function. A unique identifier (denoted by a .id postfix in the sequel) and a set including the identifier of the model element and the identifiers of all the (intended) refinements of the element (accessed by the .set postfix) are related to each of this constructs.

In order to express instantialization between model elements (type-instance relations) the instanceof relation is defined between model elements. Similarly, the supertype relation expresses inheritance between model elements. It should be noted, that (in contrast to MOF), VPM also support the later construct between relations and functions (association and attribute inheritance).

VPM contains similar concepts that of the ontology languages (for instance, Web Ontology Language (OWL) [Cone]). Ontologies also contain entities (or classes) that are related to each other via properties, and support inheritance between these elements. Although ontologies do not support the introduction of arbitrary number of meta levels, this can also be simulated by special relations. It should be noted, however, that the comparison of expressiveness of ontologies and VPM (or ontologies and metamodeling environments in general) is still an open problem.

![Figure 2.3: The MOF metamodel of the VPM approach](image)

- An entity is a set (called basic entity in this case) or a tuple (detonated as compound entity or model) consisting of sets, relations, functions, and tuples (a collection of entities, connections, and mappings, respectively).

  Entities will be represented either as UML classes or UML packages while the notion of containment will be captured by graphical containment (class inside a package), or by aggregations (leading from entities to other model elements) depending on the context.

- A relation between two entities is a binary relation between the associated sets or tuples. Relations are depicted as (directed) UML associations.

- A function from entity $E_1$ to entity $E_2$ is a function with the domain of $E_1$ and range of $E_2$. Functions can be denoted visually by an attribute assigned to entity $E_1$ and a type corresponding to $E_2$.

A more detailed and precise description of the metamodeling constructs can be found in [Var04].

2.2.5 Algebraic representation of models

Algebraic representation of models is used as the underlying data model for queries, transformations, and model constraints. The models can be represented on meta level using a generic approach, or on model level using a meta model specific formalism. In order to point out the difference between the two representations, let us consider two entities Node and $n_1$, where $n_1$ is an instance of Node.

- The meta-level representation uses the formula $entity(Node) = true$, $entity(n_1) = true$, and $instanceOf(n_1, Node) = true$ to express this relationship.
2.3. MODEL TRANSFORMATIONS

- The model-level representation introduces the function symbol `node` and therefore uses the `entity(Node) = true` (the Node metaclass is an instance of Entity), `entity(n1) = true`, and `node(n1) = true` (n1 is an instance of Node and Entity as well) to express the same relationship.

Formally, the meta level Vocabulary of VPM contains the following elements:

- `undef, true, false` for the undefined symbol and elementary logic constraints
- `entity/1`, unary function for entities
- `relation/3, function/3`: tertiary functions for connections (relations) and mappings (functions)
- `supertypeOf/2, instanceOf/2, componentOf/2` for inheritance, instantiation, and containment between basic modeling concepts.

2.2.6 VPM and MOF

Although MOF [Groe] is the most widespread meta modeling standard, it has several limitations from our point of view.

- Lack of formal semantics. Our goal is to give precise definitions on metamodeling and model manipulation levels, therefore a precise semantics for the metamodeling framework is needed.
- Lack of precise query and manipulation language. Although OMG introduced the Queries, Views, and Transformations (QVT) [Grod] specification that defines a query and transformation language for MOF based modeling languages, the new standard still does not have precise semantics for the query and transformation description.
- Structural redundancies of MOF. MOF (and even EMOF) has multiple constructs that express the same modeling concepts that can lead to ambiguous descriptions. As we want to keep the definitions clear and unambiguous a more precise description should be chosen.

We only focused on problems that directly relate to the current paper. It should be noted that a more thoroughly comparison has been done in [VP03].

2.2.7 Mapping MOF and ECore to VPM

The mapping of the MOF and ECore metamodels to the VPM model space will be described in this section. This will provide a common, formal framework for the integration of modeling languages defined on top of these metamodeling environments.

The mapping utilizes the fluid meta level support of the VPM approach. The metamodels of the ECore and MOF languages are defined as instances of the VPM core elements (see Figure 2.4), forming a metamodeling environment meta level. Modeling languages are instances of either one of these metamodels, or directly of the VPM core, forming the metamodel level. Metamodels and instance models share the same query and manipulation operations and are represented in the same environment allowing the seamless integration and transformation of them.

The only step that is needed for integration of these approaches is the generation of ECore and MOF metamodels as VPM metamodels. This is a straightforward mapping; therefore the details will not be discussed here, but Appendix C.2 contains the complete ECore metamodel, and Appendix C.3 contains the MOF metamodel.

2.3 Model transformations

The crucial role of model transformation (MT) languages and tools for the overall success of model-driven system development have been revealed in many surveys and papers during the recent years [CH03, MG06]. To provide a standardized support for capturing queries, views and transformations between modeling languages defined by their standard MOF metamodels, the OMG has issued the QVT [Grod] standard.

QVT provides an intuitive, pattern-based, bidirectional model transformation language, which is especially useful for synchronization kind of transformations between semantically equivalent modeling languages. The most typical example is to keep UML models and target database models (or UML models and application code) synchronized bidirectionally during model evolution.
However, there is a large set of model transformations, which are unidirectional by nature, especially, when the source and target models represent information on very different abstraction levels (i.e. the model transformation is either refinement or abstraction). Unfortunately, the current UML Mapping Language is far less intuitive and easy-to-use in case of unidirectional transformations. Even those MT approaches that aim at implementing the standard (like ATL [ATL]) have chosen a more intuitive language for such transformations.

Graph transformation (GT) [EEKR99], which provides a rule and pattern-based specification paradigm for the manipulation of graph models, is frequently used for specifying model transformations within and between modeling languages. When executing a GT rule on an instance model, a matching of the left-hand side (LHS) graph pattern is substituted by an image of the right-hand side (RHS) pattern. Since the early 1990s, a wide range of tool support has become available, e.g. VMTS [LLCO4], PROGRES [SWZ99], FUJABA [FNTZ00], AGG [ERT99], GReAT [KASS03], ATOM3 [dLV02], TefKat [TUoQ06], VIATRA2 [7].

On the one hand, graph transformation tools are able to handle industrial size models [VSV05] for practical model transformation problems. However, GT specifications can easily become large and scattered in case of complex model transformations due to the limited support of composability in graph transformation languages. Most typically, the graph patterns in GT rules become large (consisting of a large number of pattern elements), and have often common partitions reused in many pattern.

The following notation will be used in this paper to denote transformations:

Definition 5 (Model transformation notation)

\[ \mathcal{M}_{\text{Src}} \xrightarrow{\text{tr}_1} \mathcal{M}_{\text{Trg}} \]

denotes that the model $\text{Src}$ will be transformed to model $\text{Trg}$ by transformation $\text{tr}_1$. Similarly, on metamodel level:

\[ \mathcal{M}_{\text{Src}} \xrightarrow{\text{tr}_1} \mathcal{M}_{\text{Trg}} \]

denotes that the transformation $\text{tr}_1$ maps from metamodel $\text{Src}$ to metamodel $\text{Trg}$.

The model transformation formalism of the VIATRA2 framework will be introduced in this section. We will use this compact, yet precise notation in the following chapters to describe model queries and manipulations.

2.3.1 Graph patterns

VPM patterns are defined in order to capture the concept that will be used in both model queries and manipulations (graph transformation steps). As defined in [PV05], a pattern is a composite entity containing
entities \( \mathcal{M}_P \) and \( \mathfrak{M}_P \), where \( \mathcal{M}_P \xrightarrow{\text{instanceOf}} \mathfrak{M}_P \).

Informally, a pattern is a graph where nodes are entities, and edges are relations and functions. All elements are instances of a meta element that is contained in the metamodel.

**Pattern Representation**

Based on the previous definitions, a VPM pattern is a pair of compound entities and can be described using the basic VPM vocabulary. The pattern is composed of the functions defined by the vocabulary and of Boolean logic functions.

In order to be able to compare the various representations, we will use a simple example. Let’s assume that we have a simple class diagram metamodel (see Figure 2.5) that will be the \( \mathfrak{M}_P \) pattern for the example. The \( \mathcal{M}_P \) pattern will define the class-to-property relationship (a class with an attached property). Using the standard description the following formula defines the \( \mathfrak{M}_P \) component of the pattern:

\[
\begin{align*}
\text{entity}(\text{Class}) &= \text{true} \\
\text{entity}(\text{Property}) &= \text{true} \\
\text{entity}(\text{String}) &= \text{true} \\
\text{relation}(\text{attr}, \text{Class}, \text{Property}) &= \text{true} \\
\text{relation}(\text{type}, \text{Property}, \text{Class}) &= \text{true} \\
\text{function}(\text{name}, \text{Class}, \text{String}) &= \text{true} \\
\text{function}(\text{name}, \text{Property}, \text{String}) &= \text{true}
\end{align*}
\]

And the following formula defines the \( \mathcal{M}_P \) component:

\[
\begin{align*}
\text{entity}(C) &= \text{true} \\
\text{instanceOf}(C, \text{Class}) &= \text{true} \\
\text{entity}(P) &= \text{true} \\
\text{instanceOf}(P, \text{Property}) &= \text{true} \\
\text{relation}(A, C, P) &= \text{true} \\
\text{instanceOf}(A, \text{attr} = \text{true}) &= \text{true}
\end{align*}
\]

Although this representation is precise, it is hard to read. Therefore, we will define two alternative representations in this Section.

**Compact textual representation**

The previous description can be compacted using the Viatra2 Textual Command Language (VTCL) and Viatra2 Textual Modeling Language (VTML) notation. These languages support the compact textual representation of meta- and instance models, and graph transformations.

The previous meta model can also be described using the compact notation:

```plaintext
entity(\text{Class});
entity(\text{Property});
entity(\text{String});
relation(\text{attr}, \text{Class}, \text{Property});
relation(\text{type}, \text{Property}, \text{Class});
function(\text{name}, \text{Class}, \text{String});
function(\text{name}, \text{Property}, \text{String});
```

The description is similar to the original, but it omits the \( = \text{true} \) clauses, and uses simple concatenation instead of the logical \text{and} operator.

The compact pattern description of the \( \mathcal{M}_P \) that uses the previous model definition is the following one:

```plaintext
pattern \text{classWithProperty}(C, P) = {
    \text{Class}(C);
    \text{Property}(P);
    \text{attr}(A, C, P);
}
```

The pattern description uses the model-level vocabulary, and the pattern body is enclosed in a graph pattern definition. The definition contains the name of the pattern and the list of pattern parameters. This defines the controllable and observable subset of all pattern elements (or variables). More formally, the pattern signature defines the following projection:

**Definition 6 (Graph pattern variable quantification)** Let \( P \) be a pattern that contains \( V \) set of variables. If the \( S_P \) pattern signature contains only the \( U \subseteq V \) set of parameters, the following quantification should be applied:

\[
S_P(U) = \exists_{\forall i} V_i : V_i \not\in U \land P(V)
\]

If the projection is used, the non-visible variables will have an implicit existence quantification. Informally this means that if a correct substitution is given through the signature variables, and there exists a substitution for the internal ones, the pattern will hold on the current model.
Graphical representation

In order to increase the readability of pattern descriptions we will also use a graphical representation that is a one-to-one mapping of the previous model and pattern descriptions. The notation is illustrated by Figure 2.5.

![Figure 2.5: Sample metamodel and pattern representation](image)

2.3.2 Additional pattern constructs

In order to increase the expressiveness of the graph patterns, several extensions have been defined in the literature. Two of these, that will be introduced in the current Section, will be used extensively in this paper.

Negative application conditions

A negative application condition is a sub-pattern (a pattern embedded into an other pattern) that represents prohibitive constructs. Formally:

**Definition 7 (Negative application conditions)** A pattern that contains a basic pattern and a set of negative application conditions so that \( \forall N \in N \), \( N_i \) is also a pattern, represents the following formula (\( \neg \) denotes negation of \( V \)):

\[
P = P_i \land \neg \bigvee_{\forall N \in N} N_i
\]

The pattern holds on the model if the basic (positive) part holds, and none of the negative application conditions is fulfilled.

Additional constraints

Besides the topological constraints expressed by the basic pattern constructs, additional logic constraints (called check constraints) may be necessary in order to do attribute value checking, or to express other mathematical/logical constraints.

**Definition 8 (Additional attribute constraints)** A check constraint Check on a pattern \( P \) is a logic formula on the set of pattern variables \( V \). In this case:

\[
P_{\text{check}} = P \land \text{Check}
\]

The pattern holds only, if both the basic pattern and the constraints are fulfilled.

Textual and graphical notations

In order to illustrate the textual and graphical notation on the new constructs, the previous example will be extended with the following criteria:

- The pattern should only match properties having a type different from the containing class.
The pattern should only match attributes having a name different from identifier.

The first criteria will be implemented using a negative application condition, while the second one will be a check constraint. The VTCL description is as follows:

```vtcl
pattern classWithProperty(C,P) = {
  Class(C);
  Property(P);
  attr(A,C,P);
  String(PName);
  name(N,P,PName);
}

neg pattern nac1(C,P) =
{
  Class(C);
  Property(P);
  type(T,P,C);
}
check PName !=; 'identifier'
```

The basic pattern is extended with the name recognition part, and the check constraint (that is a bool formula after the check keyword). As the negative application condition is also a pattern, it has (after the neg) keyword the same signature and internal structure as the main pattern. As the negative pattern should also be a valid graph, it repeats some of the elements of the basic one.

The graphical representation of the pattern (Figure 2.6) is more compact. The check constraint is a simple annotation, while the negative application condition is depicted as a red rectangle. In the graphical notation, syntactic sugaring is applied: the common elements of the basic and negative patterns are not repeated.

### 2.3.3 Model Transformations

VIATRA2 [2] has chosen to integrate two popular, intuitive, yet mathematically precise rule-based specification formalisms, namely, graph transformation (GT) [EEKR99] and abstract state machines (ASM) [BS03a] to manipulate graph based models. VIATRA2 relies on the VPM approach regarding model representation. The language that is result of the integration of these approaches (VPM, GT, ASM) is called VTCL (VIATRA2 Textual Command Language).

**Basic ASM Constructs**

To control the execution order and mode of graph transformation the language includes language constructs that support the definition of complex control flow. As one of the main goals of the development of the notation was to create a precise formal language, the basic set of Abstract State Machines (ASM) language
constructs [BS03a] that correspond to the constructs in conventional programming languages have been included.

The basic elements of an ASM program are the rules (that are analogous with methods in OO languages), variables, and ASM functions. ASM functions are special mathematical functions, which store values in associative arrays (dictionaries). These values can be updated by ASM rules.

The language also allows the usage a special class of functions, called native function that are user-defined Java methods that can be called from the transformations. These methods can access any Java library (including database access, network functions, and so on), and also the VPM model space. This allows the implementation of complex calculations during the execution of model transformations. ASMs provide complex model transformations with commonly used control structures in the form of built-in ASM rules, which are illustrated by Figure 2.7.

```
// variable definition
let X = 1 in ruleA
// variable (and ASM function) updates
update X = X + 1;
// print and log rules print a term to standard output or into the log
print("Print X: " + X + 
\n");
log(info, "Log X: " + X);
// conditional branching by a logical condition or by pattern matching
if (X>1) ruleA else ruleB
if (find myPattern(X)) ruleA else ruleB
// exception handling: rule2 is executed only if rule1 fails
try rule1 else rule2
// calls the user defined ASM rule myRule with actual parameter X
call myRule(X)
// the sequencing operator: executes its subrules in the given order;
seq { rule1; rule2; }
// executes a non-deterministically selected rule from a set of rules
random { rule1; rule2; }
// iterative execution by applying rule1 as long as possible
iterate rule1;
// executes rule1 for a (non-deterministic) substitution of variable X
// which satisfies the pattern (or location) condition with X
choose X below M with (myAsmFun(X) > 0) do rule1
choose X below M with find myPattern(X) do rule1
// pseudo-parallel execution of rule1 for all substitution of variable X
// which satisfies the pattern (or location) condition with X
forall X below M with (myAsmFun(X) > 0) do rule1
forall X below M with find myPattern(X) do rule1
```

Figure 2.7: Overview of built-in ASM rules

As a summary, ASMs provide control structures including the sequencing operator (seq), rule calls to other ASM rules (call), variable declarations and updates (let and update constructs) and if-then-else structures, non-deterministically selected (random) and executed rules (choose), iterative execution (applying a rule as long as possible iterate), and the deterministic parallel rule application at all possible matchings (locations) satisfying a condition (forall).

Model manipulation

In addition to these core ASM rules, this dialect of ASMs also includes built-in rules for manipulating the model space. As a result, elementary model transformation steps can be specified either in a declarative way (by graph transformation rules) or in an imperative way by built-in model manipulation rules. The main set of model manipulation rules are summarized in Fig. 2.8.

Most of these rules are rather straightforward, we only give more insight into the copy rule, which aims at copying an entity and all the recursive contents (i.e. the subtree of the entity) to a new parent. Viatra2 provides two kinds of semantics for that copy operation: keep_edges and drop_edges. In the first case, all relations leading out from or into an entity placed anywhere below the copied entity are copied as well. In
2.3. MODEL TRANSFORMATIONS

// create a new entity C of type class and place it inside M
new (class(C) in M);
// rename class C to "Product"
rename(C, "Product");
// create a new relation Attr of type attrs between C and A
// Attr is placed under its source C
new (attrs(Attr, C, A));
// Explicitly moves entity C (and all of its contents) to NewContainer
move(C, NewContainer);
// Retargets relation Attr to A1
setTo(Attr, A1);
// copy entity C and all of its contents directly under Container together with
// ALL incoming and outgoing relations; the entity is accessed by CNew
copy(C, Container, CNew, keep_edges);
// copy entity C and all of its contents directly under Container but
// only copy relations between entities of the containment subtree of C
copy(C, Container, CNew, drop_edges);
// removes model element M together with its contents
delete(M);

Figure 2.8: Overview of model manipulation rules

the latter case, only those relations are copied where both the source and the target entity is below the
/copied entity.

These basic built-in ASM rules, combined with graph patterns and graph transformation rules, form an
expressive, easy-to-use, yet mathematically precise language where the semantics of graph transformation
rules are also given as ASM programs (see [Var04] for more details).

Graph Transformation Rules

While graph patterns define logical conditions (formulas) on models, the manipulation of models is defined
by graph transformation rules [EEKR99], which heavily rely on graph patterns for defining the application
criteria of transformation steps. The application of a GT rule on a given model replaces an image of its
LHS pattern with an image of its RHS pattern. The language allows different notations for defining graph
transformation rules using the gtrule keyword.

The sample graph transformation rule in Fig. 2.9 defines a refactoring step of lifting an attribute from
child to parent classes. This means that if the child class has an attribute, it will be lifted to the parent.

The first syntax of a GT rule corresponds to the traditional notation (see Fig. 2.9(a)). It contains a
precondition pattern for the LHS, and a postcondition pattern that defines the RHS of the rule. In general,
elements that are present only in (the image of) the LHS are deleted, elements that are present only in
RHS are created, and other model elements remain unchanged. Moreover, further actions can be initiated
by calling any ASM rules within the action part of a GT rule, e.g. to report debug information or to
generate code. This action part is executed after the model manipulation part is carried out according to
the difference of the precondition and postcondition part.

import UML;
CHAPTER 2. MODEL-DRIVEN DEVELOPMENT

gtrule liftAttrsR(inout CP, inout CS, inout A) = {
  precondition pattern lhs(CP, CS, A, Par, Attr) = {
    Class(CP);
    Class.parent(Par, CS, CP);
    Class(CS);
    Class.attrs(Attr, CS, A);
    Attribute(A);
  }
  postcondition pattern rhs(CP, CS, A, Par, Attr, Attr2) = {
    Class(CP);
    Class.parent(Par, CS, CP);
    Class(CS);
    Class.attrs(Attr2, CP, A);
    Attribute(A);
  }
  action {
    print("Rule liftAttrR is applied on attribute " + name(A));
  }
}

Negative conditions are also commonly used in precondition patterns, especially, for model transformations between modeling languages in order to prevent the application of a GT rule twice on the same matching, and to enable incremental transformations. Examples for negative application conditions will be given in the case study of Sec. 5.5.

The second compact format of graph transformation rules directly corresponds to the FUJABA [NNZ00] notation as shown in Fig. 2.9(b).

import UML;
gtrule liftAttrsR(inout CP, inout CS, inout A) = {
  condition pattern cond(CP, CS, A) = {
    Class(CP);
    Class(CS);
    Class.parent(Par, CS, CP);
    Attribute(A);
    del Class.attrs(Attr, CS, A);
    new Class.attrs(Attr2, CP, A);
  }
  action {
    print("Rule liftAttrR is applied on attribute " + name(A));
  }
}

The rule contains a simple pattern (marked with keyword condition), that jointly defines the LHS of the graph transformation rule, and the actions to be carried out. Pattern elements marked with keyword new are created after a matching for the LHS is succeeded (and therefore do not participate in the pattern matching), and elements marked with keyword del are deleted after pattern matching.

Informal semantics of graph transformation rules

The main difference with the traditional GT notation is related to the use of parameter passing as an interface between preconditions and postconditions. More precisely, matchings of the precondition pattern are passed to the postcondition as parameters, thus a parameter of the precondition pattern can be (but not compulsory to be) used in the postcondition pattern.

- All the parameters of the postcondition which are (i) shared by the precondition or (ii) passed to the entire GT rule as an input parameter are treated as input parameters for the postcondition. These parameters are already bound before calculating the effects of the postcondition. In our example, input parameters of the postcondition are CP, CS, A, Par and Attr.
- Additional parameters of the postcondition are output parameters, which will be bound as the direct effect of the postcondition. The single output parameter of the postcondition of our example is Attr2.

On the one hand, this solution allows information hiding, i.e. not all variables inside a pattern need to appear in the parameter list of the pattern, which is very convenient for large patterns. The negative
side of this solution is that the execution mechanism of a GT rule becomes slightly more complex than in case of traditional GT rules. More specifically, the postcondition of a GT rule may prescribe three different operations on the model space.

- **Preservation.** If an input parameter of the postcondition appears in the pattern body, then the matching model element is preserved. The elements matched by variables $CP$, $CS$, and $A$ are thus preserved above.

- **Deletion.** If an input parameter of the postcondition does not appear in the body of the postcondition pattern then the corresponding model element is deleted. For instance, element matched by variable $Attr$ is deleted.

- **Creation.** If a variable which appears in the body of the postcondition pattern is not an input parameter of the postcondition, then a new model element is created, and the variable is bound to this new model element. In our example above, variable $AttrR$ is not an input parameter of the postcondition, thus it prescribes the creation of a new $attrs$ relation between class $CP$ and attribute $A$. This $AttrR$ is an output parameter of the postcondition but not passed back to the GT rule itself.

Based upon these principles, rule $liftAttrsR$ can be further compacted by simply omitting the parent relation from the postcondition and from the pattern parameter lists as well.

The formal semantics of GT rules is specified in Section 4.3.4 of [Var04].

**Generic and meta-transformations**

To provide algorithm-level reuse for common transformation algorithms independent of a certain metamodel, VIATRA2 supports generic and meta-transformations [VP04], which are built on the multilevel metamodeling support. For instance, we may generalize rule $liftAttrsR$ as lifting something (e.g. an Attribute) one level up along a certain relation (e.g. parent). The following example is the generic equivalent of the previous GT rule parameterized by types taken from arbitrary metamodels during execution time.

```plaintext
gtrule liftUp (inout CP, inout CS, inout A, 
inout ClsE, inout AttE, inout ParR, inout AttR) = {
  condition pattern transClose (CP, CS, A, ClsE, AttE, ParR, AttR) = {
    // Pattern on the meta-level
    entity (ClsE);
    entity (AttE);
    relation (ParR, ClsE, ClsE);
    relation (AttR, ClsE, AttE);
    // Pattern on the model-level
    entity (CP);
    // Dynamic type checking
    instanceOf (CP, ClsE);
    entity (CS);
    instanceOf (CS, ClsE);
    entity (A);
    instanceOf (A, AttE);
    relation (Par, CS, CP);
    instanceOf (Par, ParR);
    del relation (Attr, CS, A);
    del instanceOf (Attr, AttR);
    new relation (Attr2, CP, A);
    new instanceOf (Attr2, AttR);
  }
}
```

Compared to $liftAttrsR$, this generic rule has four additional input parameters: (i) $ClsE$ for the type of the nodes containing the thing to be lifted (Class previously), (ii) $AttE$ for the type of nodes to be lifted (Attribute previously), and (iii) $ParR$ (ex-parent) and (iv) $AttR$ (ex-attrs) for the corresponding for edge types.

When interpreting this generic pattern, the model transformation engine first binds the type variables ($ClsE$, $ParR$, etc.) to types in the metamodel of a modeling language and then queries the instances of these types. Internally, this is carried out by treating subtype-of and instance-of relationships as special edges in the model space, which enables the easy generalization of traditional graph pattern matching algorithms.
Invoking graph transformation rules

The basic invocation of a graph transformation rule is done using the apply keyword within a choose or a forall construct (further details on these constructs are given in Section 2.3.3). In each case, the actual parameter list of the transformation has to contain a valid value for all input parameters, and an unbound variable for all output parameters.

A rule can be executed for all possible matches (in a single, pseudo-parallel step) by quantifying some of the parameters using the forall construct. Finally, a GT rule can be applied as long as possible by combining the iterate and the choose constructs. The following example illustrates some possible invocations of our sample rule.

```plaintext
// execution of a GT rule for one attribute of a class
choose A apply liftAttrsR(Class1,Class2,A);

// calling the rule for all attributes of a class
forall A do apply liftAttrsR(Class1,Class2,A);

// calling the rule for all possible matches in parallel
forall C1, C2, A do apply liftAttrsR(C1,C2,A);

// iterate a GT rule as long as possible for the entire model space
iterate
  choose C1, C2, A apply liftAttrsR(C1,C2,A)
```

Note the semantic difference between the as long as possible and the forall execution modes: the former applies the rule once and only then does it select a next matching as long as matching exists, while the latter collects all matchings first, and then it applies the rule for each of them one by one in a single compound step. However, the current version of the Viatra2 engine does not provide built-in mechanisms for detecting the traditional semantic problems of graph transformation, namely, the possible non-termination in the as long as possible mode, and the possible conflicts in the forall mode due to performance considerations.

2.4 Summary

The fundamental model-driven development related techniques have been introduced in this chapter. We gave a brief overview of the main metamodeling and model transformation approaches.

VPM and Viatra2 have been introduced in more detail, as these methods will be used throughout the current thesis to give a formal foundation of metamodels and model transformations, and for the discussion of novel model transformation concepts.
Chapter 3

Model-driven tool infrastructure

3.1 Introduction

The previous Chapters introduced the most important methodologies and techniques for the implementation of model-driven development tool environments, including metamodeling, domain-specific languages, model transformations, and verification/validation techniques. We argued that model transformations play a fundamental role in model-driven design processes.

In the current chapter, we will introduce different solutions that enable the efficient development and execution of model transformation based tools. We defined and implemented several additions to the state of the art modeling and tool environments that increase the productivity at design time, and the usability and performance at runtime.

3.2 Advanced GT language constructs

In the current Section, we introduce rule-level composition techniques for graph patterns and GT rules. We already introduced and used the basic graph patterns as a stand-alone and reusable specification concept, and also GT rules that are constructed from LHS and RHS patterns by composing these predefined and also local patterns.

We will extend the concept of graph patterns by new constructs that will drastically reduce the complexity of graph transformation descriptions. Our proposal also supports the reusability of GT patterns and rules by allowing the creation of predefined pattern libraries for typical GT steps. We demonstrate our concepts using the transformation language of the Viatra2 framework.

3.2.1 Composition of simple patterns

We will use a simple UML transformation example to illustrate the graph pattern composition possibilities. The next listing shows a graph transformation rule called liftAttrsR that implements a simple refactoring step: it lifts an attribute from the child class to its parent.

```
precondition pattern lhs(CP, CS, A, Attr) = {
    UML.Class(CP);
    UML.Class(CS);
    UML.Class.parent(Par, CS, CP);
    UML.Attribute(A);
    UML.Class.attrs(Attr, CS, A);
}

postcondition pattern rhs(CP, CS, A, Attr) = {
    UML.Class(CP);
    UML.Class(CS);
    UML.Class.parent(Par, CS, CP);
    UML.Attribute(A);
    UML.Class.attrs(Attr2, CP, A);
}
```
We can recognize two patterns that can be used at various places during UML model refactorings: the first is the class-attribute pattern, and the second one is the parent class-child class pattern. If we refactor the transformation by extracting these patterns we get the following code:

```plaintext
pattern classAttribute(C,A) = {
    UML.Class(C);
    UML.Attribute(A);
    UML.Class.attrs(Attr,C,A);
}
pattern parentClass(Parent,Child) = {
    UML.Class(Parent);
    UML.Class(Child);
    UML.Class.parent(Parent,Child,Parent);
}
grule liftAttrsR(in CP , in CS , in A) = {
    precondition pattern lhs(CP ,CS ,A, Attr) = {
        find parentClass(CP ,CS );
        find classAttribute(CS ,A);
    }
    postcondition pattern rhs(CP ,CS ,A, Attr) = {
        find parentClass(CP ,CS );
        find classAttribute(CP ,A);
    }
}
```

The find construct is used for pattern composition, which means (in the simple case) the inclusion of other patterns in the current one by appropriate element copying and renaming. Using this functionality we can compose our graph transformation rule using a predefined pattern set.

Simple pattern composition in the LHS of a GT rule means that all elements in the component patterns (after an appropriate merging) have to be matched. Simple pattern composition in the RHS means that the merged pattern will define the overall RHS which defines the result of rule application.

Formally, the semantics of the find construct is the following:

**Definition 9 (Pattern composition)** If the pattern $P$ is composed of the $P_L$ local pattern and the $P_{f_i}$ called patterns ($i = 1 \ldots n$), and for each pattern $\text{Sign}(P) = V_{P_i}$ ($i = 1 \ldots N_{p}$) is the pattern signature (list of pattern parameter variables), and for each called pattern $\text{Act}(P) = A_{P_i}$ ($i = 1 \ldots N_{a_p}$) is the list of actual parameters, the composed pattern is the following:

$$P = P_L \land \bigwedge_{i=1}^{n} P_{f_i} \big| V_{f_i} = A_{f_i}$$

Since both internal patterns (which reside in the current ASM machine) and external patterns (which reside in different ASM machines) can be called transparently using the find construct, this way one can create stand-alone pattern libraries, which are reusable from any other transformations. The reusability of patterns also requires the presence of a developer documentation that supports the description of components (like in many modern programming languages). This can be done using Javadoc-like [Mic06] special comments, however, currently there is no tool support for it in Viatra2.

### 3.2.2 Composition of complex patterns

In case of the LHS of GT rules, the find construct also allows recursive pattern calls (the pattern calls itself), and OR-patterns which have multiple bodies (where a match of at least one body has to be found for a successful pattern matching).

The following rule illustrates both of these concept by defining the (transitive) ancestor of a class. The first part states that a parent is ancestor of a child if there is a direct parent relation between them, the second states that class Parent is ancestor of class Child, if there is a class C that is child of Parent and ancestor of Child.

```plaintext
pattern ancestorOf(Parent,Child) = {
    find parentClass(Parent,Child);
} or {
    find parentClass(Parent,C);
}
```
Definition 10 (OR-composition) If the $P(V_1\ldots V_n)$ pattern is composed of the patterns $P_1\ldots P_k$ using the OR operator, then:

$$P(V_1\ldots V_n) = \bigvee_{i=1}^{k} P_i(V_1\ldots V_n)$$

It should be noted that the recursive pattern call is the special case of the find construct described earlier.

When using such complex patterns in the RHS of GT rules, a new problem arises during execution. If the LHS contains recursive calls, the RHS should also be executed for all of the pattern matches. This requires the maintenance of the pattern call hierarchy (the execution stack of the LHS) and the execution of the RHS for all calls (by appropriate variable (re)naming). To illustrate the problem the following VTCL code introduces a pattern that can be used to lift the attributes of a class to all of its parents.

```vtcl
grule liftAttrs2R(in CP, in CS, in A) = {
  precondition pattern lhs(CP,CS,A,Attr) = {
    find ancestorOf(CP,CS);
    find classAttribute(CS,A);
  }
  postcondition pattern rhs(CP,CS,A,Attr) = {
    find ancestorOf(CP,CS);
    find classAttribute(CP,A);
  }
}
```

The execution environment needs to recognize that we have recursive patterns in both the LHS and RHS, and it has to “execute the RHS” for each level of the recursive call hierarchy of the LHS (and with the same pattern bodies in case of multi-body patterns). Such a recursive execution is sketched in Fig. 3.1.

![Figure 3.1: Execution stack for recursive patterns in GT rules](image)

### 3.2.3 Definition of design patterns

We already gave a brief introduction of design patterns that are *reusable solution descriptions* that can be used during the system design process.

In order to be able to organize design patterns and to automate pattern execution, a precise definition formalism should be developed. As outlined in the introduction, the definition or description of a pattern usually consists of several elements:

- **Pattern identifier** identifies the pattern uniquely.
- **Pattern description** is an (informal) description of the pattern for the user
- **Parameters** are the input parameters of the pattern in order to be able to specify the application context of the pattern.
CHAPTER 3. MODEL-DRIVEN TOOL INFRASTRUCTURE

- **Precondition** is an application condition for the pattern.
- **Action** is the description of the execution of the pattern.

Based on these elements, design patterns can be defined and catalogized. It is important that the pattern should be easy to understand, and - based on the precondition - the applicability in a context should be possible to unambiguously judged.

Traditionally, most of these elements are defined using informal notations like spoken languages, diagrams, and so on. For the automation, however, new description methods should be found.

There exist several languages for pattern definition like Pattern Language Markup Language (PLML) [Fin], its extended derivative (PLMLx) [Bie], and the Pattern and Components Markup Language (PCML) [Inc] but these focus only on the formalisation of the syntax of pattern descriptions (which elements to include) not the formalisation of the content of the descriptions themselves.

OMG also launched a specification, called Reusable Assets Specification (RAS) [Groj] but this has one one hand a different scope (targeting primarily reusable component descriptions), and on the other hand also lacks formal definition of the application of reusable elements for a concrete problem.

Some UML modeling tools (like IBM Rational Software Architect [IBM]) also offer the possibility to define and execute design patterns. However, these tools usually accept pattern implementations done using a programming language (for instance Java) that needs programming skills from the user, and also deep knowledge of the underlying modeling notation and tool environment.

In order to overcome the limitations of the existing solutions, a more precise, yet easy to use method needs to be developed. Our proposal that will be described in the followings, offers a design pattern definition framework that uses the graph transformation concepts introduced earlier.

**Definition 11 (Design pattern)** A Design pattern is a tuple: \( D\text{Pattern} = (\text{Id}, \text{Desc}, \text{ParamDesc}, \text{GTRule}) \)

- **Id** is a unique identifier for the pattern.
- **Desc** is an informal description of the design pattern using natural language.
- **ParamDesc** is the description of the role of design pattern parameters.
- **GTRule** is a graph transformation rule implementing the formal, executable specification of the design pattern.

This definition represents a formal and executable specification for design patterns.

**Traceability of pattern applications**

The establishment of design decision and action traceability is a key issue during the development of any systems, and it is mandatory for safety-relevant systems. In order to be able to trace design pattern applications, a generic model extension facility should be added to the model management framework.

**Definition 12 (Pattern application trace)** A pattern application is stored as a set of VPM model elements:

```markdown
entity(PatternApplication);
relation(patternIdentifier, PatternApplication, String);
relation(variables, PatternApplication, PatternVariableSlot);
entity(PatternVariableSlot);
relation(variableIdentifier, PatternVariableSlot, String);
relation(variableValue, PatternVariableSlot, ValueSpecification);
entity(ValueSpecification);
```

Pattern application element refers to the identifier of the pattern that has been applied, and to the actual value of all pattern parameters during the execution. It should be noted, that (as pattern variables can contain values of arbitrary data type) the generic ValueSpecification type is used for the value storage.

The previous trace structure can be used for any model to store pattern applications in the model space, together with the models themselves. This way, no additional storage means are needed for the traceability information.
3.3. CASCADING GRAPH TRANSFORMATIONS

3.2.4 Conclusions

In this Section we discussed composition mechanisms for graph transformation rules by merging predefined graph patterns both in the LHS (precondition) and RHS (postcondition) of the rules.

Our techniques support the reusability on graph pattern and transformation rule levels. The design pattern specification that has been introduced here builds on the GT patterns and rules and establishes a formal and executable specification for design patterns.

These mechanisms are already integrated in the VTCL model transformation language of the VIATRA2 framework. The transformation engine of VIATRA2 already has a stable implementation for the composition of both simple and complex patterns in the LHS, while the implementation for pattern composition in the RHS is currently in an experimental phase.

Our practical transformation experiences in various industrial and academic research projects show that the specification of complex model transformation problems can be drastically reduced (by at least an order of magnitude in lines of transformation code), and by avoiding the duplication of code, improves testability and code quality.

3.3 Cascading Graph Transformations

As model transformation development is becoming an engineering discipline, conceptual and tool support is necessitated for the entire life-cycle, i.e. the specification, design, execution, validation and maintenance of transformations. However, different phases of transformation design frequently set up conflicting requirements, and it is difficult to find the best compromise. For instance, the main driver in the execution phase is performance, therefore, a compiled MT approach (where a transformation is compiled directly into native source code) is advantageous. On the other hand, interpreted MT approaches (where transformations are available as models) have a clear advantage during the validation (e.g. by interactive simulation) or the maintenance phase due to their flexibility.

In this Section, we introduce solutions for the establishment of design and execution environments supporting both interpreted and compiled approaches to separate the design (validation, maintenance) of a transformation from its execution. Interpreter-based Platform-independent Transformers (PITs) [BFJ+03, VP04] ease the testing, debugging and validation of model transformations within a single transformation framework without relying on a highly optimized target transformation technology. Platform-specific Transformers (PSTs) are compiled (in a complex model transformation and/or code generation step) from an already validated transformation specification into various underlying tools or platform-dependent transformation technologies (e.g. Java, XSLT, etc.) and integrated into off-the-shelf CASE tools as stand-alone plugins.

We will introduce the basic approach of generating platform-specific transformer plugins to be executed on top of a runtime modeling platform. PIT model transformations are captured by a high-level, rule and pattern based language (VTCL). Special emphasis will be put on the reusability of model transformations, and other model-driven tool components during the interpreted and compiled model transformation execution.

3.3.1 Overview of the Approach

An overview of generating platform-specific transformer standalone plugins from PITs is provided in Figure 3.2 The general flow of developing and executing PITs is the following.

1. Metamodel design. Metamodels of the source and target modeling languages (or domains) are designed and stored according to the metamodeling approach of the design environment.

2. Develop importers. Model importers are needed to load models from external resources into the model space.

3. Model import. These importers build up an internal representation of the source model which corresponds to its metamodel.

4. Transformation design. Model transformations between source and target metamodels are captured by the model transformation language of the environment.

5. Transformation execution. These transformations are executed on the source model by a transformation engine to derive the target model.
CHAPTER 3. MODEL-DRIVEN TOOL INFRASTRUCTURE

Figure 3.2: Overview of the PST approach

6. Model export. Finally, the target model can be postprocessed by special model transformations generating code and code formatters to be printed in the required output format.

The main goal of this section is to show how metamodels, importers, and model migration transformations can be re-used in both the design and execution environments. We propose a solution for the following tasks.

- From PIM to PSM models. Platform specific representation will be generated from the source and target metamodels including the reference objects representing the mapping between them.
- Reuse of model importers and exporters. Our experience shows that the development of a model importer also requires significant work. As a consequence, the reuse of VIATRA2 importers (exporters) implemented for building up (extracting) an internal (VPM) model representation from (to) a textual source (target) file in the platform-specific transformation plugin is also an important goal. An other option is to reuse the existing platform-specific importers and exporters during the development as VPM importers and exporters. Both options will be investigated in the followings.

While the tasks outlined here are fundamental for the successful implementation of PSTs, the key element of the chain is the compilation of transformation programs to the target platform. This issue is out of scope of our work, but is widely covered by the related literature [Hor07, HVV06, HB06, Hor06a, Hor06b].

3.3.2 Metamodel conversion

Metamodel conversion is a key step in the process of migrating transformations from the development environment to the runtime platform. Supposed that there is a $M_{design}$ that is present in the development environment, a new $M_{exec}$ should be created in the target environment, or vica versa, a legacy metamodel should be mapped to the design environment to support the definition of transformations. In both cases, the two metamodels describe the same language, but use a different metamodeling environment, meaning that the metamodel conversion is a special (meta-)transformation that only modifies the representation and not the semantics of the metamodels.

The metamodel conversion transformations can be described independently of the modeling languages, between any pairs of metamodeling environments.

3.3.3 Model migration

The migration of metamodels enables the definition of transformations in the design environment, but it is also necessary to create instance models in order to be able to test the transformation using the more sophisticated support of the design environment. This requirement necessitates the definition of model-level transformations that migrate instance models between the design-time and runtime environments. The resulting transformation chain is illustrated by Figure 3.3.
3.3. CASCADING GRAPH TRANSFORMATIONS

The model migration includes two sequential transformations both on source and target model sides of the transformation under design having several drawbacks:

- Each model is present in the memory in two copies (one in the runtime and one in the design environments) leading to problems if the test cases include large models. (Our experience shows that typical industrial models consume several GBs of memory).

- The transformation execution is sequential that results in longer execution time for the model import and export processes. This can be significant during the execution of high number of test cases.

These problems could be solved by a model importer that reads the serialized model and directly builds an instance model in the design environment (and by a similar model exporter). Model importer development, however, requires significant amount of resources, as the model storage formats are highly intuitive in case of most languages (even if there are well established standards, like XML Metadata Interchange (XMI) [Groo]), and importer testing is also a complex task. Instead of re-implementing the importer transformation, it should be integrated with the model migration into an atomic step.

3.3.4 Transformation cascading

Model transformation cascading aims at the combination of sequential transformations into a single, atomic transformation. Informally, this means that the intermediate model (result of the first transformation) will not be generated, only the final output model will be present at the end of execution. More formally:

**Definition 13 (Cascaded transformation)** Let $\mathcal{M}_{src} \xrightarrow{x_1} \mathcal{M}_{int}$ and $\mathcal{M}_{int} \xrightarrow{x_2} \mathcal{M}_{trg}$ be two consecutive transformations. The transformation $\mathcal{M}_{src} \xrightarrow{\text{casc}} \mathcal{M}_{trg}$ is a cascaded transformation based on $x_1$ and $x_2$ if for all $\mathcal{M}_{src}$:

\[
\mathcal{M}_{src} \xrightarrow{\text{casc}} \mathcal{M}_{trg},
\]

and

\[
\mathcal{M}_{trg_1} = \mathcal{M}_{trg_2}.
\]

We denote cascading as: $\mathcal{M}_{src} \xrightarrow{\text{casc}} \mathcal{M}_{trg}$

**Automatic synthesis of cascaded transformations**

The automatic synthesis of cascaded model transformations is required in order to exploit the advantages of the single step transformation without the need to manually implement it. We will introduce a method for this purpose in the current section, that is able to integrate a subset of transformations that is relevant from the point of view of model migration. Different restrictions should be fulfilled by the first and second elements of the cascaded transformation.
Restrictions on the first transformation

- **No metamodel manipulation.** The metamodel should be read-only from the point of view of the transformation.
- **No delete operation.** The transformation cannot delete elements from the model.
- **No relation re-targeting.** Although VPM enables the setting of relation source and target after the relation has been created, the transformation must not use this feature.
- **The transformation should be deterministic** meaning that two executions of it on the same input model should result in identical target models.

While these items are quite restrictive, our experience from several industrial and academic research projects show that model importer transformations fit into the subset defined by these rules.

Restrictions on the second transformation

- **No metamodel manipulation.** The metamodel should be read only from the point of view of the transformation.
- **No delete operation.** The transformation cannot delete elements from the model.
- **No imperative control structures.** The transformation should be fully declarative (pure GT).
- **Negative application condition cannot be used in GT rules.**
- **The transformation should be deterministic** meaning that two executions of it on the same input model should result in identical target models.

While imperative control structures can help to make model transformations more compact, there are several subclasses of transformations where they are not needed. The above rules define a subset of transformation similar to the QVT relations language [Grod] that is intended to be used for horizontal transformations. In our problem domain, the second transformation is a horizontal one, as it transforms between metamodeling environments of the same abstraction level.

Implementation of the cascade transformation The cascade transformation is implemented as a composite structure (see Figure 3.4). The first transformation is added without changes. This transformation is hosted by a *modelspace proxy* element that emulates the intermediate model storage. The proxy is able to intercept all model manipulation primitives that are issued during the execution of the transformation and feeds the created elements into a RETE-network [For82] created based on the precondition of rules of the second transformation. The network is built using the technique described in [BOR+08]. If a new matching for a LHS pattern is found by the RETE network, it immediately triggers the execution of the corresponding graph transformation rule of the second transformation. This way, the second transformation is executed in an *event-triggered* or *live* mode [RBÖV08b].

The benefit of this solution is that the RETE network is fed by *shallow model elements* that can be treated as *tokens* representing real model elements with a reduced memory footprint. This solution optimizes the memory consumption of the intermediate model and the pattern matching structures (the RETE network). On the other hand, executing the second transformation in an event-triggered mode reduces the overall execution time of the transformation chain as compared to the clear sequential execution. We have to show, however, that the proposed structure is a proper cascading, e.g. it fulfills Definition 13. In order to be able to prove that the cascading is proper, we need to define a precedence relation between rule applications.
Definition 14 (Rule application precedence) A rule application \( r_1 \) precedes an other \( r_1 <_p r_2 \), if the execution of \( r_1 \) must precede the execution of \( r_2 \). That means that \( r_1 \) either directly or indirectly influences the matching of \( r_2 \).

Proposition 1 (Precedence relation) Precedence is irreflexive, asymmetric, and transitive.

Proof Precedence is irreflexive because a rule application cannot influence its own precondition.

Precedence is asymmetric as if \( r_1 <_p r_2 \) then \( r_1 \) should precede \( r_2 \), therefore \( r_2 \) cannot preceede \( r_1 \):

\[
r_1 <_p r_2 \Rightarrow f_2 <_p r_1
\]

Precedence is reflexive as if \( r_1 <_p r_2 \) and \( r_2 <_p r_3 \), then \( r_1 \) should also precede the application of \( r_3 \), as it is a precondition of the precondition of \( r_3 \): \( r_1 <_p r_3 \).

If two pattern applications are not in precedence relation with each other, their sequence in the execution can be arbitrary.

Proposition 2 (Transformation cascading) The method described in this section is a proper cascading, ie. for all \( M_{src} \):

\[
M_{src} \xrightarrow{x_1} M_{int} \xrightarrow{x_2} M_{trg_1},
\]

and

\[
M_{src} \xrightarrow{casc} M_{trg_2}
\]

\[
M_{trg_1} = M_{trg_2}
\]

Proof In the composite transformation structure, \( x_2 \) is executed in parallel to \( x_1 \). We have to prove that this execution is equivalent to the sequential one.

The incremental and batch pattern matching strategies are identical in the sense that they find the same matchings if the graph patterns and the models are identical.

Based on the restrictions on \( x_1 \) and \( x_2 \), we can state that if there is a matching for one of the GT rules (\( R_1 \)) of \( x_2 \), it will not be removed. This is the consequence of that a) \( R_1 \) cannot contain negative application conditions, b) the sub-graph representing the matching of \( r_1 \) cannot be deleted, as neither \( x_1 \) nor \( x_2 \) can delete elements or retarget relations. This implies that a rule can be executed at any time on a matching, as once a matching is established, it will be valid until the end of transformation execution.

An execution of \( x_2 \) is a sequence of graph transformation rule executions: \( Exec = r_1 \ldots r_n \). There are two executions, \( Exec_s \) represents the sequential, and \( Exec_p \) represents the parallel (or simultaneous) execution of \( x_2 \).

Based on the precedence relation \( <_p \), we can rearrange the execution steps. It has been shown that if a matching of \( x_2 \) is established, it will not removed by later transformation steps, so the corresponding rule execution can be delayed. Using this property, let us partition the steps of both \( Exec_p \) and \( Exec_s \) as follows:

Let \( S_1 = \{ r_i \in Exec_s | \exists r_j \in Exec_s : r_j <_p r_i \} \).

Similarly:Let \( P_1 = \{ r_i \in Exec_p | \exists r_j \in Exec_p : r_j <_p r_i \} \)

These sets contain the rule applications that do not depend on other rule applications of \( x_2 \).

The set building can be continued:

\[
S_{i+1} = \{ r_i \in (Exec_s \setminus \bigcup_{j=1..i} S_j) | \exists r_k \in (Exec_s \setminus \bigcup_{j=1..i} S_j) : r_k <_p r_i \}
\]

\[
P_{i+1} = \{ r_i \in (Exec_p \setminus \bigcup_{j=1..i} P_j) | \exists r_k \in (Exec_p \setminus \bigcup_{j=1..i} P_j) : r_k <_p r_i \}
\]

The next set will contain the rule applications that have preceeders only in the already processed sets. The result is a sequence of set steps for each execution that partition the execution sequences. The step ordering is a sound ordering, as it fulfills the precedence relation between steps and (as has been already shown) the delaying of a step does not affect the result of the transformation.

As \( x_1 \) is deterministic, it builds identical models if the input model is the same. That means (because identical input models imply identical matching sets), that all elements in \( S_1 \) have a corresponding element in \( P_1 \) so that their matchings are equivalent. This fact, and the determinism of \( x_2 \) implies that the execution of all elements of \( S_1 \) and \( P_1 \) results in an identical (intermediate) model state. The same is true for all sets \( S_2 \ldots S_n \) and \( P_2 \ldots P_n \) as all will have the same input model (consisting of the output of \( x_1 \) and the steps in the preceeding sets) and thus they contain the same steps and again produce identical output models.
CHAPTER 3. MODEL-DRIVEN TOOL INFRASTRUCTURE

The result of this induction is that the target models built by the sequential and parallel executions of $x_2$ are identical. □

It should be noted that the idea of rearrangement of rule application sequences is similar to the more generic concept of shift equivalence that has been formulated in [Roz97].

The main usage scenario for this type of cascaded transformations is the reuse of importer and exporter modules. These modules are in most cases implemented manually, and thus require significant development efforts. The reuse of them between design time and runtime is essential. We will introduce an example described in [9] that illustrates the concept introduced in this section.

3.3.5 Performance evaluation of cascaded transformations

In order to evaluate the performance of cascaded transformations, a complex, real-life case study has been evaluated. We have chosen the problem of importing AutoSAR models to VPM, as the AutoSAR-EMF and EMF-VPM transformations were already available. The cascaded transformation is generated from these implementation using the technique introduced in the previous Sections.

The performance evaluation has been carried out on three AutoSAR models. The first, relatively small model was a software model from the DECOS project, the second model was the standard ECU parameter definition model included in the AutoSAR standard containing around 10000 model elements. Finally, a huge model from an industrial project has been used that contains over two million model elements. The results of the test runs can be seen in Table 3.1.

Table 3.1: Cascade transformation performance results

<table>
<thead>
<tr>
<th>Model</th>
<th>Method</th>
<th>Execution time (s)</th>
<th>Heap usage (MB)</th>
<th>Non-heap memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small DECOS</td>
<td>Sequential</td>
<td>5.2</td>
<td>57</td>
<td>58</td>
</tr>
<tr>
<td>Small DECOS</td>
<td>Cascade</td>
<td>1.1</td>
<td>40</td>
<td>49</td>
</tr>
<tr>
<td>ECU Paramdef</td>
<td>Sequential</td>
<td>12.6</td>
<td>124</td>
<td>59</td>
</tr>
<tr>
<td>ECU Paramdef</td>
<td>Cascade</td>
<td>6.5</td>
<td>107</td>
<td>48</td>
</tr>
<tr>
<td>Large</td>
<td>Sequential</td>
<td>236</td>
<td>1100</td>
<td>59</td>
</tr>
<tr>
<td>Large</td>
<td>Cascade</td>
<td>95</td>
<td>1000</td>
<td>59</td>
</tr>
</tbody>
</table>

Execution time and memory consumption has been measured during the test runs. Heap memory in Java is used for the storage of object data, while non-heap memory is used for class and metadata storage. The difference in non-heap memory consumption is caused by the fact that the implementation of the two separate transformations requires the loading of more classes than that of the cascade version. Heap memory usage shows that the cascade implementation reduces the memory consumption by 10 to 30 percent depending on model size. It should also be noted that the third, large model (having more than 1GB memory footprint) is near to the maximum that can be handled by in-memory repositories.

Comparing the execution time of the implementation we can state that the cascading transformation reduces the execution time by 50-60 percent in average. In case of the small model, the relatively higher time benefit is the result of the fact, that in case of a sequential transformation, the initialization of both transformations takes place, while the cascade implementation needs to be initialized only once. In case of larger models the initialization is not significant in the overall execution time. To conclude the evaluation, it can be stated that the cascade implementation has significant benefits both from the execution time and memory consumption points of view.

3.3.6 Reuse of VPM importers on the EJB platform

The EJB [Pro] technology has been selected as runtime platform in [9] as it supports the relational database based storage of information and thus it can store large models exceeding the capacity of in-memory storage mechanisms. The transformations of the case study required the usage of UML as modeling language, and the import of UML models from standard XMI sources. The EJB platform, however, did not have such import facilities. Using the trasformation cascading approach, the VPM-based model importer has been reused, by cascading it to a VPM-to-EJB model transformation. This experiment had shown that the reuse of importers is possible and results in a better productivity during transformation development by avoiding the costly re-implementation of the same software component on different technology platforms.
3.3.7 Related work

Graph transformation cascading introduced here relates to some extent to graph transformation amalgamation that has been discussed in [TB94]. However, there is a difference between the two approaches. Graph transformation amalgamation aims at the parallel application of amalgamated graph transformation rules in order to execute several rules in a single graph modification step. Our approach targets the simultaneous execution of two graph transformations, without modifying the atomic rule application steps. It can be seen that the basic intent of parallelization is the similar, but it is done on different abstraction levels.

3.3.8 Conclusions

Several techniques have been introduced in this Section that all serve the common goal of support of platform-specific model transformation development and execution by a) migrating metamodels and models between the runtime and design time modeling environments, b) supporting the cross-environment reuse of important tool components such as editors and model importers. We illustrated the utilization of the results using a case study that featured VPM as design-time environment, and ECore/EMF as runtime platform.

3.4 A formal semantics for ECore

ECore is the de facto standard metamodeling environment used by the industry and the academia as the basis for domain-specific language and tool development. Although it is widely used, it lacks formal semantics. We introduce a formal, VPM and GTASM based operational semantics for ECore in the current section.

3.4.1 Related work

Most of the current modeling languages lack of standard, uniform formal semantics. UML, and its domain specific profiles, like SysML, and MARTE are good examples for that, and most domain-specific languages (like AutoSAR) and even behavioral specification languages (Matlab Simulink) are also in this category.

There are, however, several attempts – mainly in the academia – to establish a formal semantics for different modeling languages. Several semantics definitions have been proposed for UML (for instance [BCD+07]) or parts of UML (e.g. for activities [SH05], sequence diagrams [LLJ04], and statecharts [PM06])

Metamodeling languages are also lacking of formal semantics. MOF, the standard metamodeling environment of OMG does not have semantics, but there is an open request for proposals for SMOF, the semantic enrichment of MOF. There are several semantics proposals for MOF, like [BM08] and the framework developed in the MOMENT project [Tea09]. A declarative, graph transformation based semantics for ECore model transformations is proposed in [BET08]. The authors concentrate on the formalization of model transformations and not on the formalization of the metamodeling environment itself. Although ECore is a derivative of the standard EMOF and is widely used, up to now no widely accepted semantics definitions exist for it.

3.4.2 Mapping of ECore metamodels and models to VPM

The basis for the operational semantics is the mapping of ECore metamodeling constructs to VPM in order to establish a relation between the meta level state of the two environments.

ECore metamodels of the languages involved in the transformation design are usually already implemented. It should be noted that this is a typical situation, as most of the generic and domain-specific languages have extensive support on top of ECore/EMF (for instance, UML, SysML, AutoSAR, AADL, and so on).

The mapping relies on the mapping of ECore on top of VPM as described in Section 2.2.7. ECore is completely mapped to VPM, and this mapping directly defines the transformation

\[ \mathcal{M}_{ECore} \xrightarrow{ecore2vpm} \mathcal{M}_{VPM} \]

The model level mapping is outlined by the following algorithm.

Algorithm 1 (ECore to VPM model mapping) Let \( M_e \) be the root element of the ECore model, and \( M_v \) the container for the converted VPM model. Let ASM function \( f_{emap} : \text{EObject} \rightarrow \text{Entity} \) be the mapping function between ECore and VPM entities, and set \( \text{Ref} \) be a set of tuples \( R = (\text{Source}, \text{Type}, \text{Target}) \),
where Source is a source entity, Type is a relation, and Target is a target EObject representing a relation leading to the target from source. We also assume that there is an ASM function metaref : ECoreElement → ModelElement that maps the meta level elements.

The main algorithm:

1. call mapEObject($M_e, M_v$)
2. for each $(S, R, T)$ in Ref:
   (a) let $\text{Trg} = f_{\text{emap}}(T)$
   (b) new(relation($\text{Rel}, S, \text{Trg}$))
   (c) new(instanceOf((R, $\text{Rel}$)) create and set type of the relation.

The first step calls a utility method to convert the objects and attributes of the source model, and the second step creates the references in the target model.

The above algorithm calls the following method:

$\text{mapEObject}(\text{Src, Container})$

1. let $\text{Meta} = \text{metaref(type(Src))}$ the type of the actual element in VPM.
2. new(entity($\text{Trg}$)) in Container, new(instanceOf($\text{Trg, Meta}$)) (create the corresponding element in VPM).
3. Update the reference function: update $f_{\text{emap}}(\text{Src}) = \text{Trg}$.
4. for each $A$ in attributes($\text{Src}$) do:
   (a) for $i := 0 \ldots \text{size}($Src, $A$) do:
      i. $V := \text{get}($Src, $A$, $i$)
      ii. let $\text{AT} = \text{metaref(type(V))}$ the type of the attribute
      iii. let $\text{RT} = \text{metaref(type(A))}$ the type of the attribute reference
      iv. new(entity($\text{Val}$)), new(instanceOf($\text{Val, AT}$)) create the attribute entity.
      v. new(relation($\text{R, Trg, Val}$), new(instanceOf($\text{R, RT}$)) create the attribute relation.
      vi. setValue($\text{Val, getValue(Val)}$) set the attribute value.
6. for each $E$ in contents($\text{Src}$) do:
   (a) call mapEObject($E, \text{Trg}$)

Steps 2-4 create the entity that corresponds to the ECore object, step 5 creates the attributes, and step 6 puts the reference information to the temporary Ref set. Step 7 makes a recursive call for all the contained elements of the current EObject.

Proposition 3 The above algorithm always terminates.

Proof The main algorithm terminates in finite steps, as step 1 executes once, and step 2 executes once for all references in the source model.

The mapEObject method also terminates as steps 1-3 are executed once, step 4 is executed once for all attributes of the element, step 5 once for all references starting from the actual element, and step 6 once for all contained element of the current EObject. Step 6 cannot cause infinite loop by recursion, as ECore does not allow the cycles in the containment hierarchy. That means, that if an EObject is processed by mapEObject, it will not be processed again.

Proposition 4 (Type conformance) If $\mathcal{M}_{VPM}$ is derived from $\mathcal{M}_{ECore}$, and the source model $\mathcal{M}_{ECore}$ is type conforming to $\mathcal{M}_{ECore}$ then the derived VPM model $\mathcal{M}_{VPM}$ will also be type conforming to $\mathcal{M}_{VPM}$. 

3.4. A FORMAL SEMANTICS FOR ECORE

Proposition 5 (Object correspondence)

∀o : EObject, oECore ∈ EMCore : ∃e : Entity, eVPM ∈ MVPM, where typeof(o) corresponds to typeof(e)

Proof If the mapEObject method is executed for an EObject, steps 1 and 2 assure the creation of an entity with corresponding type, and the function femap will establish a link between these objects. Starting the algorithm with a model root as input parameter assures that all objects in the model will be visited by mapEObject as it follows the containment hierarchy, and by definition all EObject of an ECore model are (either directly or indirectly) contained by the model root.

Proposition 6 (Attribute correspondence) Let o be an EObject, and d be an EDataType in EMCore, with o.attr containing d. Moreover let e, val be the VPM equivalents (in MVPM) of o, and d, respectively. Then there exists a VPM relation rel in MVPM with typeof(attr) conforming to typeof(rel), where rel.src = o and rel.trg = val.

Proof We have already shown that all EObjects have corresponding entities in the target model. If an EObject is visited, all of their attributes are processed (step 4 of mapEObject). Step 4.a iterates through all values contained by the attribute, and steps 4.a.i - 4.a.v create the entity corresponding the data type instance (also setting its type), and the relation corresponding the attribute (also setting its type).

This way a) for each attribute value, an entity with a type corresponding to the data type is created, b) the data element entity is connected to the entity corresponding the attribute owner EObject using a relation that has a type corresponding the attribute.

Proposition 7 (Reference correspondence) Let o1, o2 be two EObjects in EMCore, with o1.ref containing o2. Moreover let e1, e2 be the VPM equivalents (in MVPM) of o1, and o2, respectively. Then there exists a VPM relation rel in MVPM with typeof(ref) conforming to typeof(rel), where rel.src = o1 and rel.trg = o2.

Proof We have already shown that all EObjects have corresponding entities in the target model. If an EObject is visited, all of their references are processed (step 5 of mapEObject) and the information about the source, target, and type of the reference is stored in the metaref set.

In the second phase, all elements of metaref are processed by the main algorithm (step 2), and for each of them a relation between the entities corresponding to the source and target of the original reference is created. Furthermore, an instanceOf relationship between the relation and the type corresponding the original reference is also created.

We have shown that a) all references are processed by the algorithm, b) the type of the relation created corresponds to the type of the ECore reference.

3.4.3 Operations on ECore models

The EMF/ECore framework offers the possibility of introducing custom model representation techniques by implementing a pre-defined interface of the framework (called EStore). All the model query and manipulation operations use this interface to access the model.

The list of operations is the following:

```java
void add(InternalEObject object, EStructuralFeature feature, int index,
         java.lang.Object value)
//Adds the value at the index in the content of the object's feature.

void clear(InternalEObject object, EStructuralFeature feature)
```
CHAPTER 3. MODEL-DRIVEN TOOL INFRASTRUCTURE

// Removes all values form the content of the object's feature
boolean contains(InternalEObject object, EStructuralFeature feature, java.lang.Object value)
    // Returns whether the content of the object's feature
    // contains the given value.

EObject create(EClass eClass)
    // Creates a new instance of the class.

java.lang.Object get(InternalEObject object, EStructuralFeature feature, int index)
    // Returns the value at the index in the content of the object's feature.

InternalEObject getContainer(InternalEObject object)
    // Returns the object's container.

EStructuralFeature getContainingFeature(InternalEObject object)
    // Returns the object's containing feature.

int hashCode(InternalEObject object, EStructuralFeature feature)
    // Returns the hash code of the content of the object's feature.

int indexOf(InternalEObject object, EStructuralFeature feature, java.lang.Object value)
    // Returns the first index of the given value in the
    // content of the object's feature.

boolean isEmpty(InternalEObject object, EStructuralFeature feature)
    // Returns whether the content of the object's feature is empty.

boolean isSet(InternalEObject object, EStructuralFeature feature)
    // Returns whether the object's feature is considered set.

int lastIndexOf(InternalEObject object, EStructuralFeature feature, java.lang.Object value)
    // Returns the last index of the given value in the
    // content of the object's feature.

java.lang.Object move(InternalEObject object, EStructuralFeature feature, int targetIndex, int sourceIndex)
    // Moves the value at the source index in the content of
    // the object's feature to the target index.

java.lang.Object remove(InternalEObject object, EStructuralFeature feature, int index)
    // Removes the value at the index in the content of
    // the object's feature.

java.lang.Object set(InternalEObject object, EStructuralFeature feature, int index, java.lang.Object value)
    // Sets the value at the index in the content of the object's feature.

int size(InternalEObject object, EStructuralFeature feature)
    // Returns the number of values in the content of the object's feature.

java.lang.Object[] toArray(InternalEObject object, EStructuralFeature feature)
    // Returns a new array of the values in
    // the content of the object's feature.

java.lang.Object[] toArray(InternalEObject object, EStructuralFeature feature, java.lang.Object[] array)
    // Returns an array of the values in the content of the object's feature.
3.4. A FORMAL SEMANTICS FOR ECORE

In order to provide an operational semantics for ECore, the operations defined by the EStore interface should be defined in a formal way. As the ECore model manipulation operators use a higher level of abstraction, each method is implemented as an ASM rule using the VTCL language.

3.4.4 GTASM definition of ECore operations

We introduce a formal, operational semantics for the ECore environment, by defining the EStore operations using the GTASM language.

Definition 15 (GTASM based semantics for ECore) The semantics of the EStore operations is defined by the rules of a GTASM machine. The VTCL implementation of the machine can be found in Appendix C.4.

We utilize standard Object-oriented testing techniques [Bin99] to validate the semantics definition. The particular testing method is called Alpha-Omega cycle where the object under test is taken from the initial (alpha) state to a final (omega) state by invoking its method in a predefined sequence.

We adapt the alpha-omega approach to GTASM and specify method call sequences in order to validate the functionality of the semantics. It should be noted that neither the original alpha-omega testing, nor we are aiming at a full testing, or to reach some test coverage criteria. This tests only aim at the validation of basic functionality of the semantics definition.

Proposition 8 Let Cls be an EClass, and Attr an attribute, starting from Cls. Moreover, let DT the type of Attr.

Let us suppose that the following sequence is executed on an empty instance model:

```
seq {
    call create(Cls, Obj);
    call create(DT, AtVal);
    call add(Obj, Attr, 0, AtVal);
}
```

The result of the execution is an EObject that is instance of Cls, and has a single attribute.

Proof Let us inspect the atomic model manipulation operations executed:

```
//create(Cls, Obj)
new(entity(Obj));
new(instanceOf(Obj, Cls))
//create(DT, AtVal)
new(entity(AtVal));
new(instanceOf(AtVal, DT))
//add(Obj, Attr, 0, AtVal)
//rule isSet is called: it return false (no relations from Obj)
//the else block is executed in add
new(relation(Rel, Obj, AtVal));
new(instanceOf(Rel, Attr));
```

If we look at the atomic calls, we can notice that the first two calls create the EObject instance and set its type, the next two calls create the attribute value entity and set its type, while the last two calls create the attribute relation and set its type. This means that the resulting model will have a single object with a single attribute and all elements have the intended types.

Proposition 9 Let Cls be an EClass, and Ref a reference, starting from Cls. Moreover, let Cls2 be the type of Ref.

Let us suppose that the following sequence is executed on an empty instance model:

```
seq {
    call create(Cls, Obj);
    call create(Cls2, Obj2);
    call create(Cls2, Obj3);
    call add(Obj, Ref, 0, Obj2);
}```
CHAPTER 3. MODEL-DRIVEN TOOL INFRASTRUCTURE

call get(Obj,Ref,0,Val);
//assert : Val == Obj2
call add(Obj,Ref,0,Obj3);
call get(Obj,Ref,0,Val);
//assert : Val == Obj3
}

The result of the execution is a model consisting of three objects, where Obj has references to both of the other elements, and the assertions (defined in the source code as remarks) are fulfilled during execution.

Proof Let us inspect the atomic model manipulation operations executed:

//create(Cls,Obj)
new(entity(Obj));
new(instanceOf(Obj,Cls))
//create(Cls2,Obj2)
new(entity(Obj2));
new(instanceOf(Obj2,Cls2))
//create(Cls2,Obj3)
new(entity(Obj3));
new(instanceOf(Obj3,Cls2))
//add(Obj,Ref,0,Obj2);
//the else block is executed in add
new(relation(Obj,Obj2));
new(instanceOf(Rel,Ref));
//get(Obj,Ref,0,Val);
//get uses the pattern findFirstFeature that uses
//findFirstFeatureInt. In this case the first body of findFirstFeatureInt
//is fulfilled, with the following variable substitutions:
//Obj=Obj, Feat=Ref, A=Cls, B=Cls2, Value=Obj2, X=Rel created in the last step.
//the negative patterns in findFirstFeature are not fulfilled (no next relation)
//internalGet returns: FeInst=Rel
//get returns: Val=target(FeInst)=target(Rel)=Obj2

//add(Obj,Ref,0,Obj3);
//the if(Idx==0) block is executed
//internalGet returns the Rel, the actual 0th relation of reference Ref
new(relation(Obj,Obj3));
new(instanceOf(Obj3,Ref));
new(EOBject.orderedRelation.next(N,Rel,Rel2));

//get(Obj,Ref,0,Val);
//get uses the pattern findFirstFeature that uses
//findFirstFeatureInt. In this case the first body of findFirstFeatureInt
//is fulfilled, with the following variable substitutions:
//Obj=Obj, Feat=Ref, A=Cls, B=Cls2, Value=Obj2, X=Rel created in the last step.
//the negative patterns in findFirstFeature are not fulfilled (no next relation)
//internalGet returns: FeInst=Rel2
//get returns: Val=target(FeInst)=target(Rel2)=Obj3

The result of the execution is the creation of three entities (Obj,Obj2,Obj3), and two relations (Rel,Rel2) all having the proper typing. Both of the assertions are also fulfilled.

□

Proposition 10 Let Cls be an EClass, and Ref a reference, starting from Cls. Moreover, let Cls2 be the type of Ref.

Let us suppose that the following sequence is executed on an empty instance model:

seg {
call create(Cls,Obj);
call create(Cls2,Obj2);
call create(Cls2,Obj3);
call add(Obj,Ref,0,Obj2);
call add(Obj,Ref,0,Obj3);
call clear(Obj,Ref);
}
3.4. A FORMAL SEMANTICS FOR ECORE

```java
null

```call isSet(Obj,Ref,IsSet)

//assert: IsSet==false

```}

The result is the execution that variable IsSet is false, that means that the clear rule deletes all values from the reference Ref of object Cls.

**Proof** We have already shown that the result of the first five calls is a model, where Obj has references to Obj2 and Obj3. Let us inspect the atomic model manipulation operations executed beginning with the call to the clear rule:

```java
null

```//clear(Obj,Ref); //clear queries for all feature instances V that fulfill //the pattern findFeature(Obj,Ref,V) //this results in two occurrences, created in previous steps (Rel,Rel2)
delete(Rel);
delete(Rel2);
//isSet(Obj,Ref,IsSet)
//as all feature instances are deleted findFeature(Obj,Ref,V) has no matching
//IsSet=false is returned

The result is: IsSet = false that satisfies the assertion. □

**Proposition 11** Let Cls be an EClass, Ref a reference, starting from Cls. Moreover, let Cls2 be the type of Ref.

Let us suppose that the following sequence is executed on an empty instance model:

```java
null

```seq {
call create(Cls, Obj);
call create(Cls2, Obj2);
call create(Cls2, Obj3);
call add(Obj,Ref,0,Obj2);
call set(Obj,Ref,0,Obj3,OldVal);
call get(Obj,Ref,0,Val);
//assert: Val==Obj3
//assert: OldVal==Obj2
}

The result is the execution that Val == Obj3 and OldVal == Obj2.

**Proof** We have already shown that the result of the first four calls is a model, where Obj has references to Obj2. Let us inspect the atomic model manipulation operations executed beginning with the call to the set rule:

```java
null

```//set(Obj,Ref,0,Obj3,OldVal);
//set queries for the referenced element at index 0
//internalGet returns SI = Rel (Rel is the reference between Obj and Obj2)
update OldVal = target(SI) // = Obj2
setTo(SI,Obj3);
//the old value returned is Obj2

//get(Obj,Ref,0,Val);
//internalGet returns Rel
//get returns Val=target(Rel)=Obj3

The result is: OldVal = Obj2, Val = Obj3 that satisfies the assertion. □

**Proposition 12** Let Cls be an EClass, Ref a reference, starting from Cls. Moreover, let Cls2 be the type of Ref.

Let us suppose that the following sequence is executed on an empty instance model:

```java
null

```seq {
call create(Cls, Obj);
call create(Cls2, Obj2);
```
The result is the execution that \( \text{Empty} = \text{true} \) and \( \text{OldVal} = \text{Obj2} \).

**Proof** We have already shown that the result of the first three calls is a model, where \( \text{Obj} \) has references to \( \text{Obj2} \).

Let us inspect the atomic model manipulation operations executed beginning with the call to the `remove` rule:

// remove(\text{Obj}, \text{Ref}, 0, \text{OldVal});
// the first block of if is executed
// internalGet returns \( \text{Rel} \), the relation between \( \text{Obj} \) and \( \text{Obj2} \)
update \( \text{OldVal} = \text{target}(\text{Rel}) \) //\( = \text{Obj2} \)
delete(\text{Rel})

// isEmpty(\text{Obj}, \text{Ref}, \text{Empty})
// as all relations from \( \text{Obj} \) have been deleted,
// internalGet() will not succeed
update \( \text{Empty} = \text{true} \); 

The result is: \( \text{OldVal} = \text{Obj2}, \text{Empty} = \text{true} \) that satisfies the assertion.

**Proposition 13** Let \( \text{Cls} \) be an EClass, \( \text{Ref} \) a reference, starting from \( \text{Cls} \). Moreover, let \( \text{Cls2} \) be the type of \( \text{Ref} \).

Let us suppose that the following sequence is executed on an empty instance model:

\[
\text{seq} \{ \\
\text{call create(}\text{Cls}, \text{Obj}); \\
\text{call create(}\text{Cls2}, \text{Obj2}); \\
\text{call create(}\text{Cls2}, \text{Obj3}); \\
\text{call create(}\text{Cls2}, \text{Obj4}); \\
\text{call add(}\text{Obj}, \text{Ref}, 0, \text{Obj2}); \\
\text{call add(}\text{Obj}, \text{Ref}, 1, \text{Obj3}); \\
\text{call add(}\text{Obj}, \text{Ref}, 2, \text{Obj4}); \\
\text{call movee(}\text{Obj}, \text{Ref}, 1, 2, \text{MovedVal}); \\
\text{call get(}\text{Obj}, \text{Ref}, 1, \text{Val}) \\
\text{// assert: MovedVal = \text{Obj4}} \\
\text{// assert: Val = \text{Obj4}} \\
\} \\
\]

The result is the execution that \( \text{MovedVal} = \text{Obj4} \) and \( \text{Val} = \text{Obj4} \).

**Proof** We have already shown that the result of the first seven calls is a model, where \( \text{Obj} \) has references to entities \( \text{Obj2}, \text{Obj3}, \text{Obj4} \).

Let us inspect the atomic model manipulation operations executed beginning with the call to the `movee` rule:

//movee(\text{Obj}, \text{Ref}, 1, 2, \text{MovedVal});
// first call of internalGet returns SI=Rel3 (leading to \text{Obj4})
// next call of internalGet returns TI=Rel2 (leading to \text{Obj3})
update \( \text{MovedVal} = \text{target}(\text{SI}) \) //\( = \text{Obj4} \)
// call to remove() deletes Rel3
setTo(\text{TI}, \text{MovedVal})
// get(\text{Obj}, \text{Ref}, 1, \text{Val})
// get returns the target of Rel2 (that has been set to \text{Obj4})

The result is: \( \text{MovedVal} = \text{Obj4}, \text{Val} = \text{Obj4} \) that satisfies the assertion.

We defined at least one sequence for all model manipulation-related rules (add, clear, create, movee, remove, set), and the sequences covered also the most important query rules. Although this is suitable for basic validation of the semantics definition, it cannot be considered as a full test campaign.
3.4.5 Live model sharing

The model migration (or mapping) transformation introduced earlier has a strong drawback in real modeling environments, as it requires the storage of two model instances (an ECore and a VPM). In real environments, where model sizes are huge, this can be infeasible. In order to overcome this problem, a live VPM-ECore model integration solution has been developed that will be introduced here.

The architecture of the proposed solution is illustrated by Figure 3.5. The VPM model space stores the model instance, and serves the requests of VPM-based tools directly. There is an ECore to VPM bridge component based on the GTASM operational semantics for ECore that offers ECore/EMF interface towards the EMF tools, and acts as a memoryless proxy between the tools and the VPM model space. This bridge is the key component of the live model sharing concept.

The GTASM-based operational semantics is directly executable, therefore it is the core of the custom EStore implementation. The GTASM code is complemented with a Java wrapper that integrates the solution to the EMF environment.

3.4.6 Conclusions

A formal, operational semantics for ECore has been defined in the current section. The formalization is based on the GTASM language and results in an executable semantics. The formal semantics enables the precise description of ECore metamodels and models, and the semantics based systematic testing of model manipulation transformations and tools. The executable specification also allowed the definition of a live model integration solution that enables the runtime model sharing between ECore and VPM having solid formal foundation.

3.5 Workflow-based transformation integration

Model transformations play an important role in model-driven tool environments. Our project experiences from industrial and academic research projects [Git08,DIA,Kad08,12] have shown, however, that in complex environments, several transformations have to be composed in order to be able to provide proper tool support for the system designers.

Transformations in Model-Driven Development (MDD) tools are often integrated into workflows that implement complex, multi-step operations on models. The steps of these workflows are either automatic or interactive transformations, or general workflow steps like decision, fork, or join nodes. Most typically, workflows are implemented in an ad-hoc manner that hinders tool verification-validation and testing. In this section we propose a formal workflow model and a systematic, generative approach for the integration of transformation steps into workflows.

3.5.1 High level architecture of MDD tools

The high level architecture of the mapping tools follows a layered approach (see Figure 3.6). The tool relies on a modeling environment for model persistence. The tool-specific model management functionality is implemented in the so-called model management layer. It contains a main model manager that is responsible
for the model-wide functionalities, like loading and saving the models, maintaining the workflow state, and so on.

The step managers implement the model query and manipulation functions needed by a given mapping step. The behaviour of these managers depends on the type of the step they belong to. Step types include automatic transformation (in most cases, without GUI), and several typical interactive steps, like model import, manual model marking, and so on.

Each step has a separate step user interface that visualizes the step to the user. This also depends on the type of the actual step. The user interface components are integrated to the common user interface.

![High level architecture of a mapping tool](image)

The step user interface components are reusable elements, as the appearance of a step type is always the same. The step managers are only defined on interface level, but the implementation (including the specific model queries and manipulations) should be done for each actual step separately. The main model manager layer should also be customized in order to reflect the actual modeling workflow. The conclusion of this inspection is that the following components are to be implemented for each tool separately:

- Step managers including
  - model queries
  - model manipulation functions
  - event handling that converts model change events to user interface events (refreshing the UI on model changes)

- The main model manager, including
  - model load/save functionality
  - internal mapping workflow initialization

We will inspect the most important base technologies and implementation possibilities in the followings.

### 3.5.2 Key components to be implemented

#### Model queries

Model queries are collecting a set of model elements that fulfill a search criteria. During the definition of the steps, model queries are expressed in most cases a declarative way using for instance graph patterns or a constraint language (like Object Constraint Language (OCL) [Groi]) expressions.

Although the declarative specification is compact and easy to understand, it cannot be directly used without having a proper graph pattern or constraint evaluation engine; therefore in the traditional tools, the queries have been implemented using imperative programming languages that resulted in a more complex source code structure that is hard to debug or understand.
3.5. WORKFLOW-BASED TRANSFORMATION INTEGRATION

The current modeling environments offer, however, support for the execution of either graph pattern queries or constraint language expressions. This allows the direct utilization of the declarative description during the implementation. There can be, of course, cases when the declarative specification is not suitable, so the possibility of using direct, imperative implementation should still be maintained.

Event handling

The other aspect, strongly tied to model queries is the behavior of the tool in case of model changes. The expected behavior from the tool user point of view is that the user interface should immediately reflect the changes of the model. For instance, if a query contains the unmarked elements in case of a marking step, after the execution of a new marking, the query should be updated and the element should be removed from the result set.

There are two main strategies for the implementation of live, or event-triggered queries. The first - more traditional - approach is the re-execution of queries on model changes. The problem in this case is the degree of context sensitivity wrt. the type of model changes. The most simple approach is to re-execute the query after each model change, while the most sophisticated one is the update of the results set based on the analysis of the changes. Of course, several intermediate solutions can also be implemented (handle some classes of changes in an intelligent way, and backup to full re-execute in case of complex changes).

The second approach is simpler from the implementation point of view, but requires support for event-triggered queries from the modeling environment side (like in [VVS06,RBÖV08a]). In this case, the query execution engine itself listens for model changes and updates the results set automatically. The tool has only to listen for result set changes and to update its user interface if needed.

Model manipulations

Model manipulations are described mostly using graph transformation or ASM rules. These can directly be executed using the appropriate engine. In some cases, however, the implementation may necessitate the usage of a traditional programming language, so there should be an option that allows the introduction of custom modules for model manipulation.

3.5.3 Generative Framework for Tool Synthesis

Based on the specification formalism introduced in Section 5.4 and the implementation possibilities introduced in the previous sections, a generative framework can be established that supports the automatic synthesis of PIM-PSM mapping tools.

The generative framework (and also the generated tools) builds on a basic set of code generation transformations that implement the basic functionalities of the tool on top of a selected tool platform, and a set of step code generators that are specific to the step types actually defined. These step generators generate the implementation of the individual steps and integrate them to the tool.

Figure 3.7 illustrates the elements of the framework, and the layers of abstraction of the generative architecture. These layers will be introduced in more detail in the followings.

Workflow language definition layer

This abstraction layer contains the metamodel of the integration workflow language that is used to describe the actual workflow. The metamodel is a simple workflow language, with tasks, transitions, and decisions. The tasks are specialized according to the various integration step types as described in Section 5.4.

The most important (and problem specific) aspect of the metamodel is the description of queries and model manipulations. Model queries can be described either as graph patterns, or constraint language expressions, or using imperative programming language source code. Model manipulations can be described using graph tranformations, ASM rules, or by source code.

This layer also contains the platform independent metamodel of the metamodeling language that will be used for the specification of modeling languages used by the tool (like ECore, or MOF) and its implementation-specific extension metamodel. The second metamodel describes the implementation details of the modeling language on top of the target tool platform.

Workflow model definition

This abstraction layer contains the description of a specific MDD tool. The description (model) is instance of the integration workflow metamodel. It should be noted that this model references the modeling languages
that are instances of the language meta-metamodel.

![Image](image.png)

**Figure 3.7: Overview of the tool generation framework**

**Tool generation layer**

The tool generation phase is an automatic step where the generator reads the workflow model and generates the implementation for all components that together build up the target tool for the specified workflow. In parallel, the model persistence generator generates the implementation of the persistence layer. This layer stores the models runtime, and also supports the execution of model queries and transformations.

**Tool implementation layer**

On this layer, there is the concrete implementation of the tool, and of the associated modeling languages. The tool is often integrated into a generic tool framework (like Eclipse [Foue]) in order to get base services like resource management, generic GUI, version control among others.

**3.5.4 Conclusions**

A generative framework for the synthesis of workflow-driven MDD tools has been introduced in this section. The framework supports the declarative definition of such workflows, and generates the implementation using the state of the art model management, model transformation, and tool implementation technologies. The main benefit of this approach is the short time-to-market of the mapping tools that enables the rapid prototyping of the tools, and also the utilization of model-driven techniques in cost-sensitive domains.

**3.6 Applications**

**3.6.1 VIATRA2 Framework**

VIATRA2 is an open-source model transformation framework and is part of the official Eclipse Generative Model Transformations sub-project [Foub]. It implements the VPM metamodeling concepts and the VTCL language that has already been discussed in this paper.

VIATRA2 has been successfully used in different national and international research projects (like DE-COS, DIANA, SENSORIA, BelAMI). OptXware Research & Development Ltd. also maintains an industrial edition of the framework called VIATRA-I that is used in industrial projects for the development of model-transformation based tools.
3.7 CONCLUSIONS

VIATRA2 incorporates several techniques that have been discussed in this Chapter. Its model transformation engine implements the advanced graph transformation constructs introduced in Section 3.2. The model migration and live model sharing techniques introduced in Section 3.3 are currently part of the VIATRA-I edition but some of them will be made open-source by OptXware Ltd.

3.6.2 DECOS Toolchain

The DECOS tool chain [DEC] supporting the model-driven development of dependable embedded systems, benefits from the achievements introduced in the current chapter. The PIM domain-specific editor incorporates the design pattern framework in order to increase the productivity of PIM model designers. A standard pattern library has been developed in the framework of the DECOS project that has also been extended by users of the editor. That way the usability and extensibility of the framework has been proved.

3.6.3 PIM-PSM mapping tool synthesis

The generative approach introduced in Section 3.5 has been implemented based on the Eclipse Platform, the Eclipse Modeling Framework, and on VIATRA2. The tool has been validated by re-implementing the DECOS mapping tool, and the prototype GENESYS proof-of-concept mapping tool.

The potential of the approach will be further investigated by the INDEXYS project where the solution will serve as a basis that enables the rapid development of custom mapping tools for different domains (automotive, railway, aerospace). An important issue is that the tools can even be customer-specific allowing for the utilization of custom development workflows.

3.7 Conclusions

We have introduced several constructs that support the composition of graph transformations on different levels. First, graph pattern and transformation level composition has been introduced in order to facilitate the reuse of these elements through multiple rules and transformations. This results in a more compact transformation code (reduction in our practice has been typically 50 to 80 percent in terms of source lines of code), and better testability due to improved modularization. The traditional pattern and rule constructs have been extended with direct support for design patterns that enables the easy, declarative specification of design patterns for any modeling languages, and the execution of patterns using the capabilities of the graph transformation engine.

We have introduced a transformation level composition in order to be able to execute sequential transformations more effectively (in terms of execution time and space). This result enables the reuse of importer components during transformation design and runtime, and also the runtime integration of different metamodeling environments.

We proposed a formal semantics for ECore, the de facto standard metamodeling environment for domain-specific language development. The precise operational semantics enables the systematic test generation for ECore based models and transformations, and enables the live integration of ECore and VPM and the utilization of existing model transformation technologies in the ECore environment.

Finally, we discussed the transformation workflow level integration of model transformations. This results in a framework that supports the high-level definition of workflows composed of automatic and interactive transformations that is complemented by a generative solution in order to synthetise model-driven tools from the high-level descriptions.

Most of the results introduced in this chapter are already incorporated into the VIATRA2 open-source model transformation framework and have been used in different industrial and academic research projects. These results establish a solid infrastructure for the model-driven development and analysis methodologies and techniques that will be introduced in the upcoming chapters.
Contribution 1  I have developed novel constructs for the composition of graph transformations on different levels. In particular, the following novel procedures were elaborated:

1.1  I defined composition mechanisms on graph pattern and transformation rule level that extend the traditional graph transformation approach and reduce the complexity of transformation specifications. [2, 7, 8, 10]

1.2  I have developed a mechanism for the transformation-level composition of graph transformations in order to achieve better runtime performance and to allow the reuse of transformations. [9]

1.3  I defined a formal, executable operational semantics for the ECore metamodeling environment. [2, 7, 9]
Chapter 4

Embedded Systems

4.1 Introduction

Embedded systems (ESs) have a major share in electronics applications. Both functional and non-functional (e.g., dependability [ALRL04], timeliness, performance, power, cost, size, weight etc.) attributes play an equally important role in general for ES.

A distinguished class of ESs deals with safety-critical applications demanding the compliance to dependability requirements as a hard priority constraint. As the assurance of an ultra-high reliability of the individual electronic components is extremely costly, thus confined to applications requiring the topmost Safety Integrity Level (SIL-4), typical solutions rely on redundant architectures. Replication uses multiple instances implementing the same functionality and assures, that after a fault in a unit (or a few of units) the remaining fault free ones still execute the designated functionality safely.

Component-based system composition has a long tradition in ES engineering in order to facilitate cost effective product families and IP reuse. The architecture composition paradigm dominating ES design develops over time in synchrony with component design and manufacturing [Rus99].

Traditionally, the notation of a component was associated with a dedicated hardware unit performing a specific functionality (frequently referred to as an Electronic Control Unit (ECU)) with a limited flexibility (configurability). The initial architecture composition paradigm was federation – creating systems by interconnecting ECUs in a peer-to-peer way or by networking them. The functional decomposition and the architecture of a federated system were strongly correlated.

Safety-critical systems used the physical replication of the corresponding ECUs as a basis.

However, the growing amount of functional features to be realized by embedded systems lead to a drastic increase in the number of components in typical ESs. Moreover, their interaction (and the underlying communication network) became overly complex. The large number of components and the complexity of the system architecture led to decreasing system dependability.

The use of dedicated ECUs offering only a limited flexibility (configurability) became to a major limiting factor in the terms of manufacturing volumes and accordingly in production cost efficiency. The appearance of low cost and high computational performance programmable devices changed the ES implementation paradigm drastically.

General purpose ECUs substituted the former dedicated ones with the exception of extremely high volume products using dedicated microelectronic solutions. ES specific devices, like single chip microcomputers and controllers and DSP units entered the market complementing general purpose microcomputers. The high volume production of HW components and the wide (re)use of standardized software components (SW-Cs) drastically reduced the unit prices.

Programming became the main means for all the implementation of the designated functionality, basis for system customization by configuration and in-field augmentative maintenance, as well.

Resource sharing evolved as a promising way for cost reduction by exploiting the excess computational power offered by modern ECUs after deploying the dedicated functionality onto them.

The dominant system architecture composition paradigm became integration, which systematically co-deploys onto the same ECU multiple SW-Cs implementing potentially different sub-functionalities even of different non-functional requirements. The different ECUs are interconnected by a communication network structured by active devices.

Replication is still the main means to achieve a high-level of dependability. Deployment diversity allocates the replicas of a safety-critical functionality onto different HW nodes in order to protect them.
against correlated impacts originating in single-node hardware faults and assures a similarly high-level of dependability as dedicated ECUs replication in integration-based composition.

However, resource sharing is another source of error propagation as functionally independent software components may interfere especially in the case of a fault. For instance, it would be a highly unwanted side effect of a fault leading to a crash of a non-critical task (i.e. a task belonging to the car entertainment system) would have any impact on a critical functionality (like electronic brake).

Rushby [Rus99] advocated a design principle called partitioning, which calls for well-defined boundaries between modules to ensure the continuity of operation in the presence of faults/errors. The goal of partitioning is to create fault containment units or partitions such that behavior in a partition is left unaffected by the behavior in another partition, including any faulty behavior in it.

Resource sharing may allocate private resources to the individual SW-Cs for an exclusive use either the spatial or temporal dimensions, like allocating a private memory segment to a task or allocating computing and communication time slices to them. As the corruption of the exclusivity due to a fault may create dangerous interference between SW-Cs, mechanisms carefully controlling interactions and accesses to shared resources along both these dimensions are needed, similarly to task separation in all multi-threaded operating systems.

Spatial partitioning can be protected by using a combination of HW and SW techniques such as use of memory management unit or of different wavelengths in optical communication. Temporal partitioning can generally be achieved through enforcing the scheduling policy even in the presence of faults.

Perfect partitioning completely isolates any two functionally independent applications/components bidirectionally and symmetrically, this way partitioning in an integrated architecture helps emulate a corresponding federated architecture composed of independent tasks. Accordingly, it allows by principle the integration of a SW-C of an arbitrary criticality level without any further check of the relation between its criticality level vs. that of the others SW-Cs deployed onto the same node.

However, perfect partitioning is the rather costly both in the terms of specific computing and communication node architectural features and computational resource overhead for a perfect temporal and spatial isolation. Accordingly, simpler devices cannot support it in a full extent. Moreover, partitioning is unable to compensate faults in the shared resource or the partitioning mechanism potentially leading to correlated errors in all SW-Cs relying on them. Accordingly, even a perfect partitioning has to be complemented by replication of at least of those components serving critical functionalities.

At the same time, aiming at a perfect partitioning of a system independently of the criticality is frequently an overly strong requirement. Typically, non-interference requirements are asymmetric, as the safety-critical SW-Cs have to be protected of influences originating in low criticality SW-Cs, but not vice versa. For example, a corruption of the entertainment subsystem due to a fault in a flight control subsystem may be acceptable from the safety point of view, but the other direction would be catastrophic.

As integrated systems become important in many different domains, like the automotive, aerospace, railway, and industrial process control, we will assume this architecture in the upcoming chapters.

4.2 ARTEMIS Challenges

Advanced Research & Technology for EMbedded Intelligence and Systems (ARTEMIS) is a European Technology Platform started in 2004 by the European Commission, a set of member states, and the main players of the European embedded systems industry. The ARTEMIS association published a Strategic Research Agenda in 2006, that defines the mid- and long-term goals of embedded systems related research and development in Europe. The research agenda is complemented with annual Work Programmes that define the most actual research goals that will also be financed by the Commission and its member states in the framework of ARTEMIS.

Figure 4.1 illustrates the common objectives of ARTEMIS. Common reference architectures and designs, and intellectual property reuse are key goals of the programme. Connecting to these items, a distinguished objective is the research on novel development methods and tools that support the realization of embedded systems fulfilling both functional and non-functional requirements, and to facilitate IP reuse in different phases of the development process in order to improve time-to-market and provide competitive advantage to the European embedded systems industry.
4.3 Event and Time Triggered Paradigms

There are two main paradigms in the field of real-time embedded systems, these are the time-triggered (TT) and the event-triggered (ET) approaches [Kop97]. Both of them play an important role in our discussion; therefore we will introduce both of them briefly. Before that, a common, generic timing model for communication will be defined in order to give a basis for timing related discussion.

4.3.1 Timing Model of a Software Component

A software component is modeled as a black box, so there is no information about its internal behavior, the model only contains the inbound and outbound message interfaces.

The temporal behavior of a component is specified using several non-functional properties. These include the worst-case execution time ($t_{WCET}$), period (in case of periodic behavior), debounce time and priority (in case of event-triggered approach) of the component.

In order to be able to analyze the temporal behavior of the application, the following timing model will be used for the individual components.

**Definition 16 (Temporal behaviour of software components)** If the execution of a software component starts at $t_{start}$, it consumes all input messages from its inputs at $t_{start}$, and produces all output messages at the end of its execution that is due at $t_{start} + t_{WCET}$.

The ET and TT specific attributes will be discussed in the appropriate sections.

4.3.2 Timing Model of Communication

Inter-component communication is a complex mechanism that usually involves the software components, middleware (or operating system) resources, and some kind of communication hardware and medium. In order to be able to analyze this aspect on various levels, a generic timing model will be introduced in this Section.

Figure 4.2 illustrates the flow of inter-component communication. The sender component produces a message that is written into a middleware buffer. After that, the content of the buffer is written to the (hardware) buffer of the communication controller that transmits the message through a communication medium.

On the receiver side, the communication controller stores the received message in its buffer. This will be read by a middleware process, and the message content will be copied to a middleware buffer. After that, the recipient component will be triggered for execution.

There are several phases in this scenario, each having its own timing property.

- $t_{appl}$ is the execution time of the application-to-middleware buffer copying of messages. This is either done by the application (via middleware API calls), or by the middleware by copying the output buffers of the application to its own buffers.
**Figure 4.2: Timing model of inter-component communication**

- $t_{mwt}$ is the time that is needed for packing the outgoing messages to frames, and performing conversions (if needed).
- $t_{drvt}$ is the time that is needed for the copying of the outgoing frames from the middleware buffer to the (hardware) buffer of the communication controller.
- $t_{prott}$ is the protocol-specific transmission time that elapses between the preparation of the outgoing frame and the reception of the frame on the receiver side.
- $t_{drvr}$ is the time that elapses during the copying of the received frame from the controller buffer to the middleware.
- $t_{mwr}$ is the time needed for the unpacking of messages from the frame and performing conversions/voting, if necessary.
- $t_{appr}$ the delay after the message is available to the activation of the receiver component.

In order to capture the time that is elapsed before transmission, and after transmission the following parameters will be used:

\[ t_{bt} = t_{appt} + t_{mwt} + t_{drvt} \]

\[ t_{at} = t_{drvr} + t_{mwr} + t_{appr} \]

The total time of transmission from the sender component to the receiver is:

\[ t_{t} = t_{bt} + t_{prott} + t_{at} \]

The calculation of these timing parameters depends on the actual communication protocol.

### 4.3.3 Event Triggered Systems

A system is *event-triggered* if the internal processing is triggered by asynchronous events resulting in the need for dynamic scheduling. The frequency of task executions and message transfers is determined by the frequency of the trigger events.

Several component and message properties should be introduced in order to be able to analyze the temporal behaviour of the system. Components have a *debounce time* property that specifies the minimum interval between two consecutive activations of the component. If the input event is arriving with higher frequency, it will be delayed by the runtime environment. Component *priority* defines an ordering between components if more than one is competing for execution.
4.4. HOST AND NETWORK CHARACTERISTICS

Message debounce time is similar to task debounce time, and specifies the minimum required delay between two consecutive transfers of the message on a communication network. Message priority defines an ordering between messages that is used by the communication subsystem to determine the order of message transmissions.

4.3.4 Time-Triggered Systems

The time-triggered paradigm is a fully synchronous approach to distributed system design. Such systems are built around a TT network (core network) used both for message transmission and for synchronization and provides a global time base. The component execution on the nodes is also synchronized using the global time. The processing of the system is periodic, and the period is called cluster cycle.

The cluster cycle is usually subdivided into a number of message rounds. The round determines the period of time synchronization and node membership calculation. Each message can be sent only once per round, that means that the length of the round determines the minimal message transmission period. Both components and messages have pre-defined periods that specify the execution frequency of components and transmission frequency of messages, respectively.

The schedule for the system, specified at design time, consists of two types of elements: the communication schedule and the individual node schedules. As all of the nodes share the same network for broadcasting their messages, the communication schedule (defining when to transmit and receive messages) is identical for all nodes, so all nodes contain the information on the transmission instance of all messages. Node schedules are different, as different software jobs run on them. A configuration file describes the node schedule. The operating system uses it to execute the jobs in a TT way.

4.4 Host and network characteristics

The typical embedded control systems are networked (mostly involving multiple embedded networks), and the computing nodes are resource constrained. Systems can include several communication networks and dozens of computing nodes.

The typical architecture of a system contains a high capacity core network that interconnects a set of general-purpose power nodes offering relatively more computing resources. There may be other networks (called field buses) that interconnect a power node with several peripheral nodes. The later ones have a limited capacity, and are often dedicated smart sensors or actuators.

4.4.1 Network protocols

The network protocols that are typically used are also varying depending on the application and the bandwidth and dependability requirements. There are two large groups of protocols: event-triggered (CAN [SI03], Ethernet [EEE08], EIA-485 [IATIA98], TTP/A [ea02], etc), and time-triggered (AFDX [RIARI05], TTP/C [CT01], FlexRay [Con05], LIN [Con06], TT-Ethernet [TTTTT08]).

- Event-triggered protocols are asynchronous, in the sense that each network node can send packets to the network without any restrictions in the temporal domain. If two or more nodes try to transmit simultaneously, a collision occurs and the protocol facilitates some kind of collision resolution. This means, that the communication scheme is dynamic, but the latencies can vary due to collisions and concurrent communication traffic.

- Time-triggered protocols are synchronous as they have a design-time defined static communication schedule. Each message is periodic, and can only be transferred at a given time frame specified by the schedule. This results in a collision free operation and deterministic but, compared to the event-triggered case, longer network latencies. The modification and extension of such systems requires the re-design of the schedule.

4.4.2 ECU resources

Computing nodes usually contain one or more communication controllers that implement network protocols, (non-volatile) code memory, data memory and one or more microprocessors. Most typically, embedded processors have several on-board peripherals that are used to connect sensors/actuators to the system. The set of peripherals can also be extended using external ones connected to the microprocessor bus.
4.4.3 ECU middleware

With the introduction of complex runtime architectures for dependable real-time systems like FT-CORBA [Gro04a], OSEK [Grob], AutoSAR [Alla], or ARINC 653 [RIARI06], the platform middleware complexity of these systems has significantly increased. On the other hand, the standardization of runtime architectures leads to better reusability of application level software components.

4.5 Domain specific Languages

4.5.1 SysML

SysML [Groa], based on the widely known UML [Grom] is the systems modeling standard of the OMG. It has been designed to serve as a platform independent modeling language for systems engineering.

The language puts emphasis mainly on the architectural system design based on modified and new structural diagrams (static structure and internal block diagram). It offers a componentization technique based on hierarchical blocks. The flow between these can be either information or control flow like.

Besides the static structure, SysML supports also the behavioral description of system components using customized variants of the UML activity, state chart, and interaction diagrams.

4.5.2 AADL

AADL (Architecture Analysis and Design Language) [SAE] is the standard architecture description language of the aerospace domain (AS5506). The intended usage of the language is the early and repeated analysis of a system’s architecture.

The standard defines a description language that uses concepts from the multi-threaded software development domain (process, thread) and offers a component-based design framework. The structural description is combined with communication-oriented aspects (communication and control flows), and is extended with an execution platform description (processors, memories, busses).

4.5.3 Matlab Simulink and StateFlow

Simulink [Mat] is a dataflow network based modeling environment that is integrated into the Matlab tool suite. The language also contains a state machine modeling notation called StateFlow. Simulink is the most widespread behavioral modeling tool in the automotive and industrial control domains. However, its utilization in safety-critical applications is limited, because there it lacks a precise, formal semantics.

Simulink is often used in conjunction with an architectural modeling language, like AutoSAR, and allows the specification of software component internal behaviour. Simulink offers simulation services for the validation of early system design in the developer environment, and also by involving real sensors and actuators (hardware-in-the-loop simulation).

4.5.4 SCADE

SCADE Suite [Teca] is an environment for the development of safety critical software. It supports a model-based development paradigm, where the model is the software specification.

The modeling language of SCADE is a formal, synchronous dataflow language that is based on Lustre [HCRP91]. The dataflow notation is complemented with a safe state machine language that allows the description of reactive behaviour.

The model created using the SCADE suite can be exercised by simulation, using the same code as the embedded code. Formal proof techniques can also be applied to the model to detect corner bugs or prove safety properties. Code is automatically generated from the model with the Qualifiable Code Generator so the code is correct and up to date by construction.

Today, SCADE Suite is the de-facto standard for the creation of safety-critical embedded software in the European avionics industry and is the emerging standard for the creation of critical-embedded software in the automotive industry.

4.5.5 LabView

LabView [Ins] is an environment for the development of data acquisition, measurement, and process control applications. The goal of LabView is to provide a high level, intuitive programming interface for real and
virtual instruments and data collector hardware allowing even non-IT researchers to quickly set up complex, software based measurement systems.

The language of LabView is based on the data flow network formalism, and contains a comprehensive library of special components that can be reused during programming (including virtual and real instruments, data filtering, and control structures). LabView supports different target platforms for program execution including FPGAs, PDAs, embedded boards, and Windows PCs.

4.5.6 **UML MARTE**

The *UML Profile for Modeling and Analysis of Real-time Embedded Systems* (MARTE) [Grok] is an OMG draft standard for model-driven development and analysis of real-time systems. It is intended to replace the existing UML Profile for Schedulability, Performance, and Time.

![Figure 4.3: Structure of the MARTE Profile](image)

The structure of the MARTE profile is illustrated by Figure 4.3. The foundation part of the profile contains the definition of basic infrastructure. The *core elements* package defines the fundamental infrastructure for the profile. The *time modeling* package is a framework for precise time description, the *generic resource modeling* package discusses the description of various (hardware or software) resources, and the allocation package specifies component to resource allocation modeling. The non-functional properties package will be discussed in more detail, as we will use it in the upcoming chapters.

The *MARTE design model* provides a *generic component model* inspired by SysML, a *high-level application model* that support real-time and embedded features modeling, and a *detailed resource model* for the fine-grained description of system resources.

The *MARTE analysis model* contains a core *generic analysis model* that addresses the common concepts of the different analysis domains, and specialized models for *performance* and *schedulability* analysis.

**Uniform treatment of Non-Functional Properties**

Several existing languages defined non-functional properties (NFPs) for model elements like software components, messages, and so on that are necessary in the given application domain. These properties, however, form a fixed set that cannot be easily extended and limits the expressiveness of the models.

Several attempts have been made to define a common non-functional property definition formalism, but the most promising is the upcoming UML MARTE profile. The NFP package of the profile contains a generic metamodel for the definition of NFPs, including qualitative and quantitative ones (Figure 4.4). These properties can be instantiated and attached to any model element of the system model in order to express new aspects of non-functional characteristics.

The VPM representation of the NFP metamodel is as follows:

```plaintext
entity(NFP_Type);
entity(NFP);
relation(type,NFP,NFP_Type);
```
The NFP definition will be used as a meta meta model for the definition of non-functional properties in the domain models.

The basic definition framework should be, however, extended with a standard taxonomy of the properties [WE95] in order to have a common semantics of the models.

4.5.7 AutoSAR

AutoSAR (Automotive Open System Architecture) [Alla] is the evolving standard in the automotive applications domain. Most of the important players in the industry (including car manufacturers, part suppliers, and tool vendors) participate in the standardization process. It should be noted that the AutoSAR standard is not only a modeling language, but it covers nearly all aspects of automotive electronics development, including the standardization of embedded software components, architectures, and runtime environments. The modeling notation of AutoSAR integrates the architecture, platform, and hardware-software integration modeling aspects in a single language. We will focus on the software architecture definition part (called SoftwareComponentTemplate in the standard) in the followings.

4.5.8 DECOS PIM

A stand-alone modeling language for platform-independent software modeling has been developed in the framework of the DECOS [DEC] project. The language has been developed in parallel with the SysML language and is aligned with it [20], although as the target domains were automotive, aerospace, and industrial process control, DECOS PIM contains only the constructs that are relevant in these application areas.

The main advantage of the DECOS PIM language is the integration of functional and non-functional (performance, dependability) properties in a single language. While supporting all major concepts that can be found in other architecture description languages (see Table 4.1), DECOS PIM directly supports the specification of the main non-functional system characteristics.

The DECOS PIM language is defined using its (MOF) metamodel consisting of three packages (see Figure 4.5). The `functionality` package contains the model elements that describe the purely functional part of the design. The other two packages depend on the functionality package, because they use some of its...
elements. The performance and dependability packages introduce elements that describe the performance and dependability related aspects of the elements, respectively.

![Figure 4.5: Packages of the DECOS PIM Language](image)

The main architecture of DECOS PIM is illustrated by Figure 4.6. The root of a model is the Distributed Application Subsystem (DAS) representing a high level, distributed application or function. It is reflected in the PIM so that a DAS is a set of logically cohesive jobs. This is denoted by the aggregation relation from Job to DAS.

A DAS has operating modes. It is represented by the aggregation relation from OperatingMode to DAS. Some examples of operating modes are: normal operation, startup, flashing, maintenance, degraded. The specific jobs of the DAS run in these operating modes. It means that a different set of jobs can be run in different operating modes. It is represented by the association runsIn between Job and OperatingMode.

Possible operating mode changes are described in the model by using the after association that denotes a transition between the operating modes. The initial operating mode always has to be defined. This is denoted by the initial association between DAS and OperatingMode.

Jobs are stand-alone, schedulable software entities which communicate with each other. There are two types of jobs: time-triggered and event-triggered. It is described by the inheritance relation between Job and TimeTriggeredJob as well between Job and EventTriggeredJob.

Jobs can have state variables. It is described by the aggregation relation from StateVariable to Job. A state variable is a data element, its data type and length is supplied by the designer. The initial value of the state variable can be defined too. A state variable can be fixed or variable in length. The possible values of state variables form the state space of the job. The values of the state variables at a particular instant denote the state of the system at that instant.

The declared state is a projection of the state that is relevant for the future behaviour of the system and is specified as an expression over the set of state variables. It is denoted by the association between StateVariable and DeclaredState and the association class StateDeclaration. State declaration is a specialization of assertion.

Jobs can have interfaces. It is described by the aggregation relationship between Interface and Job. An interface is a common boundary between two subsystems and consists of one or more ports. There are five types of Interfaces, Service Providing Linking Interface (SPLIF), Service Requesting Linking Interface (SRLIF), Controlled Object Interface (COI), Configuration Planning interface (CP) and Diagnostic and Management interface (DM). SPLIF, SRLIF, COI, CP and DM are specializations of Interface. Four of these five interfaces serve for job-job communication. The controlled object interface is for communicating with sensors and/or actuators.

Sensor and Actuator are specializations of Resource. Resource itself is an abstract entity; it will never be directly used by the designer. It is used to group resource-like entities together. Two types of resources are defined, Sensor and Actuator, but this is a primary point of extension, since the product of resource modeling activities can appear here in a complete tree structure.

An interface consists of one or more ports through which jobs communicate. It is described by the aggregation relation between Port and Interface. A port is an access point where a job reads/consumes an input message (input port) or writes/produces an output message (output port). A port to a time-triggered
virtual network is a state message port, while a port to an event-triggered virtual network is denoted as an event message port.

Through the ports, messages can be sent or received to or from other ports. Two ports can be used for communication if they belong to different jobs and use the same message type. Ports can see the state variables of the job they are assigned to. It is described by the isVisible association between StateVariable and Port. Actually, the state variables of the port are a subset of the state variables of the job the port is assigned to. This way a clear distinction can be done between the state of a LIF and the state of the job.

Over the state variable expressions can be defined that poses restrictions on the value of the variables. This is represented by the class AcceptanceCriteria holding the expression. Using this, it can be checked, whether the interface is in a specific state (i.e. if all the expressions on the values are satisfied). This way operational input assertions on interfaces / ports can be realized. Please note that as a special case, an expression can contain a constant value.

The image or the change of state variables can be packed in messages. This way a message can carry the value of one or more variables, and variables can be transmitted through multiple messages. The partID identifies the parts of the message. In case of communicating ports, the PartIDs of the message parts must match. StateMessages carry the image (i.e. the exact copy) of state variables while EventMessages carry the change of state variables. It is represented by the associations Image and Delta respectively.

A state message is a periodic message that contains state observations. An observation is a state observation, if the value of the observation contains the state of a real-time entity. The time of a state observation denotes the point in time when the real-time entity was sampled. The handling of state messages occurs through an update in place and non-consuming read.

An event message is a message that contains event observations. An event observation contains the difference between the old state (the last observed state) and the new state. The time of the event observation denotes the point in time of the state change. In order to maintain state synchronization, the handling of event messages requires exactly-once semantics. The arrival of an event message usually gives rise to a control signal, which triggers subsequent computational and communication activities.

The entity called Message is abstract, therefore it cannot be instantiated. It serves only to describe the common features of state messages and event messages. Both state- and event messages are subtypes of message. Messages can depend on each other, this is represented by the self-association.

A DataStream is a specialization of messages, a data stream consists of periodically sent data chunks. It is described by the specialization relation between Message and DataStream.

ONA (Out-of Norm Assertion) and Symptoms are used by the Diagnostics. A symptom is an expression
over interface state variables. That is why Symptom is a specialization of Assertion and has an association to StateVariable. An ONA is an expression over symptoms. That is why ONA is a specialization of Assertion and has an association to Symptom.

Restriction expresses an offline restriction, that is, an assertion that will be used as a fact by the analysis.

4.5.9 Summary

All of the languages introduced here support the definition of component-oriented application models, however there are differences in the details of modeling. Component models play an important role in the upcoming chapters; therefore a comparison of modeling concepts has been done and is presented in Table 4.1. It should be noted that UML MARTE is not included, because its generic component model is nearly identical with that of SysML.
Table 4.1: Comparison of architecture description languages

<table>
<thead>
<tr>
<th>Concept</th>
<th>AutoSAR</th>
<th>SysML</th>
<th>AADL</th>
<th>DECOS PIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software component</td>
<td>ComponentType</td>
<td>Block</td>
<td>Process</td>
<td>Job</td>
</tr>
<tr>
<td>Component instance</td>
<td>ComponentPrototype</td>
<td>Block instance</td>
<td>Subcomponent reference</td>
<td>none</td>
</tr>
<tr>
<td>Component hierarchy</td>
<td>CompositionType</td>
<td>nested blocks</td>
<td>nested processes</td>
<td>nested jobs</td>
</tr>
<tr>
<td>Component interfaces</td>
<td>PortPrototype</td>
<td>Port</td>
<td>DataPort</td>
<td>Interface and Port</td>
</tr>
<tr>
<td>Component inter-connection</td>
<td>Connector</td>
<td>Service and data flow</td>
<td>Flow</td>
<td>via Messages</td>
</tr>
<tr>
<td>Connection type</td>
<td>Client/server (operation invocation) or Sender/Receiver (data flow)</td>
<td>Operation (control) or Flow (data)</td>
<td>Subprogram call or Flow (data)</td>
<td>Event-triggered, Time-triggered, data flow messages</td>
</tr>
<tr>
<td>Constraints</td>
<td>-</td>
<td>Constraint block</td>
<td>-</td>
<td>Assertions, Value restrictions</td>
</tr>
</tbody>
</table>
Chapter 5

Model Driven System Design with Quality Aspects

5.1 Introduction

The MDA concept has been originally proposed by the OMG as a standard process for model-based systems development. It is extensively used in enterprise and desktop software development, but not in the embedded systems field. In this chapter, we will discuss the elements of the process, propose modeling notations for the different assets of the development, and a novel, interactive and iterative development approach for model-based hardware-software integration.

5.2 Motivating example: The DECOS tool chain

We will introduce the tool chain that has been developed in the framework of the DECOS EU IST Framework 6 Project [DEC]. The project aimed at the definition and implementation of a model-driven tool environment to support the development of integrated, safety-critical embedded systems. We use this tool chain to introduce the concepts of MDD tools that will be used throughout this chapter.

As shown in Figure 5.1, the tool chain adopts a model-driven approach. First, the PIMs of the DASs are created, which serve two purposes. On the one side, together with the specification of the target cluster
(Cluster Resource Description - CRD) and other information (job size etc.), it is used to derive configuration and scheduling information, as well as to generate the PIL (Platform Interface Layer - configurable middleware), by transforming the the PIMs into the the PSM. In Figure 5.1, "Candidate PSM" is denoted rather than "PSM", because if scheduling fails, another allocation has to be chosen. On the other side, the PIMs are used to guide the development of jobs, by modeling their behaviour. Finally, the results of both activities are integrated to achieve the target executables, which can then be downloaded to the application cluster.

5.2.1 PIM modeling

The purpose of the PIM is to formalize the functional, dependability, and performance requirements of the DAS in an implementation platform independent manner. It is the place of the first steps of system architecture conceptualization. DECOS platform services - both at core and high-level - are handled in an abstract form that is easy to use and understand at this level of design.

Two solutions are provided for easing the generation of a PIM:

- to use the same UML tool as for high-level system design. In this case, the DECOS-PIM XML file is generated from the XMI output of the UML editor. Currently Rational Rose 2003 and Rational Software Modeler are supported.
- to use a Domain Specific Editor (DSE), which allows for creating only metamodel compliant PIMs. Such a DSE has been implemented under the Eclipse technology. It runs directly inside of Viatra2 which is the selected tool for PIM-PSM mapping. This solution provides domain specific editor commands and consistency checks in order to improve the quality and speed of the design process.

5.2.2 CRD and its generation

It is the purpose of the so-called Cluster Resource Description (CRD) to capture the relevant characteristics of the platform for the hardware-software integration in the DECOS design flow. These characteristics include among others computational resources (e.g., CPU and memory), communication resources, and dependability properties.

In order to ease CRD creation, a graphical, domain-specific modeling environment is developed, using the Generic Modeling Environment (GME). GME is a configurable framework for creating domain-specific modeling environments [LMB01]. The configuration of GME is performed via the Hardware Specification Model (HSM), a meta-model which formally describes the targeted modeling domain, i.e. it describes the entities, its attributes, the relationships, and constraints that can be expressed with and that are validated by the resulting modeling environment.

5.2.3 PSM generation

Being (still) a model, the main purpose of the PSM is to precisely specify which application jobs are to be assigned to which cluster nodes, under consideration of all constraints defined in the PIMs of the DASs and the available resources described in the CRD.

The PSM generation process (see Figure 5.2) encompasses a number of steps like PIM marking, feasibility checks, and the allocation process. The significant part of the mapping process is to allocate jobs with different criticality to a shared HW platform (HW nodes) subject to constraints and requirements of fault-tolerance and real-time.

PIM marking is necessary for incorporating additional information to the PIMs that reflect designer decisions (hardware sensor/actuator allocation, job pre-allocation) and legacy information (job interface type, message protocol definition, etc.) The result is a marked PIM containing elements and associations reflecting the additional information. Feasibility checks are executed both after the marking and after the allocation step to validate the (partial) models in order to achieve early detection of design problems. If a design constraint is violated, the designer can step back in the PIM/PSM mapping process and modify markings and/or extra-functional requirements to get a feasible system design. The main (automatic) step is to assign jobs to nodes under the considerations of the functional and non-functional (i.e. performance and dependability) constraints given in the PIMs. Examples for such constraints are:

- Resource requirements (e.g. memory, CPU, sensors, actuators, bandwidth).
- Dependability constraints (e.g. replicas must be assigned to different nodes).
A dual-track approach is taken in DECOS to generate the PSM. First, a transformation based mapping process has been developed which deals constraints one-by-one. It finds a feasible solution for resource allocation while satisfying different constraints. A heuristics based systematic resource allocation approach has been explicated for this and presented in [ILS06]. Considering dependability and real-time as prime drivers, we presented a schedulable allocation algorithm for the consolidated mapping of safety-critical and non safety-critical applications onto a distributed platform. Although the allocation problem is NP-hard [FB89], exploiting symmetry (job replicas, identical nodes, etc.) can improve the performance of such approaches [GW05], but it is difficult to assess solutions with respect to certain criteria like reliability maximization or cost minimization. Therefore, in a second phase, a Multi-Variable Optimization (MVO) approach is implemented where multiple objectives are optimized together with satisfaction of constraints. Here, a so-called MVO function is used, which associates a scalar-valued function $v(q)$ to each point $q$ in an evaluation space, representing the system designer’s preferences, provided that choosing a feasible alternative from a set of contenders such that $v$ is maximized or minimized.

5.2.4 Conclusions
The DECOS tool chain successfully adopted the model-driven development principles in the embedded systems domain and defined a novel approach for interactive, iterative hardware-software integration. This approach will be discussed in detail throughout this chapter.

5.3 Formal metamodels for MDD
In the current section formal, simplified platform-independent architecture model (SAM) and a simplified platform model (SPM) for distributed, safety-critical, real-time embedded systems will be introduced. Our goal with these languages is to provide a cross-domain formal framework for the discussion of hardware-software integration. The languages contain the common, most important concepts of the various domain-specific standard languages introduced earlier, but do not contain the specifics of the application domains (like automotive, aerospace, or industrial control). The languages can be treated as a common ancestor of the domain-specific languages that inherit and specialize the core concepts (see Figure 5.3).

5.3.1 Simplified Architecture Model
In order to keep the following descriptions compact, a simplified component model will be introduced. This contains the essential elements of the languages introduced earlier but allows for a more compact represen-
tation by cutting low-level details. The definition of the metamodel uses the VPM approach introduced in [Var04] and [PV05]. Besides the mathematically precise description, the metamodel is also illustrated by a diagram using the MOF/UML concrete syntax (Figure 5.4).

The atomic element of the model is the component. A component is a black box element, that has well defined interfaces to exchange data element values. Data elements have (logical) data types, which are treated as simple labellings.

**Definition 17 (Data type)** Data types are entities. Data types can refine other data types. It should be noted that the VPM supertypeOf relation can be used in order to express the common object-oriented inheritance relation between data types.

**Definition 18 (Component interface)** Component interfaces are entities that have relations to the data types of the contained data elements.
Compatibility of component interfaces is a fundamental problem that has several aspects. In [UBHB01] a four level model is presented:

1. **Syntactic level** Description of the message signatures (data elements).
2. **Semantic level** Description of the semantics of the component interfaces, including pre- and post conditions.
3. **Synchronization level** Description of the sequence of messages, loops, and alternative scenarios.
4. **QoS level** Specifies the non-functional characteristics of the components and messages.

On the current PIM (or architecture model) level, only the syntactic compatibility will be checked. QoS level can be analyzed after introducing such attributes. Semantics of the components is not described by the current approach, but pre- and post condition constraints on the data elements can be added as non-functional constraints. Synchronization properties can be analyzed using the platform-specific information that will be introduced by the hardware-software integration process.

We will introduce the syntactic compatibility of interface pairs in order to be able to validate the definition of communication links. Interface compatibility is based on the inspection of the data elements contained by the interfaces. Data elements are paired based on their names. More formally:

**Definition 19 (Interface compatibility)** A sender interface \( S \) and a receiver interface \( R \) are compatible if there is a bijection (based on name equality) between their data element, and all data element pairs are of compatible data types:

\[
\text{pattern } \text{compatibleInterfaces}(S,R,DS,DR) = \{
\begin{array}{l}
\text{Interface}(S); \\
\text{Interface}(R); \\
\text{DataElement}(DS); \\
\text{DataElement}(DR); \\
\text{containedElements}(C1,S,DS); \\
\text{containedElements}(C2,R,DR); \\
\text{String}(\text{NameS}); \\
\text{String}(\text{NameR}); \\
\text{name}(N1,DS,\text{NameS}); \\
\text{name}(N2,DR,\text{NameR}); \\
\text{DataType}(DTS); \\
\text{DataType(DTR));} \\
\text{type}(TS,DS,DTS); \\
\text{type}(TR,DR,DTR); \\
\text{check}(\text{NameS}==\text{NameR}) \#\# \text{compatibleDataType}(DS,DR);
\end{array}
\}
\]

and:

\[
\forall DS, \exists DR \text{compatibleInterfaces}(S,R,DS,DR) = true \land \\
\forall DR, \exists DS \text{compatibleInterfaces}(S,R,DS,DR) = true
\]

The compatibility of data types is also a complex relation. We distinguish between source (sender-side) and target (receiver-side) data types for this definition. A source data type is compatible with a target, if:

1. If the two data types are identical.
2. If the source data type is a subtype of the target data type.
The formal definition of the compatibility is the following:

**Definition 20 (Data type compatibility)** A source data type $DS$ is compatible with a target data type $DT$, if:

\[
\text{pattern } \text{compatibleDataType}(DS, DT) = \{
\text{DataType}(DS);
DS=DT; // DS and DT are the same datatype
\text{DataType}(DT);
\}
\]

\[\text{or } \{
\text{DataType}(DS);
\text{DataType}(D);
\text{supertypeOf}(D,DS);
\text{DataType}(DT);
\text{find } \text{compatibleDataType}(D, DT);
\}\]

**Proposition 14** The data type compatibility is reflexive and antisymmetric.

**Proof** Reflexitivity is a direct consequence of the definition of the compatibility, because $\text{compatibleDataType}(D, D)$ is true for all datatype $D$, as the first part of the pattern is satisfied.

The compatibility is antisymmetric, as

$$\forall D_1, D_2, \text{compatibleDataType}(D_1, D_2) \land \text{compatibleDataType}(D_2, D_1) \Rightarrow D_1 = D_2$$

If there are two data types $D_1 \neq D_2$, then according to the definition of compatibility, $\text{compatibleDataType}(D_1, D_2)$ is true if $D_2$ is direct or indirect supertype of $D_1$, similarly, $\text{compatibleDataType}(D_2, D_1)$ is true if $D_1$ is direct or indirect supertype of $D_2$. That requires that $D_1$ and $D_2$ are mutually refinements of each other (in VPM terms) that is contradictory to the definition of the VPM refinement relation. □

It should be noted, that even the syntactical correctness check can be extended, if the modeling notation supports the definition of domains for the data types (as in case of AutoSAR or DECOS PIM) or physical units [SHS06].

**Definition 21 (Component)** A component $C$ is an entity having relations to the set of its input and output interfaces.

\[
\text{entity}(Component) \land \text{relation}(input, Component, Interface) \land
\text{relation}(output, Component, Interface)
\]

Several generic platform services are defined in case of most embedded systems development frameworks. These can include for example clock synchronization, messages, membership, and diagnostics. The interface of these services is defined in a platform independent way, and the implementations map these descriptions to the actual execution platform. The service requirement of atomic components only refers to the set of required services, without knowledge on the actual implementation.

**Definition 22** Atomic component is a component that has relation to the set of required services. A service is an entity.

\[
\text{entity}(AtomicComponent);
\text{supertypeOf}(Component, AtomicComponent);
\text{entity}(Service);
\text{relation}(requiredServices, AtomicComponent, Service);
\]

Components can be composed in order to describe interactions between them. A composition is a component with internal structure: it contains component instances and connections. Connections represent communication relationships between compatible interfaces. Assembly connections interconnect interfaces of the internal component instances, while delegate connections interconnect an external interface of the composition with an internal interface of a contained component.

**Definition 23 (Composite component)** A composite component $C$ is a component that has relations to the set of contained component instances and connectors.
5.3. FORMAL METAMODELS FOR MDD

Definition 24 (Component instance) A component instance is an entity, and has a type relation to its component type.

The composition of components should be constrained in order to avoid the recursive containment of composite components that would lead to infinite-depth composite structures.

Constraint 1 (Component composition) A composite component must not contain (directly or indirectly) a component instance of its own.

We can define the composition depth function that calculates the depth of the composition hierarchy starting at a given composite component.

Definition 25 (Composition depth) Composition depth is a function: \( \text{compDept}(\text{CC}) \rightarrow \mathbb{Z}_{\geq 0} \) that denotes the depth of the component composition hierarchy below a component. Let \( \text{Comps}(\text{CC}) \) denote the set of composite component instances in \( \text{CC} \).

\[
\text{compDept}(\text{CC}) = \begin{cases} 
0 & \text{if } \text{Comps}(\text{CC}) = \emptyset \\
1 + \max_{C \in \text{Comps}(\text{CC})} (\text{compDept}(C).type) & \text{else}
\end{cases}
\]

Proposition 15 (Composition depth is finite) If the model is finite (contains finite number of elements), and fulfills Constraint 1, the \( \text{compDept}(\text{CC}) \) function returns a finite integer.

Proof Constraint 1 implies that a composition cannot contain instances of itself. If we look at the function \( \text{compDept}(\text{CC}_1) \) for an arbitrary \( \text{CC}_1 \in \text{Compositions} \) composite component of the model, \( \text{CC}_1 \) can contain composite instances from the set \( \text{Compositions} \setminus \text{CC}_1 \). If \( \text{CC}_1 \) contains an instance of composite component \( \text{CC}_2 \), then \( \text{CC}_2 \) can only contain instances of \( \text{Compositions} \setminus \text{CC}_1, \text{CC}_2 \) and so on. It is obvious that, even in worst case, after \( n = |\text{Compositions}| \) levels, \( \text{CC}_n \) can contain only atomic component instances. That means, that \( \forall \text{CC} \in \text{Compositions} : 0 \leq \text{compDept}(\text{CC}) \leq n \).

Definition 26 (Connectors) A connector is an entity, and either an assembly, or a delegate connector. Connectors have references to the source and target interfaces and components instance they are connected to.

The definition of the connectors is generic, but there are several constraints that apply on the various subtypes of the connectors.

Definition 27 (Assembly connector) An assembly connector always connects the interfaces of two component instances that are contained by the same composition. The direction of the interfaces (input, output) should correspond to the connector references (target, and source, respectively). The formal specification of the validity constraint can be found in Appendix B.1.1.
Definition 28 (Delegate connector) A delegate connector either connects an input interface of the containing composition with an input interface of one of the component instances in the composition, or it connects an output interface of a component instance to an output port of the composition. The formal specification of the validity constraint can be found in Appendix B.1.2.

An application contains the components that are present in it, all the data type labeling that are used by the component interfaces, and a dedicated top level composition that defines the structure of the application.

Definition 29 (Application) An Application App is an entity that has references to the defined data types, component, and the declared services. It also has a reference to the main (top-level) composition of the application.

```plaintext
entity (Application);
relation (dataTypes , Application , DataType);
relation (declaredService , Application , Service);
relation (components , Application , Component);
relation (main, Application , CompositeComponent);
```

It should be noted that this representation focuses only on the architecture and composition of the application, and does not contain any information about the behavior of the components.

For the integration, however, several non-functional aspects have to be considered. In the followings, the most important timeliness and dependability related parameters will be defined. All these parameters are added to the model using the technique introduced in Section 4.5.6.

Timing properties

In order to be able to represent timing properties in the model, the respective physical units and NFP_Type should be defined.

Definition 30 (Timing units) The VPM definition (based on the NFP framework introduced in Section 4.5.6) of the timing related units and NFP_Type is as follows:

```plaintext
Unit (sec);
String (secsym) -> "s";
symbol (symbol , sec , secsym);

Unit (msec);
String (msecsym) -> "ms";
symbol (symbol , msec , msecsym);
Double (msecconv) -> "0.001";
basedUnit (base , msec , sec);
convFactor (factor , msec , msecconv);

Unit (usec);
String (usecsym) -> "us";
symbol (symbol , usec , usecsym);
Double (usecconv) -> "0.001";
basedUnit (base , usec , msec);
convFactor (factor , usec , usecconv);

Unit (nsec);
String (nsecsym) -> "ns";
symbol (symbol , nsec , nsecsym);
Double (nsecconv) -> "0.001";
basedUnit (base , nsec , usec);
convFactor (factor , nsec , nsecconv);

NFP_Type (DurationType);
allowedUnits (au1 , DurationType , sec);
allowedUnits (au2 , DurationType , msec);
allowedUnits (au3 , DurationType , usec);
allowedUnits (au4 , DurationType , nsec);
defaultUnit (du1 , DurationType , sec);
```
Worst-case execution time (WCET) is the longest execution time of a component. A component execution is started by the execution environment, includes the consumption of input data elements and is terminated after preparing new data elements on the outputs of the component.

**Definition 31 (Worst-case execution time)** Worst-case execution time (WCET) is a NFP that is bound to an atomic component, and its type is DurationType. WCET denotes the maximum execution time of the component.

The components can be executed in a time-triggered or an event-triggered mode. Time-triggered (TT) components are synchronized to the progress of (node local or global) time, while event-triggered (ET) components are synchronized to the occurrence of an event (message reception in our case). Several properties are associated to both types of component execution modes.

**Definition 32 (Component execution period)** Component Period (Period) is a NFP that is bound to a component and denotes the time between consecutive executions of the component with time-triggered scheduling.

**Definition 33 (Debounce time)** Debounce time (DT) is an NFP that is bound to components and denotes the minimum time that must be between two consecutive execution of the component.
Definition 34 (Priority)  Priority is a NFP of IntegerType that defines an ordering between (ET) components. If multiple components are triggered for execution, the one with the highest priority will be executed first.

\[
\text{NFP(Priority);} \\
\text{type(t, Priority, IntegerType);} \\
\text{relation(value, Priority, ValueSpecification);} \\
\text{relation(priority, Component, Priority);} \\
\]

Similarly to components, interfaces can also have timing properties. These properties control the transmission of data element values between sender and receiver components. Timing properties of senders are definitions of the intended behavior, while on the receiver side they are requirements that the sender has to fulfill. Data element transmissions can occur either using time-triggered, event-triggered, or streaming modes.

- Time-triggered transmission follows a time-synchronized scheme that is defined during design time. It allows for a predictable temporal behaviour.
- Event-triggered transmission is triggered by the update of source data element values. It allows for a fast reaction to events but has a more complex temporal behavior.
- Streaming transmission is a guaranteed-bandwidth service used mainly for multimedia content where there is a constant data-flow that tolerates sporadic jitters.

Definition 35 (Message debounce time)  Message Debounce Time (DebounceTime) of an event-triggered data flow is the minimum interval between two consecutive transmissions of the data on the communication system.

Formally, DebounceTime is an instance of NFP, and is attached to an interface of a component.

\[
\text{NFP(DebounceTime);} \\
\text{type(t, DebounceTime, DurationType);} \\
\text{relation(value, DebounceTime, ValueSpecification);} \\
\text{relation(debounceTime, Interface, DebounceTime);} \\
\]

Definition 36 (Message period)  Message Period (MsgPeriod) of a time-triggered data flow is the repetition interval of message transmissions.

\[
\text{NFP(MsgPeriod);} \\
\text{type(t, MsgPeriod, DurationType);} \\
\text{relation(value, MsgPeriod, ValueSpecification);} \\
\text{relation(period, Interface, MsgPeriod);} \\
\]

Definition 37 (Deadline)  Deadline (Deadline) defines the maximum lifetime of the data values of the port. The values should be transmitted from the sender buffer to all of the receivers within this time (including all network and gateway delays, and other internal processing in the communication middleware).

\[
\text{NFP(Delay);} \\
\text{type(t, Deadline, DurationType);} \\
\text{relation(value, Deadline, ValueSpecification);} \\
\text{relation(deadline, Interface, Deadline);} \\
\]

Definition 38 (Bandwidth)  Bandwidth is the average bandwidth requirement of a data stream.

\[
\text{NFP(Bandwidth);} \\
\text{type(t, Bandwidth, DataRateType);} \\
\text{relation(value, Bandwidth, ValueSpecification);} \\
\text{relation(bandwidth, Interface, Bandwidth);} \\
\]

Definition 39 (Jitter)  Jitter is the length of the maximum allowed break during the transmission of the data stream.

\[
\text{NFP(Jitter);} \\
\text{type(t, Jitter, DurationType);} \\
\text{relation(value, Jitter, ValueSpecification);} \\
\text{relation(jitter, Interface, Jitter);} \\
\]
Dependability-related properties

Dependability related properties represent high-level requirements on PIM level, and the fulfillment of properties should be verified later during the development process.

Similarly to the timing properties, the first step of the parameter definition is the definition of NFP Types.

Definition 40  
The VPM definition of the dependability related units and NFP Types is as follows:

\[
\text{NFP Type}(\text{IntegerType}); \\
\text{NFP Type}(\text{DoubleType});
\]

As all parameters are scalar values, only two basic NFP types are needed.

Definition 41 (Safety integrity level)  
Safety Integrity Level (SIL) defines the safety level requirement of an application according to (IEC61508 [RIARI00]). This information is used later for the configuration of the verification and validation plans, and the development process.

\[
\text{NFP}(\text{SIL}); \\
type(t,\text{SIL},\text{IntegerType});
\]

\[
\text{relation}(\text{value},\text{SIL},\text{ValueSpecification}); \\
\text{relation}(\text{safetyIntegrityLevel},\text{Application},\text{SIL});
\]

Definition 42 (Redundancy degree)  
Redundancy Degree (Red) defines a replication degree for a given element (application or component). The element should be executed using a redundancy pattern in order to achieve better dependability.

\[
\text{NFP}(\text{Red}); \\
type(t,\text{Red},\text{IntegerType});
\]

\[
\text{relation}(\text{value},\text{Red},\text{ValueSpecification}); \\
\text{relation}(\text{redundancyDegree},\text{Application},\text{Red});
\]

\[
\text{relation}(\text{redundancyDegree},\text{Component},\text{Red});
\]

Definition 43 (Availability)  
Availability (Aval) defines an application level requirement for the availability of the services offered by the application.

\[
\text{NFP}(\text{Aval}); \\
type(t,\text{Aval},\text{DoubleType});
\]

\[
\text{relation}(\text{value},\text{Aval},\text{ValueSpecification}); \\
\text{relation}(\text{availability},\text{Application},\text{Aval});
\]

It should be noted, that the NFP framework is open in order to be able to define new non-functional properties for analysis or synthesis purposes upon needs.

5.3.2 Execution Platform Modeling

The execution platform model describes the main characteristics of the hardware and basic software (operating systems, middleware) of the target execution platform. The hardware model has a course granularity focusing only on the most important aspects of the system (components, connections, dependencies). The basic software description contains information that is necessary for the allocation and scheduling of application components (implemented services, size of code, etc.).

The structure of the hardware model (see Figure 5.5) is built following the component orientation principle. (It should be noted that component orientation is a traditional design method in hardware design). The components represent various hardware elements like devices, PCB cards, micro controllers, etc. The components can, of course, form a hierarchy with the definition of composite components.

Definition 44 (Platform component)  
A Component \( C \) is an entity that has relations to its implemented platform services, required and provided feature, and connectors.

A platform service is an identifier. Connector is also an identifier that represents interaction interfaces of the components.
CHAPTER 5. MDSD WITH QUALITY ASPECTS

Figure 5.5: UML Representation of the Platform Model

entity(Component);
relation(implementedServices, Component, PlatformService);
relation(requiredFeatures, Component, FeatureInstance);
relation(providedFeatures, Component, FeatureInstance);
relation(connectors, Component, Connector);

entity(PlatformService);
entity(Connector);

Definition 45 (Component instance) A Component instance is an entity, referring to its type, and connector instances.

entity(ComponentInstance);
relation(type, ComponentInstance, Component);
relation(connectorInstances, ComponentInstance, ConnectorInstance);

A connector instance is an entity having reference to its type connector.

entity(ConnectorInstance);
relation(type, ConnectorInstance, Connector);

Definition 46 (Composite Hardware Component) A Composite Hardware Component is a Component, and contains component instances and connections that interconnect connector instances

entity(CompositeComponent);
supertypeOf(Component, CompositeComponent);
relation(components, CompositeComponent, ComponentInstance);
relation(connection, CompositeComponent, Connection);

A connection represents a physical connection between a set of connector instances.

entity(Connection);
relation(members, Connection, ConnectorInstance);

Delegate connection is a specialization of the generic connector that interconnects an external connector of the composite component with several internal connector instances.
5.3. FORMAL METAMODELS FOR MDD

Similarly to the SAM model, we have to restrict the structure of the compositions to avoid infinite recursive containments.

**Constraint 2 (Component composition)** A composite hardware component must not contain (directly or indirectly) a component instance of its own.

We can define the hardware composition depth function that calculates the depth of the composition hierarchy starting at a given composite component.

**Definition 47 (Composition depth)** Composition depth is a function:

\[ hwCompDept(C) : \text{CompositeComponent} \to \mathbb{Z} \geq 0 \]

that denotes the depth of the component composition hierarchy below a component. Let Comps(CC) denote the set of composite component instances in CC.

\[
hwCompDept(CC) = \begin{cases}
0 & \text{if Comps(CC) = } \emptyset \\
1 + \max_{C : \text{Comps(CC)}} hwCompDept(C.type) & \text{else}
\end{cases}
\]

**Proposition 16 (Hardware composition depth is finite)** If the model is finite (contains finite number of elements), and fulfills Constraint 2, the \(hwCompDept(CC)\) function returns a finite integer.

**Proof** The proof is similar to the proof of Proposition 15.

Constraint 2 implies that a composition cannot contain instances of itself. If we look at the function \(hwCompDept(CC)\) for an arbitrary \(CC\) composite component of the model, \(CC\) can contain composite instances from the set Compositions \(\setminus CC\). If \(CC\) contains an instance of composite component \(CC\), then \(CC\) can only contain instances of Compositions \(\setminus CC\), \(CC\), and so on. It is obvious that, even in worst case, after \(n = |\text{Compositions}|\) levels, \(CC\) can contain only atomic component instances. That means, that \(\forall CC : 0 \leq hwCompDept(CC) \leq n\).

Features are introduced to support the uniform representation of various hardware services of the components like power distribution, clock generation, etc. Feature providers are producers of resources that are consumed by the feature requesters through feature connections.

Feature definitions define the set of possible parameters that can be represented. Feature definitions can be added during modeling in order to support the introduction of new analysis or synthesis aspects. Model analyzers may support a subset of these parameters.

Features can be characterized using non-functional properties. For this, the same NFP framework that has already been introduced in Section 4.5.6 will be used.

**Definition 48 (Feature)** A Feature represents a hardware level measurable service item. For instance, for power distribution, voltage and current can be defined as two separate features. Feature is an entity, and has relation to non-functional properties that characterize the feature.

\[
\text{entity(Feature);} \\
\text{relation(characteristics,Feature,NFP);}
\]

Actual feature values are described by feature instances. These elements refer to a given feature definition and contain a value specification for the specified feature.

**Definition 49 (Feature instance)** A Feature Instance is an entity that has relation to its feature type and the actual characteristic values of all NFPs that are defined by its type.

\[
\text{entity(FeatureInstance);} \\
\text{relation(feature,FeatureInstance,Feature);} \\
\text{relation(characteristicValues,FeatureConnection,NFP);}
\]

**Definition 50 (Feature Connection)** A Feature Connection is an entity that refers to the supplier and the consumer of the connection and to all feature instances that are characterizing the connection.
CHAPTER 5. MDSD WITH QUALITY ASPECTS

The plain hardware model defines the topology of components, but it is not containing any information about the functionality or role of the elements. In order to support allocation and configuration, several utility functions should be defined.

Definition 51 (Component role) ComponentRole is an entity and there are three different predefined roles.

- **ExecutionEnvironment** denotes an execution unit able to run software components.
- **CommController** denotes a communication controller that manages (part of) a protocol stack.
- **Transceiver** denotes the physical layer device that connects the hardware component to a communication network.

It should be noted, that the definition of ComponentRole does not allow the compositions to have roles. This way, compositions are only containers, and only atomic components can play distinguished role in the hardware model.

Definition 52 (Connector role) ConnectorRole is an entity, and has two predefined instances.

- **LinkingConnector** denotes a connector that serves as inter-component communication link that is able to transmit user data and needs configuration and/or scheduling to be done.
- **LocalConnector** denotes a connector that is used as inter-component communication resource and is handled by a middleware module.

A result of connector roles is that the concept of linking connections and local connections can be defined, and no mixed connections should be allowed.

Constraint 3 A connection should only connect linking connectors, or local connectors, but not mixed. The graph patterns formalizing this requirement can be found in Appendix B.2.

Part-specific properties

The component concept of the platform model represents generic hardware elements. Specific part types, however, have several part-specific properties or parameters that should be specified in order to be able to generate the configuration or to perform detailed analysis based on the model. In order to support the description of custom properties without the need to extend the metamodel if a new part type is introduced, a generic part-property description mechanism has been defined. Part types can be defined together with their properties. Components can refer to their part types, and contain property values that are instances of the properties of their respective part.

Definition 53 (Hardware part type) HWPart is an entity that contains properties. Components refer to their part by a relation.
Property values are represented by instances of the HWPropertyValue that refer to their HWProperty types.

entity (HWPropertyValue);
relation (property, HWPropertyValue, HWProperty);
relation (value, HWPropertyValue, ValueSpecification);
relation (properties, Component, HWPropertyValue);

Platform data types

The platform data type definition is an integral part of the platform model describing the set of supported data types for an execution platform. The set of data types contains several basic types (real, integer, etc.), and several composition types (array, record) that can be used to create new, composite data types. It should be noted, however, that even if the platform supports the creation of composite types, it can limit the possible compositions (dimension of arrays, etc.). These limitations are defined using platform-specific constraints.

Definition 54 (Primitive data types) A data type is an entity. Primitive type is a data type, that has a size (measured in bits). IntegerType, RealType, and BooleanType are primitive types.

entity (DataType);
entity (PrimitiveType);
relation (size, PrimitiveType, Integer);
supertypeOf (DataType, PrimitiveType);

entity (IntegerType);
entity (RealType);
entity (BooleanType);
supertypeOf (PrimitiveType, IntegerType);
supertypeOf (PrimitiveType, RealType);
supertypeOf (PrimitiveType, BooleanType);

entity (ArrayType);
relation (element, ArrayType, DataType);
relation (length, ArrayType, Integer);
supertypeOf (DataType, ArrayType);

entity (RecordType);
relation (element, RecordType, RecordElement);
relation (type, RecordElement, DataType);
supertypeOf (DataType, RecordType);

Definition 55 (Complex data types) ArrayType is a subclass of DataType and represents an array of uniform data elements. RecordType is also a data type, and represents a heterogeneous structure where the elements are identified by element names.

entity (ArrayType);
relation (element, ArrayType, DataType);
relation (length, ArrayType, Integer);
supertypeOf (DataType, ArrayType);

entity (RecordType);
relation (element, RecordType, RecordElement);
relation (type, RecordElement, DataType);
supertypeOf (DataType, RecordType);

Definition 56 (Data type size) Function size : DT \rightarrow R^+ defines the size of storage space required for the given data type. size can be calculated as follows:
for PrimitiveTypes size is the value of the size property.

for array type size(DT) = \text{size(DT}_{\text{element}}) \times \text{length}

for record type size = \sum_{\text{element}} \text{size(DT}_{\text{base}})

The implementation of the size function using VTCL can be found in Appendix B.3.

Platform specific constraints (expressed using a high level constraint language, like OCL) define the valid set of composite data types.

**Definition 57** ValidTypes is a set: \( \text{ValidTypes} = |C_{DT}| \), where \( C_{DT} \) is a function: \( C_{DT} : DT \rightarrow \text{Bool} \) that is true if the data type is valid for the specific platform, and false if not.

### 5.4 Iterative, Transformation-based Hardware-Software Integration

While the traditional MDA approach is widely used in the desktop and enterprise application domains, it did not gain too high importance in the embedded systems field. There are some academic and commercial tools [Graa,Tel,Aon] that offer support for model-driven development of embedded systems, but up to now, their impact on the industry is quite limited. These solutions are dedicated to a single platform, and are not able to support the configuration of complex middleware modules, like the communication subsystems.

In the following, we propose an improved approach for model-driven hardware-software integration for embedded systems that is able to support the configuration and scheduling of both software component execution and communication. As this domain is broad, we will focus on safety-critical, real-time embedded systems.

#### 5.4.1 Requirements for the integration process

The goal of the hardware-software integration in these systems is to map the software components to computing nodes, the sensors/actuators needed by them to hardware peripherals, and the messages to networks while fulfilling the functional and non-functional requirements of the designer. Functional requirements mean that the behavior of the resulting system implementation has to be equivalent with the models, while non-functional requirements may include timeliness, availability, power consumption, and other aspects.

Due to the extensive set of constraints and possibilities that characterize the development of embedded systems, the traditional one-step MDA approach cannot be successful. In some points of the integration process, there is need for design decisions that rely on more information than that is present in the formal models. To overcome this problem, we designed an interactive, iterative design process that involves the system designer to the most important decisions, and automates the mechanical steps. Figure 5.6 illustrates the main phases of a typical integration process.

![Figure 5.6: Typical hardware-software integration process](image)

#### 5.4.2 Inputs and outputs of the process

The mapping process has several inputs and outputs. The most important inputs are the models that describe the application(s) and the target platform. The mapping tool is specific to the metamodel of the input and languages, and the output notation. It must be noted that although the steps described in the following sections are general, and can be applied to models of various software and hardware description languages, the actual implementation of the concept depends on the current metamodel.
The input model instances are several software models (PIMs), each describing a software subsystem, and a PM that describes the hardware configuration of the target system. The output of the process is the PSM of the system that describes the integrated hardware-software system in a single model. This model can be the input of code generators (generating configuration files and middleware wrapper code), or analysis tools (for certification and validation).

**Definition 58 (Platform specific model)** The Platform Specific Model (PSM) is a tuple:

\[ PSM = (PIM, PM, IM, state) \]

where:
- \( PIM \) is the set of platform-independent models,
- \( PM \) is the platform model,
- \( IM \) is the integration model,
- and \( state : \text{Steps} \rightarrow \text{States} \) is the function returning the actual step states.

![Integration model](image)

The integration model is describing the information added during the PIM-PSM mapping process. The model has dedicated sub-models (or packages) that relate to specific steps of the mapping and each step manipulates its dedicated model part (see Figure 5.7). The model parts may refer to each other (a later step may use information generated during earlier steps). The model packages related to a step are also called step models.

The possible states of the steps can be: \( \text{States} = \text{empty}, \text{inProgress}, \text{finished} \). Empty means that the step is not started, inProgress means that it has already started, and finished marks the completed steps. The state transitions are specific for the different kinds of steps. The most important step types will be described in the following sections.

### 5.4.3 Model import

Model import steps are used to add a new model to the PSM. In case of PIMs, this means: \( \text{import}(PIM_{\text{new}}) :: PIM' := PIM \cup PIM_{\text{new}} \). In case of platform models, the new model will replace the other one: \( \text{import}(PM_{\text{new}}) :: PM' := PM_{\text{new}} \).

The import steps are in empty state, if no model has been imported yet. If at least one model is present, the step state changes to complete. That means that the import steps cannot be in in progress state, as the number of desired input models (in case of PIM import) is unknown for the PSM.
5.4.4 Model marking

Model marking steps are used to add new information to the model on a given abstraction level. This can be treated as a user hint to guide the further (automatic) steps of the development process by providing additional input to the models.

Definition 59 A model marking step is a function: \( \text{marking} : \text{Base} \rightarrow \text{Mark} \), where:
- \( \text{Base} \) is the set of model elements that can be marked by this step, and
- \( \text{Mark} \) is the set of possible markings.

The marking step definition is a tuple \( \text{Step} = (\text{Base}, \text{Mark}, \text{marking}) \).

5.4.5 Mapping

Mapping steps define relations between model elements on different abstraction levels. This results in a step toward the realization of the high-level PIM on a given platform. A typical example is the mapping of logical data types to platform-level ones. Mapping steps are manual steps.

Definition 60 A mapping step is a function: \( \text{mapping} : \text{Abstr} \rightarrow \text{Conc} \), where:
- \( \text{Abstr} \) is the set of abstract model elements to be mapped by this step, and
- \( \text{Conc} \) is the set of concrete model elements that can be used for the mapping.

The mapping step definition is a tuple \( \text{Step} = (\text{Abstr}, \text{Conc}, \text{mapping}) \).

5.4.6 Allocation

Allocation steps are special mapping steps that are used to perform the allocation of resource consumer elements (software components, messages) on execution resources (hardware nodes, network segments). This can be performed either manually, using automatic (customizable) heuristic algorithms, or by explicit mathematical optimization techniques.

Definition 61 An allocation step is a function: \( \text{allocation} : \text{Comp} \rightarrow \text{Exec} \), where:
- \( \text{Comp} \) is the set of resource consumer components, and
- \( \text{Exec} \) is the set of execution elements.

The allocation step definition is a tuple \( \text{Step} = (\text{Abstr}, \text{Conc}, \text{allocation}) \).

5.4.7 Transformation Steps

Model transformation steps are automatic, hidden steps that execute a model-to-model transformation on the actual model space in order to perform work-intensive operations that do not require human decisions. Model transformation steps are triggered by the state change of another PIM-PSM step or manually. If the modeling and model manipulation framework support incremental transformations, these steps can also be implemented as live transformations.

5.4.8 Code generation

Code generation steps are automatic model-to-text transformation steps creating output files for the configuration of the target system according to the PSM, and the source code of the application. These steps are triggered by the user, and the generated output is out of the control of the PIM-PSM mapping framework. Both the model-to-model and model-to-text transformation steps rely on a model transformation engine. As these solutions follow different principles and have different formalisms for the definition of the transformations, the step definitions are always specific to the selected execution engine.

5.5 PIM-PSM mapping

A PIM-PSM mapping based on the SAM and SPM models will be defined in this Section. We will assume that the target system is following the architectural principles defined by the Genesys Project [Conb]. The workflow of the mapping is illustrated by Figure 5.8. The PSM will be introduced step-by-step, together with the mapping steps.
5.5. PIM-PSM MAPPING

5.5.1 Model import

The model import step includes the PIM and PM import. These functionalities are used to add the source models to the PSM model space. The behaviour of these steps is described in Section 5.4.3.

5.5.2 PIM flattening

The platform independent architecture model introduced in Section 5.3.1 is hierarchical, as it can contain composite software components that have sub-components on multiple levels. In order to be able to allocate and schedule the components on the target, the model should be flattened.

Flattening is an automatic transformation step (see Sec. 5.4.7) that performs a simple model transformation:

**Algorithm 2 PIM flattening**

1. Find the top-most composition in the model ($C_{top}$).
2. If $C_{top}$ contains only atomic component instances, stop.
3. Find a complex component instance $CC$ below $C_{top}$.
4. Copy all component instances from $CC$ to $C_{top}$.
5. Copy all assembly connectors from $CC$ to $C_{top}$.
6. For each $A$ assembly connectors in $C_{top}$, that are leading to input ports of $CC$, and for all $D$ delegate connectors in $CC$, create assembly connectors from the source of $A$ to the target of $D$. That way, the input delegations will be replaced by assembly connectors in $C_{top}$.
7. For each $A$ assembly connectors in $C_{top}$, that are leading from output ports of $CC$, and for all $D$ delegate connectors in $CC$, create assembly connectors from the source of $D$ to the target of $A$. That way, the output delegations will be replaced by assembly connectors in $C_{top}$.
8. Delete $CC$ from $C_{top}$.
9. Go to step 2.

The detailed algorithm (represented in VTCL) can be found in Appendix B.4.

**Proposition 17** The above algorithm always terminates if the input model is a valid component model.

**Proof** The proof of the proposition is straightforward: step 2 terminates the execution, if $C_{top}$ contains only atomic component instances. In each iteration of the algorithm, a composite component instance $CC$ is deleted from $C_{top}$ and its content is copied to the top composition. The consequence of Proposition 15 is that the depth of the composition hierarchy below $C_{top}$ is less or equal to the number of composite components in the model ($n = |Compositions|$). As the model contains finite number of elements, there is a number $m$ that is the maximum number of composite component instances in a single composition.

Given $m$ and $n$, the number of composite composition instances below $C_{top}$ is less or equal to: $n \times m$. As in each iteration exactly one of these instances is deleted, the number of iterations is also less than or equal to $n \times m$. The algorithm terminates in finite steps. \[\Box\]
Proposition 18 The result of the above algorithm is a flat PIM, where the top level composition contains only atomic software component instances.

Proof The proof is the trivial consequence of the termination condition of the algorithm (step 2), and that the algorithm always terminates (Proposition 17).

5.5.3 Component instance replication

In case of fault tolerant systems, replication is one of the fundamental strategies to achieve better dependability. In our case, replication is done on component level as an automatic transformation step (see Sec. 5.4.7).

The transformation replicates all component instances that’s application or component type defines a replication degree that is higher than one. The component instance and its inbound connection links are copied. In case of outbound connections, the connections are also replicated, but for all receivers, a voter component is needed (see Figure 5.9). The voter implements an arbitrary voting algorithm on the messages coming from all the replicas.

![Component instance replication](image)

In the current architecture, the voter will be part of the receiver component, and the voting will be executed before the execution of the main component behavior. This assumption (also taken by other platforms, such as DECOS) simplifies the allocation and scheduling of the system.

The complete replication transformation can be found in Appendix B.5.

5.5.4 PM flattening

The platform model introduced in Section 5.3.2 is hierarchical, as it can contain composite platform components that have sub-components on multiple levels. In order to be able to perform allocation, configuration, and scheduling of the system, the model should be flattened.

Flattening is an automatic transformation step (see Sec. 5.4.7) that performs a simple model transformation that is similar to the PIM flattening transformation. The formal VTCL specification of the transformation can be found in Appendix B.6.

Algorithm 3 PM flattening

1. Find the top-most composite component in the model ($C_{top}$).

2. If $C_{top}$ contains only atomic component instances, stop.

3. Find a CC complex component instance below $C_{top}$.
4. Find the CType type of CC.
5. Copy all component instances and their connector instances from CType to C\textsubscript{top}.
6. Copy all non-delegation connections from CType to C\textsubscript{top}.
7. For all connections that contain connector instances of CC among their members, add all connector instances to the connection that’s original is connected to the delegation connector in CType.
8. Delete CC and its connector instances.
9. Go to step 2.

The algorithm is analogous to the PIM flattening, and has the same properties:

**Proposition 19** The above algorithm always terminates if the input model is a valid component model.

**Proof** The proof of the proposition is straightforward: step 2 terminates the execution, if \( C\textsubscript{top} \) contains only atomic component instances. In each iteration of the algorithm, a composite component instance CC is deleted from \( C\textsubscript{top} \) and its content is copied to the top composition. The consequence of Proposition 16 is that the depth of the composition hierarchy below \( C\textsubscript{top} \) is less or equal to the number of composite components in the model (\( n = |Compositions| \)). As the model contains finite number of elements, there is a number \( m \) that is the maximum number of composite component instances in a single composition.

Given \( m \) and \( n \), the number of composite composition instances below \( C\textsubscript{top} \) is less or equal to: \( n \times m \). As in each iteration exactly one of these instances is deleted, the number of iterations is also less than or equal to \( n \times m \). The algorithm terminates in finite steps. \( \square \)

**Proposition 20** The result of the above algorithm is a flat PM, where the top level composition contains only atomic component instances.

**Proof** The proof is the trivial consequence of the termination condition of the algorithm (step 2), and that the algorithm always terminates (Proposition 19). \( \square \)

It should be noted that both in case of PIM and PM, traceability between the flat and hierarchical models should be established in the tool implementation in order to be able to navigate between the generated and original models.

### 5.5.5 Data type mapping

Data type mapping is a manual mapping step (see Section 5.4.5) that establishes a mapping between PIM level logical data types and platform data types. The mapping data is stored using the following PSM metamodel part:

```plaintext
type entity (DataTypeMapping);
relation (pimDataType , DataTypeMapping , pim . DataType);
relation (platformDataType , DataTypeMapping , dt . DataType);
```

An instance of the mapping is a `DataTypeMapping` entity. It refers the mapped PIM and platform data types. The creation of the mapping is a simple graph transformation rule (see Appendix B.7), while the deletion of the mapping is done by the deletion of the mapping entity. The mapping is complete, if all of the PIM data types are mapped. Formally:

**Constraint 4** The data mapping is complete, if:

\[ \exists DT : \text{unmappedDataType}(DT), \]

where \( \text{unmappedDataType} \) is a graph pattern:

```plaintext
pattern unmappedDataType(DT) = {
    pim . DataType(DT);
    neg pattern negp(DT) = {
        pim . DataType(DT);
        DataTypeMapping (ExistingMap);
        DataTypeMapping . pimDataType(X, ExistingMap, DT);
    }
}
```
5.5.6 Job compatibility marking

Job compatibility marking is a manual marking step (see Section 5.4.4) that is used to establish a set of compatible execution units for application components. Compatibility consists of two aspects. One aspect is the compatibility of the application component with an execution component regarding formal criteria present in the models, like service requirements. The other compatibility aspect involves informal requirements that are not (or cannot be) represented in the PIM or PM models like physical arrangement constraints. The first type of the constraints is checked by the mapping framework, but the second type should be established, maintained, and verified by the developer.

The mapping data is stored using the following PSM metamodel part:

```plaintext
entity (JobCompatibilityMapping);
relation (component, JobCompatibilityMapping, pim.ComponentInstance);
relation (executionElement, JobCompatibilityMapping, platform.ComponentInstance);
```

An instance of the mapping is a `JobCompatibilityMapping` entity. It refers to the mapped PIM component instance and a platform execution element. The creation of the mapping is a simple graph transformation rule (see Appendix B.8), while the deletion of the mapping is done by the deletion of the mapping entity.

The mapping is complete, if all of the component instances from the main composition of the application are mapped. Formally:

**Constraint 5** The job compatibility mapping is complete, if:

\[ \forall CI : \neg \text{unmappedAppLevelCompInstance}(App, CI), \]

where `unmappedAppLevelCompInstance` is a pattern:

```plaintext
pattern unmappedAppLevelCompInstance (App : pim.Application, CI : pim.ComponentInstance) = {
  pim.Application (App);
  pim.Application.main (M, App, Comp);
  pim.CompositeComponent (Comp);
  pim.CompositeComponent.components (C, Comp, CI);
  pim.ComponentInstance (CI);

  neg pattern negp (CI) = {
    pim.ComponentInstance (CI);
    JobCompatibilityMapping (ExistingMap);
    JobCompatibilityMapping.component (X, ExistingMap, CI);
  }
}
```

The compatibility mapping is a complete allocation, all component instances have exactly one compatibility mapping. Formally:

**Definition 62 (Complete allocation)** A compatibility mapping defines a complete allocation, if

\[ \forall Application(App), \forall ComponentInstance(CI) : \neg \text{unmappedAppLevelCompInstance}(App, CI) \land \text{hasSingleMapping}(CI) \]

where `hasSingleMapping` is a pattern:

```plaintext
pattern hasSingleMapping (CI : pim.ComponentInstance) = {
  pim.ComponentInstance (CI);

  neg pattern negp (CI) = {
    pim.ComponentInstance (CI);
    JobCompatibilityMapping (ExistingMap);
    JobCompatibilityMapping.component (I, ExistingMap, CI);
    JobCompatibilityMapping.component (I2, ExistingMap2, CI);
  }
}
```

In this case, the allocation step only verifies the pre-allocated system. Manual allocation of jobs to execution elements is required by some of the target domains (eg. automotive) where the industrial practice does not allow the use of automatic allocation methods.
5.5.7 Job allocation

Job allocation (based on the former compatibility marking step) is an automatic allocation step (see Section 5.4.6). It allocates jobs (application component instances) to execution platform components. Formally, the job allocation is a function: \( \text{JobAlloc} : \text{Jobs} \rightarrow \text{Comps} \), where \( \text{Jobs} \) is the set of application components (or jobs), and \( \text{Comps} \) is the set of platform components. The allocation is described by the following VPM meta model part:

\[
\begin{align*}
\text{entity (JobAllocation);} \\
\text{relation (component, JobAllocation, pim.ComponentInstance);} \\
\text{relation (executionElement, JobAllocation, platform.ComponentInstance);}
\end{align*}
\]

Several different constraints have to be fulfilled by the resulting allocation:

1. **Compatibility.** All jobs should be allocated to an execution component compatible with the job.
   \( \forall J \in \text{Jobs} : \text{JobAlloc}(J) \subseteq \text{JobComp}(J) \)

2. **Completeness.** All jobs should have an associated execution component.
   \( \forall J \in \text{Jobs} \exists C \in \text{Comps} : \text{JobAlloc}(J) = C \)

3. **Workload.** The workload caused by the components and the middleware should not exceed the capacity of the execution element.
   (a) In case of processors, workload is mainly characterized by the execution time the component consumes. This should extended with the memory usage (code and data) of the component.
   (b) In case of PGAs/ASICs, workload translates to space consumption which is either defined in \( \text{mm}^2 \) or number of logical elements (flip-flops, gates) depending on the type of the hardware.

4. **Dependability.** In case of replicated components, each replica should be allocated to different execution elements in order to avoid the effect of common mode faults.

The set of constraints can be freely extended, but it should be noted that in order to be able to allocate large systems in an affordable time frame, the constraint evaluation should not be computationally complex. In several cases the on-line constraint evaluation is complemented by a post-allocation verification that is able to evaluate more complex constraints, or even to perform in-depth analysis of the allocated system model using formal methods.

For the current case study, the heuristic heuristic method in [ILS06] will be adopted to the current architecture, and to the graph transformation based approach. An optimization based method will be introduced later in Section 6.4.

The heuristic approach separates the jobs to be grouped into several partitions that will be allocated in different phases. In our example, we will use the following partitioning of jobs:

1. **Replicated safety-critical jobs.** This partition contains each job that belongs to an application with defined (non-zero) SIL level, and have greater than one redundancy degree.
2. **Non Replicated safety-critical jobs.** This partition contains each job that belongs to an application with defined (non-zero) SIL level, and have redundancy degree equal to one.
3. **Jobs from non safety-critical applications.**

The outline of the algorithm:

**Algorithm 4 (Job allocation)** The allocation uses the set of nodes (\( \text{Nodes} \)) and the set of jobs (\( \text{Jobs} \)) as inputs.

1. Enumerate the set of execution environment components into the set \( \text{Nodes} \).
2. \( \text{Phase} = 1 \)
3. Enumerate the set of jobs that will be allocated in the current \( \text{Phase} \) to the \( \text{Jobs} \) set.
4. Order \( \text{Nodes} \) and \( \text{Jobs} \) according to the heuristics that applies to this phase.
5. Select the first \( J \). If \( \text{Jobs} \) is empty, then done.
6. Select the first Node \( N \) that has not been evaluated already as a possible allocation for \( J \).
7. Evaluate all criterion whether \( J \) can be assigned to \( N \). If assignment is possible, then assign \( J \) to \( N \). If not, go to step 6.

8. If the assignment was successful, remove \( J \) from Jobs, and go to step 5.

9. If the assignment was not successful, undo the last successful assignment and go to step 5. (back track)

The ordering heuristics proposed by [ILS06] are ordering nodes based on the number of compatible jobs (the node that has the highest number is the first). That way, the algorithm will try the node with the most freedom first.

Job ordering depends on the mutual communication of jobs. The ordering calculates the amount of communication between each pair of jobs, and the pair with highest mutual communication will be placed to the ordered list. This ordering is expected to result in an allocation where the most intensively communicating jobs share the same node, that also results in a reduced inter-node communication bandwidth requirement. These are used for the implementation of our graph transformation based solution.

5.5.8 TT job scheduling

In case of synchronous (or time-triggered) real-time operating systems, the OS level tasks should have an explicit, design-time scheduling that defines the activation pattern of each task. The scheduling is an automatic transformation step that involves an external optimizer. The job and communication scheduling algorithms will be introduced in Section 6.4.

The schedule of a node is represented using the following meta model part:

```plaintext
entity (NodeScheduleTable);
relation (node, NodeScheduleTable, platform.ComponentInstance);
relation (jobTriggerings, NodeScheduleTable, JobTriggering);
relation (period, NodeScheduleTable, Double);

entity (JobTriggering);
relation (job, JobTriggering, pim.ComponentInstance);
relation (baseTime, JobTriggering, Double);
relation (period, JobTriggering, Double);
```

Each execution environment has an associated NodeScheduleTable. The period of the schedule is defined on node level. The time-triggered jobs have (triggerings) that describe the activations of the job in a node period. The first activation is defined by (baseTime), and the period by period. There may be several constraints on the timing attributes, depending on the execution environment and middleware that is used in the actual system.

5.5.9 Communication graph synthesis

The communication graph synthesis is an automatic transformation step. It generates the communication graph of the system that is an abstraction of the platform model containing information only about the topology of the system (consisting of communication links and processing nodes). This will be used by the subsequent communication allocation and configuration steps.

**Definition 63 (Communication graph)** The communication graph \( (CG) \) is a bipartite graph: \( CG = (N, C, L) \), where \( N \) is the set of processing node nodes, \( C \) is the set of communication channel nodes, and \( L = \{(N, C)\} \) is the set of node-channel links.

The edges of the graph are labeled using the following labeling function: Label \( (L) : L \to \text{CommLabel} \), where CommLabel \( \subseteq \text{ET, TT, DS} \). The labeling marks the supported protocols for the given communication link (event-triggered, time-triggered, and data stream, respectively).

The communication graph is described by the following meta model part:

```plaintext
entity (CommGraph);
relation (nodes, CommGraph, CommNode);
relation (links, CommGraph, CommLink);
entity (CommLink);
entity (CommNode);
relation (connection, CommNode, CommLink);
```

The role of the communication graph is that it aggregates the topology information required for the communication allocation and scheduling. The definition of the communication graph synthesis transformation can be found in Appendix B.9.
5.5.10 Data element to Message mapping

Messages are carrying data element values through the communication subsystem. A message is the smallest routable element in our system model. According to the type of data elements packed to it, a message can be event triggered, time triggered, or data stream. As multiple data elements can be mapped to a single message, an additional status flag called *update indication bit* should be added that signs whether or not the data element carries an updated value. Data elements are mapped to messages:

**Definition 64** A message $M$ contains a set of data element mappings. The message has a predefined size (in bits). All mapping elements refer to a data element, to the starting position of the data element in the message, and (optionally) to the position of the update indication bit.

Messages also refer to their sender, and to their receivers (that are CommNodes).

The structure is described by the following VPM meta model part:

```plaintext
entity(Message);
relation(size,Message,Integer);
relation(dataElements,Message,DataElementToMessageMapping);
relation(sender,Message,CommNode);
relation(receivers,Message,CommNode);
entity(DataElementToMessageMapping);
relation(dataElement,DataElementToMessageMapping,pim.DataElement);
relation(startingPosition,DataElementToMessageMapping,Integer);
relation(updateIndicationBitPosition,DataElementToMessageMapping,Integer);
```

The data element to message mapping is a manual mapping step that can be supported by automatic packing algorithms. Several constraints apply to the synthesis of messages. As these can depend on the actual middleware and protocol, we will only introduce several generic ones here. The most fundamental constraint is that the mappings in the message should not overlap and hang out from the message.

**Constraint 6 (Data element to message mapping)** Data element mappings of a message should not overlap or hang out of the message. Formally: Let $Bits_i$ denote the set of bits occupied by a data element in message $M$, and let $Bits_M$ denote the bits of the message:

$$Bits_i = (startingPos_i..startingPos_i + size(DE_i) - 1) \cup updateIndication_i$$

$$Bits_m = (0..size_M)$$

Then: $\forall i,j : Bits_i \cap Bits_j = \emptyset$ and $\forall i : Bits_i \subseteq Bits_M$

In most target architectures, the mixed messages (containing both ET, TT, or stream data elements) are forbidden. This simplifies the timing analysis and the communication scheduling.

**Constraint 7 (Message types)** Each message can only contain data elements that have uniform semantics (time triggered, event triggered, or data stream).

The VTCL implementation of the constraint can be found in Section B.10. Sender and receivers of a message should be the same, as for the mapped data elements. Data element sender and receivers can be inferred using the allocation information and the communication graph introduced earlier.

**Constraint 8 (Message sender and receivers)** Let $M$ denote a message, $M.snd$ denote its sender, and $M.rcv$ the set of its receivers. Similarly, for a data element $DE$, $DE.snd$ is the sender node, and $DE.rcv$ is the set of receiver nodes. Then the following restrictions should apply:

$$\forall DE_i : M.snd = DE_i.snd$$

$$M.rcv \supseteq \bigcup_i DE_i.rcv$$

It should be noted that an implication of the previous constraint is that all data elements in a message must be sent by the same network node.
5.5.11 Message to frame mapping

Frames are protocol-specific units of data transmission. A frame is always bound to a communication channel and has low level protocol-specific format and attributes.

Messages are mapped to frames:

**Definition 65** A frame $F$ has a size (in bits) and contains several message mappings. Each message mapping consists of a reference to the mapped message, a start position and an (optional) update indication bit position.

Update indication, similarly to the data element to message mapping case, is used to determine in the receiver side whether the message has been updated by the sender or not.

The mapping is described by the following VPM meta model part:

```plaintext
entity (Frame);
relation (size, Frame, Integer);
relation (messages, Frame, MessageToFrameMapping);
relation (sender, Frame, CommNode);
relation (receivers, Frame, CommNode);
relation (link, Frame, CommLink);

entity (MessageToFrameMapping);
relation (message, MessageToFrameMapping, Message);
relation (startingPosition, MessageToFrameMapping, Integer);
relation (updateIndicationBitPosition, MessageToFrameMapping, Integer);
```

All elements of the definition are in alignment with that of the message definition, the only new element is the link relation that references the communication channel on which the frame is allocated.

The message to frame mapping step is a manual mapping that can be supported by automatic packing algorithms. This step generates the communication allocation for the system. Constraints similar to the data element to message constraints can be defined. It should also be noted, that as the frame is bound to a communication channel, several low-level protocol specific constraints should also be applied. As a generic implementation platform is assumed for this case study, we introduce only some basic, general constraints here.

**Constraint 9 (Message to frame mapping)** Message mappings of a frame should not overlap or exceed the frame. Formally: Let $\text{Bits}_{i}$ denote the set of bits occupied by a message in a frame $F$, and let $\text{Bits}_{F}$ denote the bits of the frame:

$$\text{Bits}_{i} = (\text{startingPos}_{i}..\text{startingPos}_{i}+\text{size}(M_{i}) - 1) \cup \text{updateIndication}_{i}$$

$$\text{Bits}_{F} = (0..\text{size}_{F})$$

Then: $\forall i, j: \text{Bits}_{i} \cap \text{Bits}_{j} = 0$ and $\forall i: \text{Bits}_{i} \subseteq \text{Bits}_{F}$

In most target architectures, the mixed frames (containing both ET, TT, or stream messages) are forbidden. This simplifies the timing analysis and the communication scheduling.

**Constraint 10 (Frame types)** Each frame can only contain data elements that have uniform semantics (time triggered, event triggered, or data stream).

5.5.12 Message gateway configuration

In order to be able to transmit messages through multiple network segments, the messages should be gatewayed by computing nodes between communication channels. The gateway definition creates static routing rules for messages.

**Definition 66** A message gateway rule references a node that will execute the gateway function, a message that should be transmitted, an incoming frame that contains the message, and several outgoing frames that will carry the message to other network segments. The VPM representation is as follows:

```plaintext
entity (GWRule);
relation (node, GWRule, CommNode);
relation (message, GWRule, Message);
relation (sourceFrame, GWRule, Frame);
relation (targetFrames, GWRule, Frame);
```

Message gateway rule definition is a manual mapping step.
5.5.13 Cluster configuration

Cluster configuration is a protocol specific, manual marking step that includes the specification of low-level communication channel and controller related attributes. The set of attributes is different for each communication protocol, and should defined in protocol-specific additions.

5.5.14 Frame scheduling/priority assignment

This step is an automatic optimization based step that can be complemented by manual customization (marking). The scheduling algorithm depends on the actual communication protocol. In case of ET protocols, no scheduling is done, but the frame priorities (for instance in case of CAN and FlexRay) should be defined. The result of the PIM-PSM mapping process is the complete platform specific model of the system that contains all information necessary for the synthesis of hardware and middleware configuration files, glue code, and linker scripts that are used to build the runtime image of the software system.

5.6 Static Checking of Models During the Hardware-Software Integration Process

As the PIM-PSM mapping process is an iterative, interactive process, it is time and resources consuming. In order to avoid the potential repetition of the whole process, design problems or potential performance bottlenecks should be discovered as soon as possible.

As model evaluation should be live (simultaneously to the editing), extensive analysis methods cannot be incorporated. We propose a static consistency checking framework that is able to evaluate simple constraints on the models based on the actual state of the PIM-PSM process. The potential constraints can be classified along two different dimensions (type and scope) and several categories.

5.6.1 Constraint type

The type dimension of constraints is divided into two categories:

- **Qualitative constraints** are logical statements that define requirements for several system properties. These constraints are mostly structural ones, and do not refer to attribute values. For instance, a typical safety constraint is “two replicas of a given software component must not run on the same computing node”.

- **Quantitative constraints** confine the design space by correlating attributes of the system components. Typical representatives are limitations of the overall system characteristics (temporal, dependability related, power consumption, etc.) expressed as functions of the system architecture and component attribute values.

5.6.2 Constraint Scope

The categorization according to the scope follows a similar logic as standardization.

- **Overall dependability-related** constraints are design rules valid for all dependability-related systems. For instance a constraint that has to be fulfilled by all systems relying on robust partitioning.

- **Domain specific** constraints are derived from standards and recommendations of a given application domain (like automotive or aerospace).

- **Application specific** constraints express the specifics of an individual application.

On each level, different constraints sets or packages can be defined that can be combined in order to cover all necessary verification and validation steps of the actual systems model.

5.6.3 Constraint checking

**Constraint execution scope**

In general constraints can be executed with different scope, in terms of model elements.
The model scope means the global execution of the constraint on the actual system model. This implies that the constraint should contain some logic in order to identify the elements in the model that should be checked by the constraint. (for instance, in case of message consistency checking introduced in Section 5.5.10, the constraint should be able to find each message in the model).

The element scope means that the constraint definition contains a simple filter that is used by the execution framework to filter the model elements. The constraint is called for each model element that fulfills the filter condition. (in case of message consistency, the filter condition can be that the model element should be instance of the Message meta element) This results in a simplified constraint, as it does not need an internal query mechanism.

In most model-driven tools, the second execution model is chosen, as it results in a simpler constraint structure, and the centralized evaluation of filter conditions is also more effective runtime. In the current paper, filter conditions will be restricted to instanceOf constraints, e.g. the filter is always a meta element that’s instances can be checked by the constraint.

Qualitative constraints

Qualitative constraints can be represented using first order logic formulas. Using these formulae, never claim-like negated formulae can be generated that should never be satisfied by the target system (model). These formulae are mapped into graph patterns that represent the counter-examples of the constraint. If one of the patterns can be matched on the system model, the associated constraint is violated.

A limited subset of constraints can also be handled by a method that is supported by the ontology-based checking of constraints. This requires the transformation of the system model to the input of the ontology tool [22], and the mapping of constraints to ontology queries. This method has limitations due to the expression power of the background mathematics used in ontologies and due to the fact that the models need to be transformed in a different modeling notation for analysis.

Quantitative constraints

Quantitative expressions can be generated using the actual system structure and attribute values. The resulting formulae -usually containing open variables- can then exported to an appropriate mathematical solver that can either find a possible solution for the open attributes, or it can state that the equation system is inconsistent. The first case means that the constraints are fulfilled, the second case means that some of them are not fulfilled. On implementation level, this check can be done with a simple constraint solver package or using an external mathematical tool.

5.6.4 Integration to the mapping framework

Verification of design constraints is an integral part of the hardware-software integration tool chain. In order to optimize the evaluation of the different constraints, each of them can have trigger conditions. The trigger is a logical formula on the state of the PIM-PSM mapping and enables the constraint only if the trigger is satisfied by the current PSM model. For instance, allocation validation constraints should only be enabled after the completion of the allocation step.

Definition 67 (Step states) Integration steps can reside in different states during the execution of the hardware-software mapping process. In general, the following states are distinguished:

- Start is the initial state of the step.
- InProgress state denotes an intermediate state, where the step execution is started, but still incomplete. As automatic steps are treated as atomic actions, this state is a transient state in such steps.
- Done state is active, if the step has been completed.

The set of possible states: \( \text{States} = \{ \text{Start, InProgress, Done} \} \)

The possible transitions between these states are illustrated by the state machine in Figure 5.10. The Start state is left if the first manipulation (for instance adding a mapping in case of a mapping step) is done. InProgress state is active during the execution, and is only left after the last possible manipulation action (for instance, after the last element is mapped). Done state is active if the step execution has been finished, but can be left if a remove action is executed. The possible remove actions are depending on the
step type, but usually mean the removal of information that has been added to the model by the actual step. For instance, in case of mapping steps, remove means the removal of an existing mapping from the model.

![State machine of an integration step](image)

**Figure 5.10: State machine of an integration step**

In order to be able to define trigger conditions for the constraints, an auxiliary function is needed:

**Definition 68 (State function)** Function \( \text{State} : \text{Step} \rightarrow \text{States} \) returns the actual state of an integration step.

Using this function, arbitrary expressions can be built that refer to the state of the integration steps. Further efficiency enhancement can be achieved if a precedence of the constraints can also be defined. Typically, several different constraints can be applied for a model element. These range from the elemental syntactic checks to the complex model analyzers. The complex checks often require a syntactically correct model, thus in case of independent constraints they also have to re-check the basic consistency of the model part they are working on.

In order to avoid this type of overhead, the constraint framework is able to handle precedence relations between constraints that express that a constraint \( C_b \) builds on the successful execution of constraint \( C_a \). If \( C_a \) already resulted in a problem report, \( C_b \) will not be executed.

**Definition 69 (Constraint precedence)** Constraint precedence is a partial ordering on the set of constraints: \( C_a <_P C_b \), if \( C_b \) should only be executed after a successful execution of \( C_a \).

Based on the elements introduced in the Section, the definition of the constraint is as follows.

**Definition 70 (Constraint definition)** The definition of a constraint is a tuple:

\[
\text{Const} = (\text{Filter}, \text{Prec}, \text{Trig}, \text{Executable})
\]

where:

- \( \text{FilterEntity} \) is the filter condition meta element.
- \( \text{Prec} \) is the set of preceding constraints
- \( \text{Trig} \) is the trigger condition
- \( \text{Executable} \) is the constraint executable that can be one of the following:
  - a VPM pattern: if the pattern is fulfilled, the constraint is also fulfilled
  - an ASM rule: if the rule succeeds, the constraint is fulfilled

This framework enables the definition of constraints and constraint packages dynamically thus allowing the customization of the constraints that are used to validate the system models. The constraint checking framework has been implemented as part of the DECOS tool chain [17], [18].
5.6.5 Constraint checking examples

Typical hardware-software integration tools include a large set of constraint checkers that cover different aspects of the models. In this Section we will show several examples to illustrate the typical checks.

The most basic checks include the model well-formedness rules. These checks validate the models against the metamodel well-formedness criteria (multiplicities, OCL constraints defined in the metamodel). These checks complement the basic validation framework of the metamodelling environment that usually is not capable of checking all metamodel constraints.

The second group of checks contains the object-local checks that verify different semantical properties in the context of a single object or sub-tree. An example is the checking of properties of a model element representing a communication cluster. Clusters usually have several dozens of interdependent properties, and dozens of numeric constraints between the parameters. The checking of such setup is implemented by object-local checks.

The third group of checks consists of global model checks that verify semantic properties that require the traversal of different parts (sub-trees) of the model. A typical example is the consistency check between the communication definition of the software architecture model (ports and connectors) and of the PSM (signals, communication controller configurations).

All the previous checks are working on the models that are used during the hardware-software mapping process. The forth group, however, includes complex analysis checks that convert parts of the models to analysis domains and perform analysis on the derived models. An example is the high-level availability analysis that transforms the allocation information to a mathematical model and calculates the predicted availability of the current allocation. The result can be compared to the input requirements. Some of the complex checks can also give predictions that warn the developer that the current model will become infeasible in a later mapping step. For instance, based on the allocation, an early schedulability analysis can be done before actually getting to the scheduling phase.

5.7 Related work

Model-driven development of embedded systems is wide research area that includes target application domains from traditional, hard real-time systems to mobile phones, and low power peer-to-peer sensor networks. A large number of papers and research reports are covering the various aspects of the topic. MDD-related research, similar to the one presented in the current chapter is done at the Vanderbilt University (Tennessee, USA). The model-driven development environment developed by them is based on the GReAT model transformation tool [KASS03]. They also identified the limitations of MDA, and propose a novel approach to overcome these limitations [KAL03] that is slightly different from the proposal discussed in the current thesis, as it proposes the concept of Model-Integrated Computing (MIC) [KA03], composed of the usage of multiple, interlinked aspect models during the development that are mapped by user-created transformations. Our approach, in contrast with MIC offers a pre-defined (but customizable) integration process definition and pre-built model-driven tools and transformations. The CosMIC (Component Synthesis with Model Integrated Computing) tool [SGN+02] implements the MIC approach and supports the development of component-based embedded systems.

A modified MDA approach is used for embedded systems design in [PB03], combining the existing “Y” development approach with the MDA model-centric vision. The proposed method starts with the definition of application and platform PIMs, that are integrated using a third model, called association PIM. The association PIM is then transformed into the deployment PSM model. All the transformations are treated as atomic, automated mappings.

A UML-based approach is proposed in [dNdSOW+06] that consists of the specification of an application PIM, and a platform model (PM), that are integrated using a mapping model. The mapping model defines transformation rules that implement the PIM-PM integration and the synthesis of the implementation model (or PSM) of the system. The proposed approach suggests the definition of several transformations in the mapping model to generate alternative PSMs based on different mapping rules. The process can only be influenced through the specification of mapping rules and transformations, as all the transformations are atomic, automated ones.

The ARTIST and ARTIST2 [Cona] Network-of-Excellence initiatives, funded by the European Commission target the definition of methods for embedded systems development, including modeling, synthesis, analysis, and testing aspects, and provide education and dissemination on the related results and trends.

The AutoSAR standard [Alla] (that has been introduced earlier) also proposes a development process based on models and model integration, but it does not specify tools or techniques for the implementation
of different steps of the process. The importance of the standard process definition is that it can be used to develop pre-built hardware-software integration tool environments for automotive developers.

5.8 Applications

Model-driven development of time-triggered systems

The methodology and techniques introduced in this Chapter were elaborated and implemented in the Framework of the DECOS EU Framework 6 IP that aimed at the development of novel, model-driven methods for the development of safety-critical, time-triggered embedded systems.

Model-driven development of Aerospace systems

The methodology and the tools that have been developed in the DECOS project have been adapted in the DIANA EU Framework 6 Project addressing the development of ARINC 653 compliant airborne applications. The PIM-PSM mapping is built on the framework introduced in the previous Section. The modular architecture has proved to be customizable enough to fit the slightly different needs of the DIANA target architecture.

Methodology framework for the new, Generic Embedded Systems Architecture

The proposed methodology and tools are part of the methodology framework defined by the ongoing GENESYS EU FW 7 Project [Conb] that aims at the definition of a uniform architecture and design methodology for embedded systems development. The target application domain of GENESYS is wider than the traditional safety critical systems (as targeted by DECOS and DIANA) and also includes consumer, multimedia, and mobile systems. The implementation platforms range from the traditional micro-controller-based approach to programmable network-on-a-chip (PNoC) solutions. The introduction of these new domains and implementation techniques required a more flexible hardware-software integration solution that is built also on the basic iterative, interactive approach introduced in this paper. A first report on the development of the GENESYS PIM-PSM mapping tool has been published recently [Kad08]. Further utilization of the results and the implementation of the methods is foreseen in the upcoming INDEXYS (Industrial Exploration of the GENESYS Architecture) project that aims at the implementation and evaluation of the methods and architectures defined in the GENESYS project.

5.9 Conclusions

As a conclusion for the current chapter, I have developed novel methods for the model-driven design of embedded systems. I specified modeling languages for application architecture and platform modeling of the target domain. I proposed a workflow-based approach for hardware-software integration processes, that has already been used in several different industrial and international research projects. As an extension to the hardware-software integration framework, I also developed a constraint checking framework that supports the efficient, automatic execution of constraints on intermediate models during the integration. A step-by-step example introducing a PIM-PSM mapping from the developer perspective (based on the DECOS mapping tool) can be found in Appendix D.

Contribution 2 I have developed a novel approach for model-driven development of distributed, dependable embedded systems. In particular, the following novel procedures were elaborated:

2.1 I proposed a domain-specific architectural, platform, and system modeling style based on the combination of current modeling standards that follows strong component orientation and supports functional and non-functional property definition. [14, 19–21]

2.2 I extended the traditional MDA approach with an iterative, interactive hardware-software integration framework that is capable of handling complex system architectures and models. [13, 14, 19, 21]

2.3 I defined a static constraints checking framework that is integrated with the hardware-software design process in order to give early feedback on design errors to the developer. [3, 11, 17, 18, 21]
5.9.1 Out of scope

Although the methods proposed in the current chapter support the development of applications on an integrated architecture, it should be noted that the development of platform components and services is not supported by them. This is a result of the fact that the development of platform components and application components usually requires a different approach, and the customization of current model-driven tools requires significant amount of efforts. The platform components are developed using a more conservative approach, with general purpose modeling language support, and a limited utilization of generative technologies.
Chapter 6

Dependability Driven Synthesis

6.1 Introduction

The development method for embedded systems introduced in the previous Chapter relies on different analysis and synthesis methods in order to be able to verify, validate, and synthesise system models. While QoS properties play a key role in these systems, they are usually neglected during the system development and they are also not involved in the traditional MDA workflow. Typically, the QoS assessment takes place only in the late integration testing phase. This means that the potential QoS problems are not handled in the early phases of system design, which can result in dramatic cost increase (in case of system redesign), or inadequate service quality of the system.

To avoid such risks, we suggest using formal analysis techniques during the early system design phase to assess the key quality aspects of the system. This allows the early recognition of potential problems and the avoidance of them by modifying the system design before the implementation phase. This results in lower development costs and time, because the QoS problems can be eliminated in much earlier phases of development than in the current practice.

6.2 High-level availability analysis

High-level availability analysis supports the developer in predicting the availability characteristics of the system under design. We will introduce an availability calculation method for component-oriented distributed system in this section. We will use the simplified architecture and platform models (SAM, SPM) from the previous chapter to illustrate the methodology.

The basis of the availability calculation is the availability of the atomic components that build up the system. These values can either be measured, or be predicted based on test results. The goal of the analysis is to calculate the availability of the service that the system delivers to its users. In typical embedded systems, the inputs are sensors, and after some internal calculations and communication, the service is delivered as an action on the actuators of the system.

6.2.1 Related Work

Application model analysis has been investigated by several researchers in the past. Foster et al. [FUMK03], [FUMK05] have worked out a method for verification of web service compositions, but only from the functional modeling viewpoint. Using that technique together with our solution is a combination for functional and non-functional analysis of service-based system models.

A survey on the possible model-based performance prediction methods has been done by Balsamo et al [BMIS04]. This work summarizes the most important methods in this area. However, it only contains performance evaluation without considering other aspects of non-functional properties. Jin et al. [LjJ02] present a method for simulation-based analysis of web service compositions. They have focused on the timeliness analysis of applications, and do not calculate dependability related properties.

Current business process modeling tools that support the modeling of service composition like IBM Websphere Business Integration Modeler [Cor] are able to simulate the execution of service compositions but their capabilities are limited to the temporal domain. Our approach can also be used to complement the analysis capabilities with dependability analysis.
Similarly, the SCADE tool suite [Teca] can do model-checking of the behavioral model of the application, but is also limited to functional property verification. However, the source code generated by this tool is proven to be correct, the properties of the deployed system cannot be calculated by the tool.

There are several research results in this field that focus only on the derivation of analysis models and do not deal with the rest of the development workflow, like [GPG+05], [Gön05]. These works are useful if we want to integrate new aspects to the analysis framework. The transformation of system models to analysis models can be created based on these results.

6.2.2 Availability calculation

The analysis can be carried out on either PIM or PSM levels, where the PIM level only calculates based on application component level, while the PSM-based analysis uses also the platform characteristics for the calculation. While the scope of these scenarios is different, the underlying calculation method is the same.

Availability definition

First we give a definition of availability according to [ALRL04].

Definition 71 (Availability) Availability is the readiness of the system/component/application to deliver correct service.

More specifically, we can also define the availability of a component in our context.

Definition 72 (Component availability) Component availability ($A_{comp}$) is the probability that the component produces correct outputs upon receiving correct inputs.

If the component is replicated, the availability of the replicated structure can be calculated as:

$$A_{repl} = 1 - (1 - A_{comp})^{\text{redDegree}}$$

The availability of the outputs of the component can be calculated as follows:

$$A_{out} = A_{in} \times A_{comp}$$

We assume that the faults affecting the system are independent and that the components are fail silent. It should be noted that these assumptions are in line with typical integrated safety-critical system assumptions [HKS04b].

Availability model

The availability model is an abstraction of the system model that focuses on the structure and dependencies of the components and serves as a basis for the availability calculation.

Definition 73 (Basic Availability graph) The availability graph is a directed, labeled graph $G_a = (V, E, L)$, where the nodes ($V$) represent components, edges ($E \subseteq (V, V)$) represent dependencies between components, and the labeling ($L : V \rightarrow [0..1]$) is the availability of the component.

The availability of the output of the system can be computed using the graph. The availability of a component output in general depends on the availability of the component, and of the availability of all components that the current component depends on. The dependency has to be computed recursively, and the set of components equals to the set of reachable graph nodes from the node that represents the component under consideration. More formally, the availability of the output of component $C$ can be calculated as:

$$A_{outC} = L(C) \prod_{\forall \text{reachable } C_i} L(C_i)$$

The system availability equals to the availability of the output of the final (actuator) component that actually delivers the service visible to the user.

The limitation of the basic availability graph is that it does not support the introduction of alternative paths in the model. A component output is available if all the inputs are available and the component is working normally. In the safety-critical systems domain, however, it is a common practise that redundancies are built in and an input data element can be sent by multiple components, and (supposing a fail-silent component model) if at least one of these inputs arrives, the component can continue processing. This necessitates the introduction of so-called voter components that multiplex the alternative paths.
6.2. HIGH-LEVEL AVAILABILITY ANALYSIS

Definition 74 (Extended Availability graph) The availability graph is a directed, labeled graph \( G_a = (C, M, E, L) \), where the \( C \) represents the set of components, \( M \) represents a set of voters, \( E \subseteq ((C \cup M), (C \cup M)) \) represents the set of dependencies between components and voters, and the labeling \( L : C \to [0..1] \) is the availability of the component.

In this more complex case we will transform the model to a fault tree. The steps of the conversion are outlined by the following algorithm.

Algorithm 5 (Availability graph to fault tree) Initialization: Let \( G_a \) be the availability graph, and \( FT \) the fault tree under construction. Let \( V_0 \) be the output component (the target of availability calculation). Let \( N_{root} \) be the new root node of \( FT \). Let \( N_{act} = N_{root} \), and \( V_{act} = V_0 \).

BuildTree(\( V_{act}, N_{act} \))

1. if (\( V_{act} \) is component node \( C_i \))
   (a) Create OR gate \( G \) and connect its output to \( N_{act} \)
   (b) Create Basic Event node \( BE_i \) and assign probability \( 1 - L(V_{act}) \) to it.
   (c) Connect \( BE \) to an input of \( G \)
   (d) Let \( N_{act} = G \)
2. else (\( V_{act} \) is a voter node)
   (a) Create AND gate \( A \) and connect its input to \( N_{act} \)
   (b) Let \( N_{act} = A \)
3. ∀ Node that is reachable from \( V_{act} \):
call BuildTree(Node, N_{act})

Proposition 21 If the availability model is acyclic, the above algorithm always terminates.

Proof The proof of the termination property is straightforward. Steps 1 and 2 perform atomic activities. Step 3 invokes the BuildTree routine on the nodes reachable from the actual node \( V_{act} \). The model is acyclic, meaning that the set of possibly reachable nodes is \( V \setminus V_{act0} \). If the next processed node is \( V_{act1} \), in that step the set of possibly reachable nodes is \( V \setminus \{V_{act0}, V_{act1}\} \). This way at most \( n = |V| \) iterations can be done, as without cycles \( n - 1 \) is the length of the longest dependency path in the model. □

The resulting fault tree represents the possible error propagation paths in the model, and its basic events have assigned fault probability. The evaluation of system level fault probability can be calculated using for instance the algorithm of Veeraraghavan and Trivedi [VT91].

Unfortunately, the above analysis method can only be used in case of acyclic models. Most control algorithms, however, contain feedback loops that introduce cycles in the inter-component dependency graph. Several solutions have been proposed [BB06] in order to overcome this problem and be able to generate fault trees from cyclic models. These methods either require the inclusion of additional information to the model (on system dynamics), or require external tools (for instance, model checkers) to discover the state space of the system model and support the building of fault trees.

We will introduce an alternative method that utilizes the fault tree evaluation methods but is applicable directly on the cyclic availability model in order to avoid using extensive analysis methods or depend on user interaction. The basic idea of the approach is that the availability model can be treated as a constrain satisfaction problem over boolean variables, and the possible cut sets (set of components that can cause system-level failure) can be calculated using constraint solving techniques. The cut sets can be converted to a set of disjunct event sets and these sets can be finally used for availability calculation. We will introduce the steps of the process in more detail in the following paragraphs.

From availability model to constraint satisfaction problem We define constraint model components for the availability model components and voters, and connect them together based on the dependencies.

Definition 75 (Vocabulary of the constraint network) The vocabulary (or domain) of constraint variable is the set of bool values: \( \text{Voc} = \text{false, true} \), where false means error-free state, and true means erroneous state (component not available, message not available).
Definition 76 (Constraint node for components) A constraint node \( N_c(I_{in}, S_{ate}, O_{ut}) \) is used to model components, where \( I_{in} \) is the set of input message links \( (I_{in} = I_{1},...,I_{n}) \), \( S_{ate} \) stands for the internal state of the component, and \( O_{ut} \) denotes the output message link of the component. The behaviour is specified as follows:

\[
O_{ut} = S_{ate} \lor (\bigvee_{j=1..n} I_{j})
\]

Meaning that if either an input or the internal state is erroneous, the component will propagate the error.

Definition 77 (Constraint node for voters) A constraint node \( N_v(I_{in}, O_{ut}) \) is used to model voter elements, where \( I_{in} \) is the set of input message links \( (I_{in} = I_{1},...,I_{n}) \), and \( O_{ut} \) denotes the output message link of the component. The behaviour is specified as follows:

\[
O_{ut} = \bigwedge_{j=1..n} I_{j}
\]

Meaning that if all of the inputs are erroneous, the component will propagate the error. We treat voters as stateless elements.

It should be noted that the current interpretation of the voter components models the case, where components are fail-silent, therefore an \( n \)-way voter can mask \( n-1 \) errors. The model can be modified to match other error models if needed. The constraint nodes are used by the following algorithm that builds the constraint satisfaction network.

Algorithm 6 (Constraint network building) The input is the availability model \( M_{avail} = ((C,M), E, L) \) of the system.

1. for each component \( c_i \in C \) create a constraint node \( N_{c_i}(I_{in}, S_{ate}, O_{ut_i}) \).
2. for each voter \( m_j \in M \) create a constraint node \( N_{v_j}(I_{in}, O_{ut_i}) \).
3. for each \( e_i = (c_i, c_j) \in E, c_i, c_j \in C \): \( N_{c_i}.I_{in} = N_{c_i}.I_{in} \cup \{N_{c_j}.O_{ut}\} \) add the output of the target component to the set of inputs of the source component.
4. for each \( e_i = (c_i, m_j) \in E, c_i \in C, m_j \in M \): \( N_{c_i}.I_{in} = N_{c_i}.I_{in} \cup \{N_{m_j}.O_{ut}\} \) add the output of the target voter to the set of inputs of the source component.
5. for each \( e_i = (m_i, c_j) \in E, m_i \in M, c_j \in C \): \( N_{m_i}.I_{in} = N_{m_i}.I_{in} \cup \{N_{c_j}.O_{ut}\} \) add the output of the target component to the set of inputs of the source voter.
6. for each \( e_i = (m_i, m_j) \in E, m_i \in M, m_j \in M \): \( N_{m_i}.I_{in} = N_{m_i}.I_{in} \cup \{N_{m_j}.O_{ut}\} \) add the output of the target voter to the set of inputs of the source.

Proposition 22 The above algorithm always terminates.

Proof The proof is straightforward, as step 1 executes exactly \(|C|\) times, step 2 executes exactly \(|M|\) times and steps 3-6 execute exactly \(|E|\) times. \( \square \)

Proposition 23 The result of the algorithm is a sound model, i.e. it reflects the structure and relationships of the source availability model.

Proof It is clear that steps 1 and 2 create the components that represent the application component and voter behaviors. New variables are created for the outputs of these components, and for the internal state. During steps 3-6, the input sets of the components are appended according the edges of the original model. If there is an \( e = (src, trg) \) edge in the model, then the output of \( trg \) is put to the set of inputs of \( src \). This means that the dependency relations are modeled as opposite direction constraint value propagation relationships. Each component, voter, and edge will be represented that way in the model, so the constraint satisfaction model is reflecting the structure and relationships of the source model. \( \square \)

The resulting model can be used to generate the cut sets of the availability model. We should add an external constraint \( C_{out}.O_{ut} = true \) to the model, where \( C_{out} \) is the output component and the constraint limits the solutions to the faulty cases (where the internal errors propagate to the system output). We can use an arbitrary constraint solver to get the \( Res = S_1...S_n \) set of solutions, where \( S_i = (N_{c_i}.S_{ate_i},...,N_{c_n}.S_{ate_i}) \) is an array of component states. Each \( S_i \) represents a system state where the error propagates to the output.
Definition 78 (State vector subsetting) If there are two state vectors $S_i, S_j$, then we say that $S_i$ is a subset of $S_j$ ($S_i \subseteq S_j$), if $S_i = (St_{i1}, \ldots, St_{in})$, $S_j = (St_{j1}, \ldots, St_{jn})$:

$$(\forall k \epsilon 1..n, St_{jk} = false \implies St_{ik} = false) \land (\forall k \epsilon 1..n, St_{ik} = true \implies St_{jk} = true)$$

$\land \exists m \epsilon 1..n, St_{jm} = true \land St_{im} = false$

Meaning that $S_i$ contains false values at positions, where $S_j$ also contains false values, and $S_j$ contains true values at positions, where $S_j$ also contains true values, and there is at least one position where $S_j$ is true and $S_i$ is false.

Definition 79 (Minimal cut set) A cut set $S_i$ is minimal if there are no $S_j \subseteq S_i$ so that $S_j$ os also a cut set.

$Res$ is a set of cut sets, but these are not necessary minimal, meaning that there may be two solutions $S_i, S_j$ so that $S_i \subseteq S_j$. The evaluation algorithm that will be used for the calculation of quantitative availability values requires minimal cut sets, that will be achieved using the following algorithm:

Algorithm 7 (Minimal cut set selection)

1. while there are $S_i, S_j \epsilon Res, S_i \subseteq S_j$

   (a) $Res = Res \setminus S_j$

Proposition 24 The above algorithm always terminates and the resulting set of solutions contains only the minimal cut sets of the problem.

Proof Termination: The algorithm removes a solution in each iteration, and the iteration condition requires that $Res$ contains at least two solutions. That means that the maximum iteration count is $|Res| - 1$.

No minimal cut sets are removed: Initially, $Res$ contains all possible cut sets of the problem that represents the system error propagation model, including all the minimal cut sets. The algorithm removes those cut sets that have subsets in $Res$. As minimal cut sets do not have subsets (by definition), these cannot be removed.

Only minimal cut sets in the result: Initially, $Res$ contains all possible cut sets of the problem that represents the system error propagation model, including all the minimal cut sets. The algorithm removes all cut sets having a subset in $Res$. That results in a $Res$ set that contains only those cut sets not having subsets, and by definition, these are minimal cut sets.

We have shown that no minimal cut sets are removed, and only those are present in the results, so we can state that the resulting set contains all minimal cut sets of the model. $\square$

The set of minimal cut sets represents the possible minimal error states where the component errors can propagate to the output of the system. These event sets should be quantified to get the probability of system-level un-availability. The cut sets are, however, not necessary disjunct, so the basic probability calculation techniques cannot be applied. We used the algorithm presented in [VT91] that can be applied to non-independent event sets and creates the independent equivalent of them and calculates the attached probability values. Component probabilities are in our case the un-availability values (probability that the component is not available $1 - A_{comp}$) and the result is also the probability that the system is not available $(1 - A_{system})$.

It is possible to calculate the availability values also for systems containing feedback edges using this method. For systems with non-circular dependency structure the traditional fault tree based analysis can also be applied.

### 6.2.3 PIM level analysis

The PIM-level analysis of system models results in a prediction of application availability on an ideal, error-free platform. This analysis is able to detect design problems (single point of failures, reuse of unreliable components, lack of replication) in an early stage of the design process (software architecture design). It can also be used for the comparison of different software architectures or components.

**Availability model synthesis**

The availability model is automatically generated from the SAM of the application using a graph transformation. The following algorithm outlines the transformation.
Algorithm 8 (SAM to Availability model) Let $M_{PIM}$ be the flattened PIM model, and $C_{main}$ its main composition. The flattened model contains only atomic component instances in the main composition. The algorithm uses this to build the basic availability model $G_{PIM}$.

1. for each $C_i$ component instance in $C_{main}$: create $V_i$ vertex in $G_{PIM}$.
2. for each $(V_i, V_j)$, where there is an assembly connection from source element $C_j$ to target element $C_i$, add $(V_i, V_j)$ as an edge to $G_{PIM}$.
3. for each $C_i$ get the availability value $A_{C_i}$ and the redundancy degree $\text{redDegree}_i$ from the NFP model of the PIM, and set: $L(V_i) := 1 - (1 - A_{C_i})^{\text{redDegree}_i}$.

Proposition 25 The above algorithm always terminates.

Proof This a trivial consequence that steps 1 and 3 execute exactly once for all components, step 2 executes exactly once for all dependencies, and the activities of these steps are atomic ones (model element creation or attribute setting).

Proposition 26 The result of the algorithm is sound, i.e. it reflects the dependency structure of the input model.

Proof Step 1 of the algorithm creates a $V_i$ vertex for each component instance in the main composition of the application, and step 3 assigns the corresponding availability values to these vertexes. This means that all component instances $C_i$ are represented in the target model by a single vertex $V_i$ that has the corresponding availability value.

The dependency relations in the source domain are represented by communication links (assembly connections). The execution of a component instance $C_i$ requires all incoming messages to be present. Step 2 of the algorithm creates dependencies to all senders that send messages to the component, so this dependency is also properly mapped to the target model.

Both the components, and component dependencies are properly represented in the target model, so the model is sound.

Availability calculation

The generated availability model can be directly used for availability calculation, however, as the model does not contain explicit information about the components that represent the final, service output of the application, the user has to add additional (marking) information to the software component model. Based on this, the availability of the marked component is the availability of the application.

From the description of the algorithm, it can be seen that the availability model building and evaluation is a relatively simple task; therefore it can be implemented as an on-line check during PIM model editing. The analysis tool can continuously re-calculate the predicted availability value of the application on model changes and issue warnings to the developer if the value falls below a predefined threshold value.

6.2.4 PSM level analysis

PSM level availability analysis extends the PIM-level with the platform related information. It can be used for deeper analysis and also for the evaluation of deployment alternatives with respect to the consequences on the system availability. The availability model is based on the PSM model of the system that contains information about the application, the platform, and the hardware-software allocation.

Availability model synthesis

The availability model is built from the post-allocation PSM of the system. We have chosen to model nodes as single, atomic components, however, the mapping method could be extended to allow for building a more detailed analysis model. The rationale behind our decision is that hardware nodes (or ECUs) are usually COTS components that are black boxes from the system integrator point of view, and the hardware manufacturer specifies MTBF values on node level. The following algorithm builds the analysis model based on the PSM:
6.3. DEPENDABILITY DRIVEN ARCHITECTURE SYNTHESIS

Algorithm 9 (PSM to Availability model) Let $M_{PSM}$ be the post-allocation PSM model, and $C_{main}$ its main composition. The PSM contains a flattened, replicated component model consisting of only atomic component instances in the main composition. The algorithm builds availability model $G_{PSM}$.

1. for each $C_i$ component instance in $C_{main}$: create $V_i$ vertex in $G_{PSM}$.
2. for each $i$ hardware node in the model, create $V_{N_i}$ vertex in $G_{PSM}$.
3. for each $(V_i, V_{N_j})$, where there is an allocation connection from component $C_i$ to node $N_j$, add $(V_i, V_{N_j})$ as an edge to $G_{PSM}$.
4. for each $C_i$ component and for each I input interfaces of $C_i$ let $Conn$ be the set of components connected to I.
   a) if $|Conn| > 1$
      i. Create voter component $M_v$
      ii. Add $(V_i, M_v)$ to the edges of $G_{PSM}$
      iii. for all $C_j \in Conn$ add $(M_v, V_j)$ to the edges of $G_{PSM}$
   b) else $Conn = C_k$, add $(V_i, V_k)$ to the edges of $G_{PSM}$
5. for all $C_i$ get the availability value $A_{C_i}$ set: $L(V_i) := A_{C_i}$.
6. for all $N_i$ get the availability value $A_{N_i}$ set: $L(V_{N_i}) := A_{N_i}$.

Proposition 27 The above algorithm always terminates.

Proof The proof is straightforward, as all steps have a finite number of iterations: steps 1 and 4 execute once for all component instances, step 2 and 5 once for all node instances, and step 3 executes once for all allocations. Step 4 executes once for all interface of all components. The sum of these steps is bounded by the model size, so the algorithm will always terminate if the model is finite.

Proposition 28 The result of the algorithm is sound, i.e. the model reflects the structure and dependencies of the source model.

Proof Steps 1 and 2 create the vertexes representing application components and platform nodes in the target model, and steps 4 and 5 set the appropriate availability values to these elements. Step 3 is enumerating all input interfaces of all components and inspecting the assembly connections. If an interface is connected to a single sender, then a simple dependency is added between the receiver and sender, representing the dependency implied by the communication. If there are multiple senders to a single interface, a voter component is also added that represents the voting on input messages in the middleware. The voter depends on the message senders, and the message receiver depends on the voter. We can state that a) all nodes and components are mapped together with their availability values, b) all communication links are mapped to dependency structures therefore the target model reflects the structure and relationships of the source model.

The availability calculation is based on the generated availability model. The system designer has to specify the output component that implements the system-level output. The availability of that component will be the overall, user perceivable system availability.

6.2.5 Conclusions

High-level availability calculation methods have been introduced in the current section. The methods allow the prediction of availability both on PIM and PSM levels offering an early feedback to the designer during the application modeling and hardware-software integration phases of the development process. These methods can be used to get a rough estimate of system reliability and to compare alternative system architectures during the design phase.

6.3 Dependability driven architecture synthesis

Hardware architecture design is usually a highly intuitive process in that the designer tries to find the most suitable distributed architecture for an embedded system. The process involves several (often orthogonal) constraints from the domains of performance, reliability, power consumption, and others. We propose an optimization-based automatic architecture synthesis method in the current section that creates a platform setup and component deployment based on performance, availability, and cost factors.
6.3.1 Related work

The model-driven analysis of QoS attributes of component-based systems under design has recently become a hot research topic. Primary focus is usually put on performance issues such as, e.g., in [SRTB08,BM04,SE03]. The early assessment of traditional dependability attributes is carried out in [Gok07,CSC02]. In most of these papers, a traditional transformation-based approach is followed where the QoS parameters are generated from a higher-level initial model (semi-)automatically. In contrast to these approaches, we focus on availability and cost parameters of deployment.

In [MPB03], the authors define a method for dependability analysis of systems based on UML models. The basic idea behind that method is the transforming UML models to Timed Petri Nets. The starting point of the method is the architectural level model, so it works on a static infrastructure and does not modify the systems architecture.

Probably, the most closely related work is that work of Bastaricca et al. [BCDS01], where the authors describe two deployment optimization methods that can be used in a distributed component-based environment. Both algorithms do the optimization of the deployment, but they work on a static infrastructure that cannot be modified. This way, they cannot be used for infrastructure planning, only for deployment on existing hardware environments. Moreover, the algorithms do not optimize for cost, but for network utilization.

6.3.2 Fault model

Embedded computing nodes consist of several layers of hardware and software components. We assume that errors can only occur in lower levels (illustrated by Figure 6.1), either in the hardware or in the operating system and middleware levels. Errors of the higher level components can be easily and rapidly detected and repaired by the local diagnostic subsystem that can restart the failed component. Hardware and operating system errors cannot be repaired as fast as the higher level errors. This results in a much longer downtime. Even if the severity of the hardware errors is lower than that of the software errors, the overall service downtime is much higher because of the longer repair time.

Anticipated hardware faults include several fault types. One of the most important challenges is the bit flip faults – incidental changes of memory cells – caused by external radiation or electro-magnetic fields, manifesting in transient hardware faults. Hardware development faults are rare but their influence on overall system reliability can be significant, as they can cause unexpected changes in system state. The third important fault group is the set of permanent hardware faults that can only be handled by replacing or repairing the hardware units.

In the operating system and middleware layers we only anticipate design faults that are introduced by incorrect or incomplete specification, design, implementation, or verification and validation of the software modules. These faults cause incorrect behaviour of system calls and thus influence the functionality of the higher software layers.

Our fault model supposes that the higher level software components are stable enough not to cause a significant downtime (e.g. they do not contain design faults). As the application modules are typically generated by automatic code generators from a higher level model, such as Simulink or SCADE, we suppose that the application components cause no errors. The fault model introduced here has several limitations. If running on a dependable, highly redundant hardware, where hardware component and operating system errors do not cause system restart higher level software errors will be dominant in service downtime. As the continuous trend of miniaturization leads to the dominance of the increasing rate of transient hardware errors (bit flips, EMC issues) due to external factors, the fault hypothesis is satisfied in most cases.

6.3.3 Redundancy patterns

There are several patterns for creating redundant architectures in order to achieve high availability. Our solution uses N-way redundancy of components in order to increase availability. A replica set is a set of computing nodes that execute copies of the same component in parallel to be able to tolerate the fault of one or more copies. The result of the computation is fed to its consumers through voter components that filter the faulty data elements to mask the output of failing component instances. Our synthesis solution is able to configure redundancy degrees automatically in order to meet the global availability requirements.

6.3.4 Application and resource model

The resource model is the analogy of a hardware catalogue that contains information about the available node types, their reliability-related values, price, and the number of available units. It should be noted,
that the hardware manufacturers usually specify the MTBF (Mean Time Between Failures) value for their products, but for our analysis, the availability values are needed. Given the MTBF, and MTTR (Mean Time To Repair) values availability can be calculated as follows:

\[
A = \frac{MTBF}{MTBF + MTTR}
\]

MTTR values are usually specified based on the physical location of the embedded system (field deployment, system on a vehicle) and the availability of service personnel needed in order to carry out the repairing.

**Definition 80 (Resource model)** The resource Model is a set of tuples: \( M_{res} = \{(Id, N, P, A)\} \), where \( Id \) is the identifier of the node type, \( N \) is the number maximum available instances, \( P \) is the unit price, and \( A \) is the availability.

Our methodology is also able to create replicated nodes in order to achieve higher availability. It uses several identical units to create a cluster of identical nodes executing the same components synchronously. The new node parameters for a \( R \) degree of replication are calculated as follows:

- a new, unique \( Id' = (Id, R) \) is assigned as a pair of the original identifier and the redundancy degree
- \( N' = \lfloor N/R \rfloor \)
- \( P' = P \times R \)
- \( A' = 1 - (1 - A)^R \)

The only limitation of the method is that the clustered nodes are able to execute only identical configurations, so that each deployed component will be executed with the same redundancy degree. The application model that is used for architecture synthesis is a simplification of the SAM introduced earlier.

**Definition 81 (Component model)** The model is a tuple \( M_{comp} = (C, D, W, P) \) where \( C \) is the set of components, \( W \subseteq C^2 \) the set of component dependencies (communication links), \( W : (C, Res) \mapsto \mathbb{R}_{>0} \) is the WCET (Worst-case execution time) specification that also depends on the resource \( (Res) \) on which the component is deployed, and \( P : C \mapsto \mathbb{R}_{>0} \) is the period specification.

The components and their links can be created based on the SAM model, and WCET and period values can be read from the NFP model. Earlier we argued that component compatibility is an important input of the allocation process. The developer can define whether a component is compatible with a resource (type). The current solution also uses the concept of compatibility:

**Definition 82 (Compatibility matrix)** Component compatibility matrix \( CC = (C \times Id) \) is a binary matrix, and an element \( cc_{ij} = 1 \) if and only if component \( C_i \) is compatible with resource \( Id_j \).

The result of the optimization is an hardware architecture and a component allocation. This is described by the following structures:

**Definition 83 (Hardware node vector)** Nodes \( n_1..n_k \) is a vector and its elements \( n_i \) are node identifiers. The dimension of the vector specifies the number of nodes in the current setup, and its elements specify the type of the nodes.

**Definition 84 (Allocation vector)** Alloc \( = (a_1..a_c) \) is a vector and its elements \( a_i = [1..k] \) specify that component \( C_i \) is allocated to node \( a_i \).
In order to keep our formulae more compact, we will use $V_i$ to denote the $i$th element of vector $V$.

Availability requirements can be specified for software components. Usually, the application level availability is attached to the component that implements the final, output activity of the system, but additional requirements can be attached to any other components as well.

**Definition 85 (Availability requirements)** Availability requirement is a function $(A_{req}: C \mapsto \mathbb{R}_{>0})$.

### 6.3.5 Deployment optimization

The deployment optimization problem involves several different constraints that specify its solution space.

**Constraint 11 (Component compatibility constraint)** The component should be compatible with the node it is allocated on.

$$\forall C_i \text{ component } CC(C_i, \text{Nodes}_{\text{Alloc}}) = 1$$

**Definition 86 (Node instance count function)** Node instance count is a function that returns the number of instances of a node that is used by the current architecture:

$$\text{UsedInst}(Id) = \left| \text{Nodes}_i, \text{where } \text{Nodes}_i = Id \right| + \sum_{\text{Nodes}_i=(Id,R_i)} R_i$$

**Constraint 12 (Node cardinality constraint)** The number of used nodes from a specific type should not be more than the maximum allowed count. Replicated node configuration should also be taken into account.

$$\forall \text{NodeType} = (Id,N,P,A) : N \geq \text{UsedInst}(Id)$$

Node workload is calculated based on WCET and period values of components. In theory, 100% can be achieved, but there are always additional load factors as middleware processing, internal delays, etc. Therefore, we introduce a global **saturation factor** ($0 < SF < 1$) that can be used to limit the workload on a more realistic (and system dependent) level.

**Constraint 13 (Workload constraint)** The workload on the nodes should be less than or equal to the saturation factor ($SF$).

$$\forall i, \text{Nodes}_i SF \geq \sum_{C_j,\text{Alloc}_j=i} \frac{W(C_j, \text{Nodes}_i)}{P(C_j)}$$

One of the key constraints is the availability constraint that necessitates the fulfillment of availability constraints. The calculation is based on the simple, high-level availability calculation method of the previous section.

**Constraint 14 (Availability constraint)** First, we will use the utility function $\text{Req} : C \mapsto \{Id\}$ that returns the set of required hardware nodes for a component. The set of nodes includes the node on that the component is deployed, and all other nodes that execute components on which (directly or indirectly) the actual component depends on.

With this function, we can define the following constraint for all components that have predefined availability requirements:

$$\forall C_i, A_{req}(C_i) \leq \prod_{(Id,N,P,A), Id\in \text{Req}(C_i)} A$$

The constraints define the possible valid solutions for the deployment problem, but an objective function is also needed in order to be able to find the **optimal** solution. We have chosen the **cost factor** for optimization, and try to find the cheapest hardware architecture.

**Definition 87 (Objective function)** The objective function is the cost of the whole system (in terms of hardware).

$$\text{Cost} = \sum_{\forall (Id,N,P,A)\in \mathcal{M}_{\text{Res}}} P \ast \text{UsedInst}(Id)$$

**Proposition 29** The constraints and the objective function $\min(\text{Cost})$ define an optimization problem that results in an architecture that fulfills the availability and workload requirements using a cost-optimal deployment.
6.4. SYNTHESIS OF TIME-TRIGGERED SYSTEMS

Proof The proof of the above proposition is straightforward: the fulfillment of the compatibility (Constraint 11), node cardinality (Constraint 12), availability (Constraint 14), and workload (Constraint 13) constraints means by definition the correctness of the deployment. These constraints are reused for the optimization. That means that all solutions of the optimization problem should also be correct. The minimal cost is guaranteed by the objective function.

The optimization problem can be solved using legacy solver tools, like ILOG OPL Studio [ILO].

6.3.6 Summary

We present in this section a method for the synthesis of hardware architecture and allocation for component-based time-triggered embedded applications. The architecture is optimized for component cost while also fulfills workload and availability requirements.

It should be noted, however, that although the architecture fulfills the probabilistic availability requirements, it should be verified and validated against additional requirements (no single point of failure, etc.) as these are not guaranteed to be met. An important point is that we assume the presence of a reliable, high bandwidth network between the nodes, and do not verify whether there is enough bandwidth for communication. This check should also be done in a later phase.

The proposed methodology can be useful in early phases of hardware architecture design, where the developers have to choose the proper nodes from the possible types that usually is an intuitive, cost-driven process. The final, fine-grained architecture design should be done using the conventional techniques.

6.4 Synthesis of Time-triggered systems

6.4.1 Introduction

Current trends in the embedded industry show that there is an ongoing shift from federated towards integrated architectures for embedded systems. A new, integrated paradigm [HKS04b, DEC] is applied in these systems aiming at compact designs of a reduced number of (networked) ECUs by sharing computing nodes between applications at different safety integrity levels without corrupting safety. The preservation of safety is guaranteed by the principles of replication and robust partitioning. Safety-critical jobs are replicated to multiple nodes, thus a node fault cannot corrupt overall system safety. Robust partitioning by strict spatial and temporal isolation of different application tasks excludes error propagation from non-critical tasks to critical ones, thus prevents the corruption of SC tasks by faults in the non-critical ones.

The integrated architecture is designed for certification, as it relies on deterministic TT protocols for communication, and static computing job execution schedules. The resulting temporal predictability allows for an exhaustive assessment during verification and validation.

The novel mixed-integer linear programming based [NW88] integrated scheduling and allocation method introduced in this section generates a valid allocation of software components to hardware nodes (ECUs), and intranode job and internode communication schedules, that also fulfills extra-functional requirements like dependability, cost, and power consumption.

6.4.2 Related work

Several commercial tools support allocation and scheduling over the TT principle. TTTech [AG] offers TTP-Plan for TTP/C, furthermore, they also offer TTP-Build for job scheduling. These tools generate a single solution based on heuristics, moreover, they need user interaction in several steps of the scheduling.

Mentor Graphics also offers a network design tool (called Volcano Network Architect) [Grab] able to synthesize the communication architecture for in-vehicle networks. However, it only supports (event-triggered) CAN (Controller Area Network) and (time-triggered) LIN (Local Interconnect Network) protocols. As CAN (treated as the core communication protocol) is event-triggered, no scheduling is performed. The tool analyses only the message latencies based on the message priorities and compares them to the timing requirements.

Research results in this area complement the industrial tools (e.g. [PP99], [LA98], [SW00]) by presenting different scheduling methods for a variety of communication protocols, but they consider only the functional and timing requirements of the system.

Shariful et al. [SIH05] describe a so-called multi-variable optimization method for mapping of application components onto hardware nodes. This solution is similar to the one presented in this paper, as it handles both functional, timing, and dependability related requirements. This approach is capable of handling large system models and multiple additional constraints.
All the above mentioned approaches have the advantage of successfully addressing an extremely large number of tasks to be allocated and scheduled by using heuristics during the design space exploration. However, a single solution is typically insufficient for the designer, as he needs a clear overview on the design space and the mutual impacts of technical goals prior of selecting a particular objective function in the terms of mathematics.

Pareto optimization is an interesting approach that combines efficient computation with optimization. Basically, a solution is Pareto optimal if there exists no feasible other solution, which would decrease some objective without causing a simultaneous increase in at least one other objective [Wik]. Applying the basic principle to schedule generation results in a set of Pareto optimal solutions that are presented to the user.

An allocation and scheduling method based on the Pareto optimization principle is presented in [JKH05]. The method uses two factors, cost and dependability, for optimization. The results of the experiment the authors have done shows that the outcome of the algorithm is a set of possible implementation alternatives and the system developer has to select the most appropriate for the actual design problem. The ranking of solutions is done manually, outside of the proposed methodology.

Our method relies on global mathematical optimization, what means usually a high or even unfeasible computational complexity due to the need of traversing an entire candidate solution space. However, a proper combination of methods starting with a fast first phase delivering a feasible, but typically unoptimal solution can drastically confine the solution space for the successive global optimization. The core idea is to limit the search space to be traversed by the computationally costly global optimization algorithm to better solution candidates, then the initial one.

6.4.3 Problem model

We use a simplified PIM and platform model for the description of the allocation and scheduling.

Application components

A distributed application consist of a set of communicating software modules (called jobs) from the software point of view. Jobs have several mandatory attributes in the scheduling model, like period, worst-case execution time (WCET).

Safety of the entire application is ensured by the simultaneous use of robust partitioning and replication. The replicas of safety-critical jobs have to be allocated on different nodes in order to avoid for instance multiple correlated job failures originating in a single node fault. Replicas of a safety-critical job must be managed during resource allocation and scheduling as members of a replica set instead of treating them as independent instances. Inter-job messages are characterized by their period and transmission time.

Hardware elements

The hardware model contains computing nodes with several properties (computing power, memory size, peripherals) and communication channels between the nodes.

Pre-mapping of components

The automatic allocation feature in our method does not enforce the designer to a complete automation. This way the initial model may incorporate external constraints on the allocation originating for instance in a designer decision or from some physical modeling and design tool. (for instance, existence of a specific peripheral, the physical location of the node in the vehicle, etc.).

A similar so-called compatibility matrix supports the manual pre-allocation, as in all similar allocation approaches. This matrix describes the type compliance between jobs and nodes. This relation expresses the matching of some special features of hardware units and requirements by a particular job (e.g. sensors offered and needed), or other additional allocation constraints introduced by the designer.

6.4.4 The scheduling process

The allocation and scheduling process uses the model introduced earlier in order to calculate allocation and schedule for the cluster.
Input data

The allocation and scheduling algorithm has the following inputs:

System topology:
- System topology composed of the \(\{n_1, n_2, \ldots, n_m\}\) set of nodes;
- the jobs: \(\{j_1, j_2, \ldots, j_k\}\) and their \(e_1, e_2, \ldots, e_k\) respective worst case execution times.
- A job compatibility matrix with elements \(b_{ij} = 1\) iff job \(j_i\) can be deployed onto node \(n_j\), thus \(n_j\) offers all the resource types and capacities needed to run \(j_i\).
- Replica sets group the replicas of a given task. The function \(\text{sibling}(i, k) = \text{true}\), iff \(j_i\) and \(j_k\) are replicas of the same task.

Communication topology:
- The messages: \(m_1, m_2, \ldots, m_l\) and their respective transmission times: \(s_1, s_2, \ldots, s_l\) .
- The periods \(p_{j1}, \ldots, p_{jk}, p_{m1}, \ldots, p_{ml}\) of the jobs and messages.
- A job to message bipartite correlation graph \((\text{COMM})\) indicating input and output messages to/from a job. The nodes of the graph are the messages and jobs, and the (directed) edges are the send \((j_i, m_j)\) and receive \((m_i, j_j)\) relations between jobs and messages.

Specific schedule characteristics

The main period of the cluster is called cluster cycle \((\text{CC})\). A CC has to contain for each job and message an integer number of occurrences or \text{instances} in order to have a pure cyclic behavior at the job and message level, as well. As both jobs and messages have their respective periods as independent integers, the simplest solution is using the least common multiplicant of all job and message periods for creating a proper cluster schedule \(\text{CC} = \text{lcm}(p_{j1}, \ldots, p_{jk}, p_{m1}, \ldots, p_{ml})\). The number of instances of a given job (or message) of period \(p_x\) during a cluster cycle becomes \(\text{CC}/p_x = z_x\).

As the implementation environment requires a static structure, our method pre-calculates the cluster cycle, and unrolls the communication topology graph in order to contain each message and job \text{instance} according their respective number of instances. The unrolled graph includes precedence relations between jobs and messages if the job sends the message, and vice versa, between messages and jobs if the job receives the message. The binding of jobs to messages in the graph is a complex problem, as the number of sender, receiver and message instances can vary. A customizable strategy will be introduced to overcome this problem.

The algorithm

The objective of the algorithm is the allocation of jobs to nodes, and the scheduling of jobs and communication. Allocation is a set of job to node mappings associating each job replica with exactly a single computing node under the constraint of robust partitioning. The communication schedule contains the transmission initiation times of messages within of the CC for all messages, the job schedule contains the start time(s) of the job replicas.

1. The \(\text{CC} = \text{lcm}(p_{j1}, \ldots, p_{jk}, p_{m1}, \ldots, p_{ml})\) and the numbers of job and message instances per CC \((z_{ji} = \text{CC}/p_{ji}\) and \(z_{mi} = \text{CC}/p_{mi}\) are estimated first.

2. Graph unrolling generates an already “instantiated” message-job sequence diagram from the basic \(\text{COMM}\) graph that contains \(z_{ji}\) and \(z_{mi}\) instances of each job and message type that are scheduled in a CC.

Depending on the number of messages \((m)\), sender \((s)\) and recipient \((r)\) instances nine cases of their value ordering are possible as three kinds of relations \((<, >, =)\) are possible between the respective variable values. \(^1\)

Two embedded heuristics guide the recent version of unrolling:

\(^1\)In the current TT protocols several additional restrictive constraints may apply for \(m, s\) and \(r\). In most cases, a job can send only at most a single message instance per execution \((m \leq s)\). In case of other protocols, the sender period must equal to the message period \((m = s)\).
(a) Messages should be sent ASAP, in order to avoid late processing or superfluous waiting times in the receiver node due to missing information.

(b) If there are more job instances, than messages to be received by this type of jobs, always the latest job instances receive the corresponding messages in order to loosen the scheduling constraints for the first few jobs.

Note, that these heuristics can be customized by simply exchanging the related part of the algorithm without any major effect on the principle of the algorithm.

The message-job binding in Fig. 6.3 illustrates a simplified case of \((s < m \land s < r)\), the number of messages is less than the number of sender and receiver jobs) confined to the case of the normal precedence relation (i.e. when a job or message must terminate before another starts). Naturally, the complete precedence graph covers all the different job and message instance ordering relations.

3. Allocation and schedule calculation

Scheduling of the job-message sequence is performed by transforming it to a MILP problem.

Let us denote the start and termination time of the \(j\)th instance of job \(i\) by \(jstart_{i,j}\) and \(jend_{i,j} = jstart_{i,j} + e_i\). As the job instances follow each other by the predefined period \(p_i\): the message scheduling equation becomes to \(\forall i, j \in (1 \ldots jobinst_i - 1) : jstart_{i,j+1} = jstart_{i,j} + p_i\), where \(jobinst_i\) denotes the total number of instances of job \(i\).

Similarly, let denote by \(msginst_i\) \(i \in (1..m)\) the number of message instances. For each message instance \(msgstart_{i,j}\) and \(msgend_{i,j} = msgstart_{i,j} + s_i\) denote the start and termination time of the \(j\)th instance of the \(i\)th message. As the message instances follow each other by the predefined period: \(\forall i, j \in (1 \ldots msginst_i - 1) : msgstart_{i,j+1} = msgstart_{i,j} + p_i\) Message transmission should not overlap in the single channel communication network, thus \(\forall i_1, i_2, j_1, j_2 : (i_1 \neq i_2) \lor (j_1 \neq j_2) \rightarrow (msgend_{i_2,j_2} \leq msgstart_{i_1,j_1}) \lor (msgstart_{i_1,j_1} \leq msgend_{i_2,j_2})\)
6.4. SYNTHESIS OF TIME-TRIGGERED SYSTEMS

The Boolean array \( \text{uses}(i, j) \) represents the actual resource usage relationship between job instances and nodes.

\[
\text{uses}(i, j) = \begin{cases} 
  \text{true} & \text{if job}_i \text{ is running on node}_j, \\
  \text{false} & \text{else.}
\end{cases}
\]

Several rules apply to job-node relations:

- All jobs have an associated node: \( \forall i \exists j: \text{uses}(i, j) \)
- All jobs run on a compatible node: \( \text{uses}(i, j) \rightarrow b_{ij} = 1 \)
- Each job is associated with a single node. \( \forall i, j_1, j_2: \text{uses}(i, j_1) \land \text{uses}(i, j_2) \rightarrow j_1 = j_2 \)
- If two jobs are using the same node, none of their instances can overlap in time: \( \forall i_1, i_2, k: (i_1 \neq i_2) \land \text{uses}(i_1, k) \land \text{uses}(i_2, k) \rightarrow (\forall j_1, j_2: (j_1 \neq j_2) \land ((\text{end}_{i_1, j_1} \leq \text{start}_{i_2, j_2}) \\
\lor (\text{end}_{i_2, j_2} \leq \text{start}_{i_1, j_1})) \)
- Two replicas of the same task must not run on the same node: \( \forall i_1, i_2, j_1, j_2: (i_1 \neq i_2) \land \text{uses}(i_1, j_1) \land \text{uses}(i_2, j_2) \land (\text{sibling}(i_1, i_2)) \rightarrow (j_1 \neq j_2) \)

Additional constraints are defined in order to guarantee the validity of the schedule. They assure that all job and message instances are processed during a single CC.

- All job instances start in the CC: \( \forall i, j: (0 \leq \text{start}_{i, j} \leq \text{CC}) \)
- All job instances terminate in the CC: \( \forall i, j: (0 \leq \text{end}_{i, j} \leq \text{CC}) \)
- All message instance transmission start in the CC: \( \forall i, j: (0 \leq \text{msgstart}_{i, j} \leq \text{CC}) \)
- All message instance transmission terminate in the CC: \( \forall i, j: (0 \leq \text{msgend}_{i, j} \leq \text{CC}) \)

Each individual feasible solution defines a valid allocation and schedule. The technical goal of the optimization needs a properly formulated objective function. Multiple criteria characterise an embedded system design of a good quality. This way some of them may be taken as hard constraints, some of them may be included into the objective function depending on the application context.

In the subsequent part we give a formulation of the most typical requirements in the context of TT applications. The most typical requirement in the temporal domain is the end-to-end delay of the control system. End-to-end delay is the time that passes between a value change at the sensor input(s) to the reaction of the system observable as changes at the setpoints of the actuator output(s). This value consists of the job execution times, message delays, and sensor/actuator delays. The scheduling algorithm has influence only on the message delays. The total delay \( (T_D) \) of a message contains the waiting time in the buffer of the sender \( (T_{ws}) \), the transmission time \( (T_s) \), and the waiting time in the buffer of the receiver \( (T_{wr}) \). Jobs in typical TT systems require incoming messages at startup in order to avoid asynchronous races. Based on this assumption, the waiting times can be calculated as follows:

- The delay of a message instance in the sender buffer is the minimum interval between the end of the sender job and transmission time of the message (modulo cluster cycle length). \( \forall i, l, n : ((\text{msgstart}_i) \in \text{COMM}) \land n \in [1..\text{msginst}_i] : T_{ws, j, l, n} = \min_k (\text{msgstart}_{i, n} - \text{end}_{i, k} \mod \text{CC}) \)
- The delay of a message instance in the receiver buffer is the minimum interval between the end of the message transmission and the starting time point of the job (modulo cluster cycle length). \( \forall i, l, n : ((\text{msgstart}_i) \in \text{COMM}) \land n \in [1..\text{msginst}_i] : T_{wr, j, l, n} = \min_k (\text{start}_{i, k} - \text{msgend}_{i, n} \mod \text{CC}) \)
- The worst-case timing for message delay of message 1 is the maximum of the possible delays. \( T_{D, 1} = \max_{j, k, n, o}(T_{ws, j, l, o} + T_{wr, k, l, o}) + T_n \)

The possible sensor-actuator paths can be calculated based on these expressions and the structure of the application (job-message relations). A sensor is a source in the COMM graph, and an actuator is a sink. The calculation results in a set of feasible paths, each having a total message delay including all message delays along the source-sink path.

Different sensor-actuator interactions may be attributed in a practical application by different temporal criticality. This can be reflected by assembling the objective function as a weighted sum of communication delays, where weights are calculated in order to express the relative importance of the individual paths.
6.4.5 Non-functional properties

The basic form of the algorithm as presented in the previous section deals only with the functional and timing requirements of the system. There are, however, other aspects (like dependability, cost, power consumption, etc.) that should be considered during the design workflow. The inclusion of these aspects necessitates the formalization of the quantitative attributes and a subsequent inclusion either as hard constraints or as (parts of) the objective functions. We will introduce two examples of non-functional aspects in the following section, a simpler and a more complex one.

Hardware cost

Cost factor plays an important role in several application domains aiming at mass production. In this cases, a priority goal is the minimization of hardware costs, by using only the minimum number of computing nodes or in a more advanced form to reduce the total hardware cost. The calculation of the total cost demands the introduction of the cost function values $\text{cost}(n_i)$ for hardware nodes $n_i$.

The total cost of the system is:

$$\text{Cost}_{\text{Total}} = \sum_{i: \exists j: \text{uses}(j,i)} \text{cost}(n_i)$$

This expression can be used either as an objective function if the cost has to be minimized, or as a hard constraint ($\text{Cost}_{\text{Total}} \leq \text{Cost}_{\text{Maximum}}$) on the hardware costs.

Availability

Availability is an important factor in most of the critical applications due to its direct impact on the quality-of-service. It is usually defined at the service level, and the availability values of the individual components are irrelevant from the end-user point of view. In this context, a system is available if it performs its basic functionality and possesses enough safety-related resources as prescribed by its specification.

Several methods and tools exist for availability calculation that use either analysis or simulation for this purpose (eg. [IMM07]). However, a support for design for maximal availability integrated into scheduling needs a simplified analysis method in order to cope with the computational complexity. Such a method is presented in Section 6.2, that can be used in the current domain.

Our fault model anticipates independent single node faults with the communication bus as the only shared resource. Hardware nodes have an availability rate either specified by the part supplier, or measured by empirical observations. For simplicity we assume that all application components are design error free (the software is thoroughly tested, and runs through a rigorous V&V process), however, this simplification does not restrict the general validity of the approach.

Several expressions are used to define the availability of services. Most of the following formulae can be evaluated before of the execution of optimization in order to achieve better runtime performance.

- For each hardware node $i$, the availability is $a_{hw,i}$.
- Different services may be characterized by roughly different availability requirements, as the purpose of integrated systems is the cost-efficient implementation of both critical and non-critical functionalities. For instance, a comfort service has a low value of availability, while a safety-critical functionality demands a high value. For each service, the user defines a minimum availability requirement $A_{req}$.
- The set of necessary components for the excution of a particular job is defined based on the mutual inter-dependencies of the software components: $\text{ Req}_j$. This step can be done before the scheduling starts by extracting it from the description of the target system, as functional dependency is static and independent of the actual deployment. An application component requires an other, if it receives message from it. Each replicated job is a variation point in the $\text{ Req}$ set. The requirement in replicated case is at least one replica, because the platform guarantees fail-silent behavior. If a job has $n$ replicas, $n$ variants (branches) of the set $\text{ Req}$ are created in order to facilitate the calculation runtime.
- The runtime constraint uses the above expression in order to define the availability constraint:

$$A_{req} \leq \max_i (\prod_{j: \exists k: \text{uses}(k,i)=1}\{k \in \text{Req}_j\}(a_{hw,i}))$$

Using this simplified model, the solver can include availability aspects during allocation and scheduling.

**Proposition 30** The job and message related constraints discussed in the previous section, together with one of the functions (cost and availability) form an optimization problem. The solution of the problem provides the optimal (from the cost or availability point of view) solution for the allocation and scheduling problem.
Proof The proof is straightforward, as the constraints that define the set of valid solutions are reused by the optimization problem, and the cost function as objective function ensures the cost-optimal solution.

Allocation and scheduling algorithms can be extended with other non-functional constraints, like power consumption, space requirements, as well.

6.4.6 Multi-aspect optimization

Several different objective functions can be defined based on the various non-functional aspects, and any of these can be used for optimization. In the DECOS project [3,15] we selected five objectives for experiments in order to discover a typical design space (see Fig. 6.4). The basic objectives were the highest throughput (lowest end-to-end delay), and total hardware cost of the system. Extensibility (by integrating new functionalities on the same platform) is inspected using two related objectives: the maximization of the available CPU time on one node, and the equalization of free CPU time on all of the system nodes.

The last objective used was robustness, a simplified representative of the availability measure. The robustness value is the number of properly functioning applications in case of a given number of hardware node faults. Robust allocations do not allow the simultaneous fall-out of multiple critical applications even in case of faults.

As shown in Fig. 6.4, the experiments were rated regarding to 5 aspects. Each are scaled that way, that the optimal value is on the margin, while the worst value is in the center of the diagram. The experiments have shown that using a simple objective function is not always reasonable, as it will probably find the result on the perimeter of the design space.

Two solutions can be used to avoid this effect. The first is the combination of objective functions and additional constraints, where a selected basic function is the objective, and constraints are set for the value of other functions. For instance, the optimal throughput is measured in an initial run, and then it is used as a constraint during cost optimization (throughput must be less than the optimal $+30\%$).

The second solution is the weighted average function of the basic ones, very similar to the balanced scorecards [RSK96] used in several domains. This results in a multi-aspect optimization reflecting the priorities set by the user as weight factors.

6.4.7 Evaluation of the method

The first evaluation of the allocation and scheduling method has been done in the framework of the DECOS project using the automotive demonstrator applications and platform developed in the project [15]. ILOG OPL Studio [ILO] has been used as solver tool for the experiments. The results have shown that there is a high variance in the execution time of the solver depending on the objective function. There are optimization aspects that are not affected by the actual schedule, only by the allocation (cost, robustness, availability), and others that are depending on the schedule (throughput, end-to-end delay). Specifying a combination of the first group of aspects as objective results in a shorter solving time (10 seconds to 1 minute in case of the
DECOS models), as the solver optimizes only the allocation-related variables. The optimization according to the second group of aspects, however, can result in an execution time in the range of 5–10 minutes.

The performance bottleneck is caused by the fact that the problem that is solved by a MILP solver contains mostly integer variables that results in an integer programming problem that cannot be handled efficiently. The unacceptably long execution time even in the case of relatively simple input problems necessitated the revising of the solver tool selection. Further research and development has been done [Rác08] and a new, constraint programming based version of the scheduling model has been developed. The performance of the new implementation that uses the Prolog finite domain constraint solver has been evaluated using large models generated based on typical messaging patterns (for details see [Rác08]). The performance is found sufficient for industrial grade implementation (see Table 6.1).

<table>
<thead>
<tr>
<th>Number of jobs</th>
<th>No. of job inst.</th>
<th>No. of message inst.</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>274</td>
<td>234</td>
<td>319.00</td>
</tr>
<tr>
<td>200</td>
<td>606</td>
<td>484</td>
<td>715.33</td>
</tr>
<tr>
<td>300</td>
<td>843</td>
<td>738</td>
<td>1010.66</td>
</tr>
<tr>
<td>400</td>
<td>116</td>
<td>938</td>
<td>1381.00</td>
</tr>
<tr>
<td>500</td>
<td>1350</td>
<td>1170</td>
<td>1572.66</td>
</tr>
<tr>
<td>600</td>
<td>1722</td>
<td>1446</td>
<td>2151.66</td>
</tr>
<tr>
<td>700</td>
<td>1969</td>
<td>1705</td>
<td>2416.66</td>
</tr>
<tr>
<td>800</td>
<td>2323</td>
<td>1970</td>
<td>2854.00</td>
</tr>
<tr>
<td>900</td>
<td>2654</td>
<td>2248</td>
<td>3312.33</td>
</tr>
<tr>
<td>1000</td>
<td>2779</td>
<td>2408</td>
<td>3307.66</td>
</tr>
<tr>
<td>2000</td>
<td>5831</td>
<td>5013</td>
<td>7198.00</td>
</tr>
</tbody>
</table>

### 6.4.8 Conclusions

We presented an allocation and scheduling method for distributed time-triggered systems in this section. Our solution supports the simultaneous handling of functional, timeliness, and additional design constraints in a modular, extensible way. This results in a more effective design process, as there is no need for multi-step semi-automatic allocation, scheduling, and constraint analysis. Our method is communication protocol independent, but can be customized to any time-triggered protocols. We have used a third party commercial solver tool in order to be able to leverage the performance of built in heuristics and optimization methods.

### 6.5 Synthesis of event-triggered systems

Although the current Chapter focuses on the analysis and synthesis of time-triggered systems, we will briefly introduce some methods and techniques that are applied for the synthesis of event triggered (ET) systems. ET systems are widely used in both embedded, desktop and enterprise IT systems. The application of these systems in safety-critical real-time domains necessitates the introduction of different design methods that assure temporal predictability of the systems. ET systems, by nature, have dynamic behaviour. Their execution is not planned statically at design time, but is the result of the interaction of the system and its environment.

System predictability in ET systems is usually ensured by task and message priorities that define an ordering between the tasks and messages of the system. The ordering defines the execution/transmission order of concurrent elements, meaning for example that the task with the highest priority will be started if there are multiple tasks ready to run on a single processor. Priority assignment, however, can lead to the starvation of lower priority tasks and messages consuming all available resources. In order to avoid this effect, typical event-triggered systems introduce the concept of debounce time, the minimal time between two consecutive execution/transmission of an entity. Debounce time should be guaranteed by the execution environment at runtime, even in the presence of errors (for instance, a babbling idiot task or node). The introduction of debounce time results in a repeating interval for all tasks and messages, that is analogue to the period of TT entities, but instead of a single numeric value, it can be represented by a finite interval. The main challenge of analysis and synthesis tools is that they should ensure that the temporal requirements of the system will be fulfilled by all possible executions fulfilling the element repeating and processing intervals.
Several different approaches are used for scheduling the execution of tasks in ET systems. Rate and deadline monotonic algorithms [DAA90] are widely used in hard real-time systems. The algorithms suppose that all tasks have periods or debounce times, deadlines that is less than the repetition time, and worst case execution times. There is a dedicated point in time, when all tasks become ready for execution, and once a task is executed it will not become ready again. It has been shown [LL89] that this scheduling approach is optimal, e.g. if a static scheduling algorithm finds the problem schedulable, this algorithm will produce a valid schedule.

Priority assignment is also a fundamental task in ET systems design. Different approaches targeting this problem have been published, but in general, all have similar structure. The methods create a priority assignment, validate it, and if necessary, optimize it. A static priority assignment and scheduling approach is presented in [EPCE07] focusing on simple, uniprocessor embedded systems. The authors use simulated annealing to optimize the priority assignment. A priority assignment method for FlexRay static and dynamic segment is presented in [PEP02] that synthesizes a priority assignment calculation based on the communication and on the ET task execution.

There are also industrial framework consisting of a design environment and runtime platform (such as Volcano System [Grab]) that implement priority assignment, allocation, and scheduling for ET systems that guarantee predictable behaviour in the temporal domain. It should be noted that not all of these are fault-tolerant meaning that the presence of fault influences the behaviour of the system.

6.6 Applications

6.6.1 Early analysis of system models

We used the high-level availability analysis method in the DECOS project to give instant, integration-time feedback on possible design problems to the system designer. The implementation of the analysis tool was part of the DECOS PIM-PSM mapping tool [17].

The approach has been reused in the German-Hungarian bilateral project BelAMI that aimed at the model-driven design of ambient intelligent systems. Our approach proved to be applicable also for the mobile, peer-to-peer environment. [4,19]

6.6.2 Deployment optimization

The cost-driven optimization method introduced in the current chapter has been used in the enterprise applications domain in order to optimize the deployment of multi-tier component-based enterprise applications [1,6].

The application domains (component-based enterprise and embedded systems) have several commonalities that can be utilized during the reuse of synthesis concepts and techniques. Table 6.2 summarizes the concepts that can be matched. Applications, components, and computing nodes can easily be matched. Component timing that consists of period and WCET in case of TT embedded systems can be matched with the work load (requests served by the component). The allocation optimization uses this factor and the performance limitation of the computing node (determined by benchmarks [6]) for the calculation.

Strong partitioning and static communication allocation is matching the virtualization and network bandwidth management facilities of enterprise computing platforms. It should be noted that some server operating systems offer different levels of virtualization (hardware partitioning, OS level virtualization, application level virtualization) that enables even more sophisticated configuration than in case of simpler embedded operating systems.

In the case of this new application domain, the timing properties have been changed from worst-case to average, but the basic principles were successfully reused. This has also proved that although there are many differences between the embedded systems and enterprise systems domains, their common properties like component orientation, distributed deployment allow for the cross-domain reuse of analysis techniques.

The utilization of the optimization method is also foreseen in the INDEXYS European research project, that aims at the development of model-driven analysis and synthesis methods and tools for distributed embedded systems. Our method will be used for the high-level design of target hardware architecture.

6.6.3 Allocation and scheduling of time-triggered systems

The results on time-triggered allocation and scheduling of time-triggered systems have been utilized in the DECOS project [3,15,18]. We ported the generic method to the TTP/C protocol that has been used in the project and implemented an allocation and scheduling tool that complemented the existing DECOS
Table 6.2: Matching enterprise and embedded concepts

<table>
<thead>
<tr>
<th>Embedded systems</th>
<th>Enterprise systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw component</td>
<td>sw component, web service</td>
</tr>
<tr>
<td>application</td>
<td>application (web, multi-tier)</td>
</tr>
<tr>
<td>message</td>
<td>inter-process communication (remoting)</td>
</tr>
<tr>
<td>linking interface</td>
<td>service interface</td>
</tr>
<tr>
<td>ECU</td>
<td>server node</td>
</tr>
<tr>
<td>component period, WCET</td>
<td>component load (requests/sec)</td>
</tr>
<tr>
<td>component replication</td>
<td>component failover clustering</td>
</tr>
<tr>
<td>hardware replication</td>
<td>server clustering</td>
</tr>
<tr>
<td>Partitioning RTOS</td>
<td>virtualization</td>
</tr>
<tr>
<td>allocation</td>
<td>allocation</td>
</tr>
<tr>
<td>message scheduling</td>
<td>bandwidth management</td>
</tr>
</tbody>
</table>

tool chain. The tool scheduling concept has been further improved in the GENESYS project aiming at the definition of general embedded systems architecture and development methods.

There is an ongoing industrial project where OptXware Ltd. is a subcontractor of one of the leading automotive tool vendors, and we are responsible for the development of a FlexRay scheduling tool for AutoSAR-based automotive systems. The reuse of the basic methodology has proved that it is portable through several different communication protocols and modeling standards, and applicable for real-life, industrial problem sizes. The FlexRay scheduling capability will also be used in the INDEXYS project, with a strong emphasis on multi-aspect optimization.

6.7 Summary

We have introduced several analysis and synthesis methods in the current chapter that support the development of embedded systems. High level availability analysis is utilized in hardware-software integration tools and provides instant feedback to the designers on predicted availability.

Deployment optimization and architecture synthesis supports the system designer in finding the cost-optimal target architecture for the system under design. We have successfully applied this technique also in a different domain (enterprise information systems).

Allocation and scheduling for time-triggered systems has been utilized in several projects and complements state-of-the art heuristics-based techniques with a multi-aspect optimization-based solution. By the support of design space discovery, the designers are able to make decisions on the goal of the optimization in terms of several different non-functional aspects.

Contribution 3 I have defined analysis and synthesis methods supporting the model-based development of embedded real-time dependable systems. In particular, the following novel procedures were elaborated:

3.1 I have defined a high-level availability analysis method that estimates the availability properties of the system under design at early stages of the development process. [4, 5]

3.2 I have developed a cost-driven method for the high level platform architecture optimization in distributed systems based on the required performance and availability attributes. [1, 6]

3.3 I have developed a multi aspect optimization-based approach for the allocation and scheduling of distributed embedded time-triggered systems. [3, 15, 18]
Chapter 7

Conclusions

As a final conclusion, I compare the results presented in the current thesis with the main research objectives that have been defined in Section 1.4. Additionally, I summarize the practical applications of the results and outline some future research and application directions.

7.1 Fulfillment of Research Objectives

Objective 1: Improve the productivity of systems development by introducing interactive, semi-automated tools and methods in different phases of the development process.

I proposed an interactive, iterative approach for hardware-software integration during the system development that extends the traditional Model-driven Architecture (MDA) approach and combines the automatic execution of repetitive tasks with the manual or semi-automatic steps that allow the designer to take control of key design decisions.

Objective 2: Improve the productivity of systems development by introducing interactive, semi-automated tools and methods in different phases of the development process.

The proposed MDA-based approach clearly separates the architectural software design, the behavioral modeling and implementation, and the platform design aspects during the system design, and supports their integration during the hardware-software integration phase.

Objective 3: Improve the quality of system designs by the introduction of different consistency checks, analysis, and synthesis methods.

I proposed an open, extensible constraint checking framework that is integrated with the hardware-software integration framework and gives early feedback on potential design problems online. I also applied high-level availability analysis methods to the PIM and PSM models used during system design. In order to further improve the quality of design I also proposed a high-level architecture and deployment optimization approach, and a detailed task and communication scheduling solution. These synthesis methods rely on mathematical optimization and are customizable with respect to their objective function.

Objective 4: Define a model-driven hardware-software integration methodology that is capable of handling complex target architectures and several non-functional aspects like timeliness, dependability, and cost.

The proposed hardware-software integration methodology and framework has been investigated in the domain of safety-critical time-triggered embedded systems and its application has been successful in the automotive, aerospace, and industrial control domains. The analysis and synthesis tools that complement the framework are also capable of handling dependability, timeliness, and cost related parameters, and their architecture is open to the introduction of new non-functional aspects.

Objective 5: Provide advanced language constructs that support the reusability of model transformations.

I introduced graph pattern and transformation rule level constructs (or-patterns, pattern calls, and recursion) that improve the re-usability of patterns and rules, decrease the size, and improve the testability of transformation programs. I also introduced a graph transformation cascading technique that enables the reuse of specific tool components (model importers and exporters) and reduce the runtime and memory requirements of model import/export operations.

Objective 6: Provide support for the executable specification of design patterns that allow the integration of best practices and common solution patterns into domain-specific modeling environments.

I introduced an executable specification formalism for design patterns based on graph transformations, furthermore, I also defined a uniform trace metamodel in order to be able to explicitly trace design pattern applications.
CHAPTER 7. CONCLUSIONS

**Objective 7: Provide support for the integration of different metamodeling environments using a precise semantical foundation.**

I introduced a mapping of ECore and MOF models to the precise VPM framework that enables the integration of modeling languages defined in one of these environments. I also defined a formal operational semantics for ECore that enables not only the model exchange between the environments, but also the live integration of them, and the systematic test generation and correctness proving in the ECore environment.

7.2 Utilization of the new results

The theoretical results presented in the current thesis have been utilized in several national and international research and industrial projects. The tools that have been developed include the work of several students doing their MSc or BSc thesis, as well as the efforts of my colleagues at OptXware Research and Development Ltd.

**The Viatra2 model transformation framework**

The model transformation technology related results that have been presented in the current thesis are implemented in the open-source Viatra2 model transformation system that is extensively used in many different research projects. The advanced model transformation language constructs introduced in the current thesis have been integrated in the VTCL language of the framework, and the ECore mapping and live model sharing solutions are also being integrated.

**Hardware-software integration framework for embedded systems**

The interactive hardware-software integration approach presented in the current thesis has been utilized in different research projects. It serves as a basis for the DECONS and DIANA tools, and its application is foreseen in the INDEXYS project. Integrated with the workflow-based transformation integration approach, it is currently at its third generation, enabling the rapid prototyping of complex, interactive mapping tools.

The analysis and synthesis methods are also integrated with the mapping tool framework in order to introduce mathematical methods during system design.

**Scheduling of time-triggered embedded systems**

The scheduling method proposed in the current thesis is used for the scheduling of FlexRay networks in an industrial project that aims at the automatic generation of network schedules for automotive applications.

**Ongoing development and future work**

An ongoing activity is the development of a new generation of model integration technology based on Viatra2 and the results presented in this thesis, extended with technologies that allow the handling of large model spaces and team collaboration during modeling. The concept of the solution has already been defined in the GENESYS project and will be implemented in INDEXYS.

The hardware-software integration framework will further improved in order to allow the customization of the integration process also by the users of the framework that will result in a more customizable tool. Generative techniques will be used to synthesis most of the tool implementation to further reduce the efforts needed for tool development.

The scheduling approach introduced in this thesis will be specialized to meet the custom requirements of the different potential target domains (automotive, railway, aerospace), and to include target network protocol specific constraints and assumptions.

7.3 Open problems

There are several open problems regarding the topic of the dissertation that can serve as objectives for future research.

The reusability of model transformations can be further improved by generalizing the transformation cascading algorithm in order to be able to use it on a more generic class of graph transformations. If two arbitrary transformations could be cascaded, the runtime and space usage of long transformation chains could be further reduced.

We defined a formal semantics for ECore that can be a basis of different systematic test generation and execution methods that could verify and validate the behavior of model manipulation and transformation tools. This area is an open, unsolved research topic as today transformations are usually verified and validated manually, using ad hoc test suites. It should be noted that the certification of modeling tools is also a demanding open issue as the usage of the tools in safety-critical domains (e.g. aerospace) requires the certification of the tools.
The iterative, interactive hardware-software integration framework introduced here proved to be suitable for dependable embedded systems, but its adaptation to dynamic application domains (as ad hoc mobile networks) needs further research. A new, emerging domain is the network-on-a-chip (NoC) and system-on-a-chip (SoC) technology that will also require new approaches to hardware-software integration.

The allocation and scheduling solution introduced in the current thesis considers a single network cluster, but in a typical automotive application there can be up to ten different networks. The state-of-the-art techniques today are not able to perform global allocation and scheduling that considers all nodes and networks in a single phase and is able to fulfill global timeliness and dependability requirements. The research and development of such tools would highly influence the development productivity and manufacturing costs of such complex systems.
Chapter 8

References

8.1 References to external publications


IEEE. Dependable components and systems. an eu framework 6 integrated project. http://www.decos.at/.


Helm Richard Johnsohn-Ralph Gamma, Erich and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Systems. Addison Wesley, 1995.


IEEE. 802.3-2008 ieee standard for information technologyspecific requirements - part 3: Carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications, 2008.


8.2 References to own publications

Book part


Journal papers


International conference and workshop papers


**Local conference and workshop papers**


**Invited talks, technical reports**


## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARTEMIS</td>
<td>Advanced Research &amp; Technology for EMbedded Intelligence and Systems.</td>
</tr>
<tr>
<td>ASM</td>
<td>Abstract State Machines.</td>
</tr>
<tr>
<td>CBSE</td>
<td>Component-based Software Engineering.</td>
</tr>
<tr>
<td>CC</td>
<td>cluster cycle.</td>
</tr>
<tr>
<td>CRD</td>
<td>Cluster Resource Description.</td>
</tr>
<tr>
<td>DAS</td>
<td>Distributed Application Subsystem.</td>
</tr>
<tr>
<td>DSE</td>
<td>Domain Specific Editor.</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-specific language.</td>
</tr>
<tr>
<td>DSM</td>
<td>Domain-Specific Modeling.</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit.</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise Java Beans.</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework.</td>
</tr>
<tr>
<td>EMOF</td>
<td>Essential MOF.</td>
</tr>
<tr>
<td>ES</td>
<td>Embedded system.</td>
</tr>
<tr>
<td>ET</td>
<td>event-triggered.</td>
</tr>
<tr>
<td>GME</td>
<td>Generic Modeling Environment.</td>
</tr>
<tr>
<td>GoF</td>
<td>Gang of Four.</td>
</tr>
<tr>
<td>GT</td>
<td>graph transformation.</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property.</td>
</tr>
<tr>
<td>LF</td>
<td>Local Interface.</td>
</tr>
<tr>
<td>LHS</td>
<td>left-hand side.</td>
</tr>
<tr>
<td>LIF</td>
<td>Linking Interface.</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-Driven Architecture.</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-Driven Development.</td>
</tr>
<tr>
<td>MIC</td>
<td>Model-Integrated Computing.</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Facility.</td>
</tr>
<tr>
<td>MT</td>
<td>model transformation.</td>
</tr>
<tr>
<td>MVO</td>
<td>Multi-Variable Optimization.</td>
</tr>
<tr>
<td>NFP</td>
<td>non-functional property.</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language.</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group.</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language.</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform-independent Model.</td>
</tr>
<tr>
<td>PIT</td>
<td>Platform-independent Transformer.</td>
</tr>
<tr>
<td>PM</td>
<td>Platform Model.</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform-specific Model.</td>
</tr>
<tr>
<td>PST</td>
<td>Platform-specific Transformer.</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality-of-Service.</td>
</tr>
<tr>
<td>QVT</td>
<td>Queries, Views, and Transformations.</td>
</tr>
<tr>
<td>RHS</td>
<td>right-hand side.</td>
</tr>
<tr>
<td>SAM</td>
<td>simplified platform-independent architecture model.</td>
</tr>
<tr>
<td>SPM</td>
<td>simplified platform model.</td>
</tr>
<tr>
<td>SW-C</td>
<td>software component.</td>
</tr>
<tr>
<td>SysML</td>
<td>OMG Systems Modeling Language.</td>
</tr>
<tr>
<td>TT</td>
<td>time-triggered.</td>
</tr>
</tbody>
</table>
UML = Unified Modeling Language.
VPM = Visual and Precise Metamodeling.
VTCL = VIATRA2 Textual Command Language.
VTML = VIATRA2 Textual Modeling Language.
XMI = XML Metadata Interchange.
Appendix A

Metamodels from Chapter 5

A.1 NFP Metamodel

A.2 Software Component Architecture Metamodel

class(pim) {
    entity(Application);
    relation(dataTypes , Application , DataType);
    relation(declaredService , Application , Service);
    relation(components , Application , Component);
    relation(main , Application , CompositeComponent);

    entity(DataType);
    relation(refines , DataType , DataType);

    entity(DataElement);
    relation(name , DataElement , datatypes.String);
    relation(type , DataElement , DataType);

    entity(Interface);
    relation(containedElements , Interface , DataElement);

    entity(Component);
    relation(input , Component , Interface);
    relation(output , Component , Interface);

    entity(Service);

    entity(AtomicComponent);
    relation(requiredServices , AtomicComponent , Service);
    supertypeOf(Component , AtomicComponent);

    entity(ComponentInstance);
    relation(type , ComponentInstance , Component);

    entity(CompositeComponent);
    supertypeOf(Component , CompositeComponent);
    relation(components , CompositeComponent , ComponentInstance);
    relation(connectors , CompositeComponent , Connector);

    entity(Connector);
    relation(sourceComponent , Connector , ComponentInstance);
    relation(targetComponent , Connector , ComponentInstance);
    relation(sourceInterface , Connector , Interface);
    relation(targetInterface , Connector , Interface);
A.2.1 NFP metamodel

```java
import datatypes;
entity(nfp) {
    entity(NFP_Type);
    entity(NFP);
    relation(type,NFP,NFP_Type);
    relation(allowedUnits,NFP_Type,Unit);
    relation(defaultUnit,NFP_Type,Unit);
    entity(Unit);
    relation(convFactor,Unit,Double);
    relation(baseUnit,Unit,Unit);
    relation(symbol,Unit,String);
    entity(ValueSpecification);
    relation(defaultValue,NFP,ValueSpecification);
}
```

A.2.2 PIM NFP models

**Dependability**

```java
import datatypes;
import pim;
import dependability;
import nfp;
import timing;
entity(pimdepy) {
    NFP(SIL);
    NFP.type(t,SIL,IntegerType);
    relation(value,SIL,ValueSpecification);
    relation(safetyIntegrityLevel,Application,SIL);
    NFP(Red);
    NFP.type(t,Red,IntegerType);
    relation(value,Red,ValueSpecification);
    relation(redundancyDegree,Application,Red);
    relation(redundancyDegree,Component,Red);
    NFP(Aval);
    NFP.type(t,Aval,DoubleType);
    relation(value,Aval,ValueSpecification);
    relation(availability,Application,Aval);
}
```

**Timing**

```java
import datatypes;
import pim;
import timing;
import nfp;
entity(pimtiming) {
    NFP(WCET);
    NFP.type(t,WCET,DurationType);
```
A.3 Platform Metamodel

```mermaid
import nfp;

entity(platform) {
entity(Component); 
relation(implementedServices, Component, PlatformService);
relation(requiredFeatures, Component, FeatureInstance);
relation(providedFeatures, Component, FeatureInstance);
relation(connectors, Component, Connector);
```
entity (PlatformService);

entity (Connector);
relation (role, Connector, ConnectorRole);
entity (ConnectorInstance);
relation (type, ConnectorInstance, Connector);

entity (ComponentInstance);
relation (type, ComponentInstance, Component);
relation (connectorInstances, ComponentInstance, ConnectorInstance);

entity (CompositeComponent);
relation (components, CompositeComponent, ComponentInstance);
supertypeOf (Component, CompositeComponent);
relation (connection, CompositeComponent, Connection);

entity (Connection);
relation (members, Connection, ConnectorInstance);
entity (DelegateConnection);
supertypeOf (Connection, DelegateConnection);
relation (delegationConnector, DelegateConnection, Connector);

entity (Feature);
relation (characteristics, Feature, NFP);

dentity (FeatureInstance);
relation (feature, FeatureInstance, Feature);
relation (characteristicValues, FeatureConnection, NFP);

entity (FeatureConnection);
relation (featureInstances, FeatureConnection, FeatureInstance);
relation (supplier, FeatureConnection, ComponentInstance);
relation (consumer, FeatureConnection, ComponentInstance);

entity (ConnectorRole) {
    ConnectorRole (LocalConnector);
    ConnectorRole (LinkingConnector);
}
relation (role, Connector, ConnectorRole);

entity (ComponentRole) {
    ComponentRole (ExecutionEnv);
    ComponentRole (CommController);
    ComponentRole (Transceiver);
}
relation (role, Component, ComponentRole);

entity (HWPart);
relation (properties, HWPart, HWProperty);
relation (part, Component, HWPart);
entity (HWProperty);

entity (HWPropertyValue);
relation (property, HWPropertyValue, HWProperty);
relation (value, HWPropertyValue, ValueSpecification);
relation (properties, Component, HWPropertyValue);

A.4 Platform Specific Metamodel
import datatypes;

entity (psm) {
    entity (PSMRoot);
    relation (application, PSMRoot, pim.Application);
    relation (platform, PSMRoot, platform.CompositeComponent);
    relation (dataTypeMaps, PSMRoot, dt.DataType);
    relation (jobCompatibilityMaps, PSMRoot, JobCompatibilityMapping);
    relation (jobAllocations, PSMRoot, JobAllocation);
    relation (nodeSchedules, PSMRoot, NodeScheduleTable);
    relation (commGraph, PSMRoot, CommGraph);
    relation (messages, PSMRoot, Message);
    relation (frames, PSMRoot, Frame);
    relation (gwRules, PSMRoot, GWRule);

    entity (DataTypeMapping);
    relation (pimDataType, DataTypeMapping, pim.DataType);
    relation (platformDataType, DataTypeMapping, dt.DataType);

    entity (JobCompatibilityMapping);
    relation (component, JobCompatibilityMapping, pim.ComponentInstance);
    relation (executionElement, JobCompatibilityMapping, platform.ComponentInstance);

    entity (JobAllocation);
    relation (component, JobAllocation, pim.ComponentInstance);
    relation (executionElement, JobAllocation, platform.ComponentInstance);

    entity (NodeScheduleTable);
    relation (node, NodeScheduleTable, platform.ComponentInstance);
    relation (jobTriggerings, NodeScheduleTable, JobTriggering);
    relation (period, NodeScheduleTable, Double);

    entity (JobTriggering);
    relation (job, JobTriggering, pim.ComponentInstance);
    relation (baseTime, JobTriggering, Double);
    relation (period, JobTriggering, Double);

    entity (CommGraph);
    relation (nodes, CommGraph, CommNode);
    relation (links, CommGraph, CommLink);

    entity (CommLink);
    entity (CommNode);
    relation (connection, CommNode, CommLink);

    entity (Message);
    relation (size, Message, Integer);
    relation (dataElements, Message, DataElementToMessageMapping);
    relation (sender, Message, CommNode);
    relation (receivers, Message, CommNode);

    entity (DataElementToMessageMapping);
    relation (dataElement, DataElementToMessageMapping, pim.DataElement);
    relation (startingPosition, DataElementToMessageMapping, Integer);
    relation (updateIndicationBitPosition, DataElementToMessageMapping, Integer);

    entity (Frame);
    relation (size, Frame, Integer);
    relation (messages, Frame, MessageToFrameMapping);
    relation (sender, Frame, CommNode);
    relation (receivers, Frame, CommNode);
    relation (link, Frame, CommLink);

    entity (MessageToFrameMapping);
relation(message, MessageToFrameMapping, Message);
relation(startingPosition, MessageToFrameMapping, Integer);
relation(updateIndicationBitPosition, MessageToFrameMapping, Integer);

definition(GWRule);
relation(node, GWRule, CommNode);
relation(message, GWRule, Message);
relation(sourceFrame, GWRule, Frame);
relation(targetFrames, GWRule, Frame);
}
Appendix B

Algorithms from Chapter 5

B.1 Connector validity specification

B.1.1 Assembly connectors

The following pattern should be fulfilled by a valid assembly connector.

```
pattern validAssemblyConnector(Conn) = {
  Assembly(Conn);
  ComponentInstance(CIS);
  ComponentInstance(CIT);
  Interface(IFS);
  Interface(IFT);
  sourceComponent(SC,Conn,CIS);
  targetComponent(ST,Conn,CIT);
  sourceInterface(SI,Conn,IFS);
  targetInterface(TI,Conn,IFT);

  Component(CS);
  Component(CT);
  type(T1,CIS,CS);
  type(T2,CIT,CT);
  input(I1,CT,IFT);
  output(I2,CS,IFS);

  CompositeComponent(Comp);
  components(C1,Comp,CIS);
  components(C2,Comp,CIT);
  connectors(C3,Comp,Conn);
}
```

B.1.2 Delegate connectors

One of the following patterns should be fulfilled by a valid delegate connector.

```
pattern validDelegateConnectorA(Conn) = {
  Delegate(Conn);

  ComponentInstance(CIT);
  Interface(IFS);
  Interface(IFT);

  targetComponent(ST,Conn,CIT);
  sourceInterface(SI,Conn,IFS);
  targetInterface(TI,Conn,IFT);

  Component(CT);
  CompositeComponent(Comp);
}
```
type(T2,CIT,CT);
input(I1,Comp,IFS);
input(I2,CT,IFT);

components(C1,Comp,CIT);
connectors(C2,Comp,Conn);

pattern validDelegateConnectorB(Conn) = {
  Delegate(Conn);
  ComponentInstance(CIS);
  Interface(IFS);
  Interface(IFT);
  sourceComponent(SC,Conn,CIS);
  sourceInterface(SI,Conn,IFS);
  targetInterface(TI,Conn,IFT);
  Component(CS);
  CompositeComponent(Comp);
  type(T1,CIS,CS);
  output(I1,Comp,IFT);
  output(I2,CS,IFS);

  components(C1,Comp,CIS);
  connectors(C2,Comp,Conn);
}

B.2 Hardware connector constraints

Pattern that is fulfilled by pure linking connections:

pattern pureLinkingConnection(Conn) = {
  Connection(Conn);
  ConnectorInstance(ConI);
  Connector(C);
  type(ConI,C);
  role(R,C,LinkingConnector);
  neg pattern negp(Conn) = {
    Connection(Conn);
    ConnectorInstance(CI);
    Connector(Co);
    type(CI,Co);
    role(R2,Co,LocalConnector);
  }
}

Pattern that is fulfilled by pure local connections:

pattern pureLocalConnection(Conn) = {
  Connection(Conn);
  ConnectorInstance(ConI);
  Connector(C);
  type(ConI,C);
  role(R,C,LocalConnector);
  neg pattern negp(Conn) = {
    Connection(Conn);
    ConnectorInstance(CI);
    Connector(Co);
    type(CI,Co);
    role(R2,Co,LinkingConnector);
  }
}
B.3 Data Type Size Calculation

The implementation of the Size function returning the size of platform data types:

```vttcl
rule dtSize (in DT, out Size) = seq {
    try choose S with find primitiveDT(DT,S) do seq {
        update Size = toInteger(value(S));
    }
    else seq {
        let S=undefined in
        try choose L, DTE with find arrayType(DT,L,DTE) do seq {
            call dtSize(DTE,S);
            update Size = S*toInteger(value(L));
        }
        else seq {
            update Size = 0;
            foreach Mem, DTE with find recordMember(DT,Mem,DTE) do seq {
                call dtSize(DTE,S);
                update Size = Size + S;
            }
        }
    }
}
```

B.4 PIM Flattening

A simple reference metamodel (consisting of a single VPM relation) is required by the algorithm runtime:

```
relation(refrel, pim.ComponentInstance, pim.ComponentInstance);
```

The transformation specification in VTCL:

```vttcl
import pim;
import refmodel;

machine pimflattening {
    pattern appMain(A,M)= {
        
    }
}
```
Application(A);
Application.main(XM,A,M);
CompositeComponent(M);
}

pattern compositeInstance(M,Inst) = {
    CompositeComponent(M);
    CompositeComponent(IType);
    ComponentInstance(Inst);
    ComponentInstance.type(TT,Inst,IType);
    CompositeComponent.components(CI,M,Inst);
}

grule copyComponentInstances(in M, in Inst, out CInst) = {
    precondition pattern pre(M,Inst,CInst,CType) = {
        CompositeComponent(M);
        CompositeComponent(IType);
        ComponentInstance(Inst);
        ComponentInstance.type(TT,Inst,IType);
        ComponentInstance(CInst);
        CompositeComponent.components(CI,M,Inst);
    }
    postcondition pattern post(M,Inst,CInst,CInst2,CType,X) = {
        CompositeComponent(M);
        ComponentInstance(CInst2) in M;
        CompositeComponent.components(C2,M,CInst2);
        ComponentInstance.refrel(X,CInst,CInst2);
        ComponentInstance.type(Cd,CInst2,CType);
        ComponentInstance(Inst);
        ComponentInstance(CInst);
        ComponentInstance.type(CTT,CInst,CType);
        Component(CType);
    }
    action {
        rename(CInst2,name(CInst));
    }
}

grule copyAssemblyInstances(in M, in Inst, out Ass) = {
    precondition pattern pre(M, Inst, Ass, SI, TI, CSC, CTC) = {
        CompositeComponent(M);
        CompositeComponent(IType);
        ComponentInstance(Inst);
        ComponentInstance.type(TT, Inst, IType);
        ComponentInstance(SC);
        ComponentInstance(TC);
        ComponentInstance(CSC);
        ComponentInstance(CTC);
        ComponentInstance.refrel(X1,SC,CSC);
        ComponentInstance.refrel(X2,TC,CTC);
        Interface(SI);
        Interface(TI);
        Assembly(Ass);
        CompositeComponent.connectors(CI,IType,Ass);
        Connector.sourceComponent(A1,Ass,SC);
    }
    postcondition pattern post(M, Inst, Ass, Ass2, SI, TI, CSC, CTC) = {
        CompositeComponent(M);
        ComponentInstance(Inst);
        ComponentInstance.type(TT, Inst, IType);
        ComponentInstance(SC);
        ComponentInstance(TC);
        ComponentInstance(CSC);
        ComponentInstance(CTC);
        ComponentInstance.refrel(X1,SC,CSC);
        ComponentInstance.refrel(X2,TC,CTC);
        Interface(SI);
        Interface(TI);
        Assembly(Ass);
        CompositeComponent.connectors(CI,IType,Ass2);
        Connector.sourceComponent(A1,Ass,SC);
    }
    action {
        rename(Ass2,name(Ass));
    }
}
B.4. PIM FLATTENING

```plaintext
Connector . sourceInterface (A2 , Ass , SI );
Connector . targetComponent (A3 , Ass , TC );
Connector . targetInterface (A4 , Ass , TI );
}
postcondition pattern post (M, SI , TI , CSC , CTC , Ass2 ) = {
  CompositeComponent (M);
  Assembly (Ass2 ) in M;
  CompositeComponent . connectors (Cx , M , Ass2 );
  Connector . sourceComponent (A1c , Ass2 , CSC );
  Connector . sourceInterface (A2c , Ass2 , SI );
  Connector . targetComponent (A3c , Ass2 , CTC );
  Connector . targetInterface (A4c , Ass2 , TI );
  Interface (SI );
  Interface (TI );
  ComponentInstance (CSC );
  ComponentInstance (CTC );
}
}
grule moveInputDelegation (in M , in Inst , out Del , out Ass ) = {
precondition pattern pre (M , Inst , Iface , Del , Ass , Iface2 , Inst2 , X11 , X12 , CInst2 ) = {
  CompositeComponent (M);
  CompositeComponent (IType);
  ComponentInstance (Inst);
  ComponentInstance . type (TT , Inst , IType );
  Interface (Iface );
  Component . input (CI , IType , Iface );
  Assembly (Ass );
  CompositeComponent . connectors (X1 , M , Ass );
  Connector . targetComponent (X11 , Ass , Inst );
  Connector . targetInterface (X12 , Ass , Iface );
  Delegate (Del );
  CompositeComponent . connectors (X2 , IType , Del );
  Connector . targetComponent (X21 , Del , Inst2 );
  Connector . targetInterface (X23 , Del , Iface2 );
  ComponentInstance (Inst2 );
  ComponentInstance (CInst2 );
  ComponentInstance . refrel (X , Inst2 , CInst2 );
  Interface (Iface2 );
}
postcondition pattern post (Ass , Iface2 , Inst , Iface , X11 , X12 , Xa1 , Xa2 , CInst2 ) = {
  ComponentInstance (Inst);
  Interface (Iface );
  Assembly (Ass );
  Connector . targetComponent (Xa1 , Ass , CInst2 );
  Connector . targetInterface (Xa2 , Ass , Iface2 );
  ComponentInstance (CInst2 );
  Interface (Iface2 );
}
}
grule moveOutputDelegation (in M , in Inst , out Del , out Ass ) = {
precondition pattern pre (M , Inst , Iface , Del , Ass , Iface2 , CInst2 , X11 , X12 ) = {
  CompositeComponent (M);
  CompositeComponent (IType);
```
B.5 PIM Component Instance Replication

```plaintext
import pim;

machine compReplication {

pattern appMain(A,M) = {
   Application(A);
}

rule main(in Atmp) = let App=ref(Atmp) in seq {
   try choose Main with find appMain(App,Main) do seq {
      iterate choose Inst with find compositeInstance(Main,Inst) do seq {
         // copy component instances
         forall A with apply copyComponentInstances(Main,Inst,A) do skip;

         // copy assembly connectors
         forall A with apply copyAssemblyInstances(Main,Inst,A) do skip;

         // delegation resolve
         forall D, A with apply moveInputDelegation(Main,Inst,D,A) do skip;
         forall D, A with apply moveOutputDelegation(Main,Inst,D,A) do skip;

         // delete composition instance
         delete(Inst);
      }
   }
}
```
B.5. PIM COMPONENT INSTANCE REPLICATION

```plaintext
Application.main(XM,A,M);
CompositeComponent(M);

pattern compToReplicate(Main,Comp,Red, CType) = {
ComprisingComponent(Main);
ComprisingComponent.components(C,Main,Comp);
ComponentInstance(Comp);
ComponentInstance.type(T,Comp,CType);
Component(CType);
Component.redundancyDegree(R(CType, RedDeg);
pimdepy.Red(RedDeg);
pimdepy.Red.value(V,RedDeg,Red);
nfp.ValueSpecification(Red);
check(toInteger(value(Red)) > 1);
}

grule createComponentInstance(in Main, in Type, in Name, out Inst) = {
precondition pattern pre(Main,Type) = {
ComprisingComponent(Main);
Component(Type);
}
postcondition pattern post(Main,Type,Inst) = {
ComprisingComponent(Main);
Component(Type);
ComponentInstance(Inst) in Main;
ComprisingComponent.components(C,Main,Inst);
ComponentInstance.type(T,Inst,Type);
}
action {
rename(Inst,Name);
}
}

grule copyInboundConnector(in Main, in Comp, in NewComp, in Postfix, out Ass) = {
precondition pattern pre(Main,Comp,NewComp,Ass,SrcC,SrcI,TrgI) = {
ComprisingComponent(Main);
ComponentInstance(Comp);
ComponentInstance(NewComp);
Assembly(Ass);
Interface(SrcI);
Interface(TrgI);
ComponentInstance(SrcC);
Connector.sourceInterface(SI,Ass,SrcI);
Connector.targetInterface(TI,Ass,TrgI);
Connector.sourceComponent(SC,Ass,SrcC);
Connector.targetComponent(TC,Ass,Comp);
}
postcondition pattern post(Main,NewComp,SrcC,SrcI,TrgI,Ass2) = {
ComprisingComponent(Main);
ComponentInstance(NewComp);
Assembly(Ass2) in Main;
Interface(SrcI);
Interface(TrgI);
ComponentInstance(SrcC);
ComponentInstance.connectors(Conn,Main,Ass2);
Connector.sourceInterface(SI2,Ass2,SrcI);
Connector.targetInterface(TI2,Ass2,TrgI);
Connector.sourceComponent(SC2,Ass2,SrcC);
Connector.targetComponent(TC2,Ass2,NewComp);
}
```
action {
    rename(Ass2, name(Ass)+Postfix);
}

gtrule copyOutboundConnector(in Main, in Comp, in NewComp, in Postfix, out Ass) = {
    precondition pattern pre(Main,Comp,NewComp,Ass,TrgC,SrcI,TrgI)= {
        CompositeComponent(Main);
        ComponentInstance(Comp);
        ComponentInstance(NewComp);
        Assembly(Ass);
        Interface(SrcI);
        Interface(TrgI);
        ComponentInstance(TrgC);
        Connector.sourceInterface(SI,Ass,SrcI);
        Connector.targetInterface(TI,Ass,TrgI);
        Connector.sourceComponent(SC,Ass,Comp);
        Connector.targetComponent(TC,Ass,TrgC);
    }
    postcondition pattern post(Main,NewComp,TrgC,SrcI,TrgI,Ass2)= {
        CompositeComponent(Main);
        ComponentInstance(NewComp);
        Assembly(Ass2) in Main;
        Interface(SrcI);
        Interface(TrgI);
        ComponentInstance(TrgC);
        CompositeComponent.connectors(Conn,Main,Ass2);
        Connector.sourceInterface(SI2,Ass2,SrcI);
        Connector.targetInterface(TI2,Ass2,TrgI);
        Connector.sourceComponent(SC2,Ass2,NewComp);
        Connector.targetComponent(TC2,Ass2,TrgC);
    }
    action {
        rename(Ass2, name(Ass)+Postfix);
    }
}

pattern connectingAssembly(Comp, Asse)= {
    Assembly(Asse);
    ComponentInstance(Comp);
    Connector.sourceComponent(SC2,Asse,Comp);
} or {
    Assembly(Asse);
    ComponentInstance(Comp);
    Connector.targetComponent(TC2,Asse,Comp);
}

rule createInstance(in Main, in Comp,in Type, in Index, in Num) = seq {
    if (Index<=Num) seq {
        print("Creating instance "+Index+"\n");
        let NewInst=undef in choose with apply
            createComponentInstance(Main,Type,name(Comp)\"_\"+Index,NewInst) do seq {
                print("processing "+name(NewInst)\"\n");
                //duplicate inbound assembly connectors
                forall Assem with apply
                    copyInboundConnector(Main, Comp,NewInst, \"_\"+Index,Assem) do skip;
                //duplicate outbound assembly connectors
                forall Assem with apply
                    copyOutboundConnector(Main, Comp,NewInst, \"_\"+Index,Assem) do skip;
            }
        call createInstance(Main,Comp,Type,Index+1,Num);
    }
}
B.6 Platform Model Flattening

A simple reference metamodel (consisting of two VPM relations) is required by the algorithm runtime:

relation(refrel, platform.ComponentInstance, platform.ComponentInstance);
relation(refrel, platform.ConnectorInstance, platform.ConnectorInstance);

The transformation specification in VTCL:

import platform;

machine platformflattening {
    pattern compositeInstance(Main,Inst,IType) = {
        CompositeComponent(Main);
        CompositeComponent.components(C,Main,Inst);
        ComponentInstance(Inst);
        ComponentInstance.type(T,Inst,IType);
        CompositeComponent(IType);
    }

gtrule copyComponentInstances(in Src, in Trg, out CI) = {
    precondition pattern pre(Src,Trg,CI, CType) = {
        CompositeComponent(Src);
        CompositeComponent.components(X,Src,CI);
        CompositeComponent(Trg);
        ComponentInstance(CI);
        ComponentInstance.type(T,CI,CType);
        Component(CType);
    }

    postcondition pattern post(Src,Trg,NewX,CI2,CType) = {
        CompositeComponent(Src);
        CompositeComponent.components(NewX,Trg,CI2);
        CompositeComponent(Trg);
        ComponentInstance(CI);
        ComponentInstance(CI2) in Trg;
        ComponentInstance.refrel(R,CI,CI2);
        ComponentInstance.type(T2,CI2,CType);
        Component(CType);
    }

    action {
        rename(CI2,name(CI));
        forall Conni with apply
            copyConnectorInstances(CI,CI2,Conni) do skip;
    }
}
Algorithm 1: copyConnectorInstances

\[ \text{copyConnectorInstances}(\text{in } \text{Src}, \text{in } \text{Trg}, \text{out } \text{CI}) = \{ \]
\[ \text{precondition pattern } \text{pre}(\text{Src}, \text{Trg}, \text{CI}, \text{CT}) = \{ \]
\[ \text{ConnectorInstance}(\text{Src}); \]
\[ \text{ConnectorInstance}(\text{Trg}); \]
\[ \text{ComponentInstance}.\text{connectorInstances}(\text{Cx}, \text{Src}, \text{CI}); \]
\[ \text{ConnectorInstance}(\text{CI}); \]
\[ \text{ConnectorInstance}.\text{type}(\text{Tx}, \text{CI}, \text{CT}); \]
\[ \text{Connector}(\text{CT}); \}
\]
\[ \text{postcondition pattern } \text{post}(\text{Trg}, \text{CT}, \text{CI2}, \text{CI}) = \{ \]
\[ \text{ComponentInstance}(\text{Trg}); \]
\[ \text{ComponentInstance}.\text{connectorInstances}(\text{Cx}, \text{Trg}, \text{CI2}); \]
\[ \text{ConnectorInstance}(\text{CI2}) \text{ in } \text{Trg}; \]
\[ \text{ConnectorInstance}(\text{CI}); \]
\[ \text{ConnectorInstance}.\text{type}(\text{Tx2}, \text{CI2}, \text{CT}); \]
\[ \text{ConnectorInstance}.\text{refrel}(\text{RR}, \text{CI}, \text{CI2}); \]
\[ \text{Connector}(\text{CT}); \}
\]
\[ \text{action } \{ \]
\[ \text{rename}(\text{CI2}, \text{name}(\text{CI})); \]
\[ \}
\]
Algorithm 2: copyConnections

\[ \text{copyConnections}(\text{in } \text{Src}, \text{in } \text{Trg}, \text{out } \text{Conn}) = \{ \]
\[ \text{precondition pattern } \text{pre}(\text{Src}, \text{Trg}, \text{Conn}) = \{ \]
\[ \text{CompositeComponent}(\text{Src}); \]
\[ \text{CompositeComponent}(\text{Trg}); \]
\[ \text{CompositeComponent}.\text{connection}(\text{Cx}, \text{Src}, \text{Conn}); \]
\[ \text{Connection}(\text{Conn}); \]
\[ \text{neg pattern } \text{negp}(\text{Conn}) = \{ \]
\[ \text{DelegateConnection}(\text{Conn}); \}
\]
\[ \} \]
\[ \text{postcondition pattern } \text{post}(\text{Trg}, \text{Conn2}) = \{ \]
\[ \text{CompositeComponent}(\text{Trg}); \]
\[ \text{CompositeComponent}.\text{connection}(\text{Cx2}, \text{Trg}, \text{Conn2}); \]
\[ \text{Connection}(\text{Conn2}) \text{ in } \text{Trg}; \}
\]
\[ \text{action } \{ \]
\[ \text{rename}(\text{Conn2}, \text{name}(\text{Conn})); \]
\[ \text{forall } \text{M with apply} \]
\[ \text{copyConnectionMembers}(\text{Conn}, \text{Conn2}, \text{M}) \text{ do skip}; \]
\[ \}
\]
Algorithm 3: copyConnectionMembers

\[ \text{copyConnectionMembers}(\text{in } \text{Src}, \text{in } \text{Trg}, \text{out } \text{Mem}) = \{ \]
\[ \text{precondition pattern } \text{pre}(\text{Src}, \text{Trg}, \text{Mem}, \text{TrgMem}) = \{ \]
\[ \text{Connection}(\text{Src}); \]
\[ \text{Connection}(\text{Trg}); \]
\[ \text{Connection}.\text{members}(\text{M}, \text{Src}, \text{Mem}); \]
\[ \text{ConnectorInstance}(\text{Mem}); \]
\[ \text{ConnectorInstance}(\text{TrgMem}); \]
\[ \text{ConnectorInstance}.\text{refrel}(\text{R}, \text{Mem}, \text{TrgMem}); \}
\]
\[ \} \]
\[ \text{postcondition pattern } \text{post}(\text{Trg}, \text{TrgMem}) = \{ \]
\[ \text{Connection}(\text{Trg}); \]
\[ \text{Connection}.\text{members}(\text{M}, \text{Trg}, \text{TrgMem}); \]
\[ \text{ConnectorInstance}(\text{TrgMem}); \}
\]
B.6. PLATFORM MODEL FLATTENING

```plaintext
grule resolveDelegationConnections(
    in Src, in SrcInst, in Trg, out As, out Del, out Mem) = {
    precondition pattern pre(Src, SrcInst, Trg, As, Del, Mem, MemTrg, ME2) = {
        CompositeComponent(Src);
        CompositeComponent(Trg);
        ComponentInstance(SrcInst);
        CompositeComponent.connection(C1, Trg, As);
        Connection(As);
        DelegateConnection(Del);
        CompositeComponent.connection(C2, Src, Del);
        Connector(Conn);
        DelegateConnection.delegationConnector(DC1, Del, Conn);
        ConnectorInstance(CI);
        ComponentInstance.connectorInstances(CIIS, SrcInst, CI);
        Connection.members(ME, As, CI);
        ConnectorInstance.type(T, CI, Conn);
        ConnectorInstance(Mem);
        ConnectorInstance(MemTrg);
        ConnectorInstance.refrel(RR, Mem, MemTrg);
        Connection.members(ME2, Del, Mem);
    }
    postcondition pattern post(As, MemTrg) = {
        Connection(As);
        ConnectorInstance(MemTrg);
        Connection.members(ME3, As, MemTrg);
    }
}
```

```plaintext
pattern componentInstanceWithConnector(CompInst, CI) = {
    ComponentInstance(CompInst);
    ComponentInstance.connectorInstances(X, CompInst, CI);
    ConnectorInstance(CI);
}

pattern componentInstanceWithRefEdge(CompInst, Edge) = {
    ComponentInstance(CompInst);
    ComponentInstance(RS);
    ComponentInstance.refrel(Edge, RS, CompInst);
}

pattern connectorInstanceWithRefEdge(ConnInst, Edge) = {
    ConnectorInstance(ConnInst);
    ConnectorInstance(RS);
    ConnectorInstance.refrel(Edge, RS, ConnInst);
}

rule main(in C) = let Main = ref(C) in seq {
    iterate choose Inst, IType with find compositeInstance(Main, Inst, IType) do seq{
        // copy component instances
        forall CI with apply copyComponentInstances(IType, Main, CI) do skip;
        // copy connections
        forall Conn with apply copyConnections(IType, Main, Conn) do skip;
        // resolve delegation connectors
        forall Ass, Del, Mem with
            apply resolveDelegationConnections(IType, Inst, Main, Ass, Del, Mem)
            do skip;
    }
}
```
// delete connector instances of the processed instance
forall ConnI with find componentInstanceWithConnector(Inst, ConnI)
do delete(ConnI);

// delete the processed component instance
delete(Inst);

// remove temporary reference edges
forall CI below Main, RefEdge with
  find componentInstanceWithRefEdge(CI, RefEdge)
do delete(RefEdge);
forall CI below Main, RefEdge with
  find connectorInstanceWithRefEdge(CI, RefEdge)
do delete(RefEdge);

B.7 Data type mapping creation

The transformation specification in VTCL:

```
gtrule addDataTypeMapping(in PSM, in PIM_dt, in PM_dt, out Map) = {
  precondition pattern pre(PSM, PIM_dt, PM_dt) = {
    PSMRoot(PSM);
    pim.DataType(PIM_dt);
    dt.DataType(PM_dt);
    neg pattern negp(PSM, PIM_dt, PM_dt) = {
      // precondition: PIM data type not already mapped
      PSMRoot(PSM);
      pim.DataType(PIM_dt);
      dt.DataType(PM_dt);
      DataTypeMapping(ExistingMap);
      DataTypeMapping.pimDataType(X, ExistingMap, PIM_dt);
    }
  }

  postcondition pattern pre(PSM, PIM_dt, PM_dt, Map) = {
    PSMRoot(PSM);
    pim.DataType(PIM_dt);
    dt.DataType(PM_dt);
    DataTypeMapping(Map);
    DataTypeMapping.pimDataType(X, Map, PIM_dt);
    DataTypeMapping.platformDataType(Y, Map, PM_dt);
  }
}
```

B.8 Job compatibility mapping creation

The transformation specification in VTCL:

```
gtrule addJobCompatibilityMapping(in PSM, in PIM_comp, in PM_comp, out Map) = {
  precondition pattern pre(PSM, PIM_comp, PM_comp) = {
    PSMRoot(PSM);
    pim_ComponentInstance(PIM_comp);
    platform_ComponentInstance(PM_comp);

    // Platform component has ExecutionEnv role
    platform_ComponentInstance.type(Type, PM_comp, Comp);
    platform_ComponentRole(RoleRef, Comp, Role);
    platform_ComponentRole(Role);
    check (name(Role)=="ExecutionEnv");

    neg pattern negp(PSM, PIM_comp) = {
```
B.9 Communication Graph Synthesis

The transformation specification in VTCL:

```vtcl
import platform;
import psm;
machine commGraphSynth

gtrule createGraphNode(in M, in G, out Exe, out Node) = {
    precondition pattern pre(M,G,Exe) = {
        CompositeComponent(M);
        CommGraph(G);
        ComponentInstance(Exe);
        CompositeComponent.components(C,M,Exe);
        ComponentInstance.type(T,Exe,Type);
        Component(Type);
        Component.role(R,Type,Role);
        check (name(Role)=="ExecutionEnv");
    }
    postcondition pattern post(G,Exe,Node) = {
        CommGraph(G);
        ComponentInstance(Exe);
        CommNode(Node) in G;
        CommGraph.nodes(N,G,Node);
        ComponentInstance.node(X,Exe,Node);
    }
    action {
        rename(Node,name(Exe));
    }
}

gtrule createGraphLink(in M, in G, out Conn, out Link) = {
    precondition pattern pre(M,G,Conn) = {
        CompositeComponent(M);
        CommGraph(G);
        Connection(Conn);
        Connection.members(Memb,Conn,CI);
        CompositeComponent.connection(C,M,Conn);
        ConnectorInstance(CI);
        ConnectorInstance.type(CIT,CI,Type);
        Connector(Type);
    }
    postcondition pattern post(G,Conn,Link) = {
        CommGraph(G);
        ComponentInstance(Conn);
        CommNode(Node) in G;
        CommGraph.nodes(N,G,Node);
        ComponentInstance.node(X,Conn,Node);
        Link(Node);
    }
    action {
        rename(Link,name(Node));
    }
}
```
Connector.role(R,Type,Role);
ConnectorRole(Role);
check (name(Role)=="LinkingConnector");

postcondition pattern post(G,Conn,Link)= {
CommGraph(G);
Connection(Conn);
CommLink(Link) in G;
CommGraph.links(N,G,Link);
Connection.link(X,Conn,Link);
}

action {
rename(Link,name(Conn));
}

gtrule addCompToNode(in Node, out Conn, out Comp) = {
precondition pattern pre(Node,Conn,Comp)= {
ComponentInstance(Comp);
ComponentInstance(CDorig);
CommNode(Node);
ComponentInstance.node(X,COrig,Node);
Connection(Conn);
ComponentInstance.connectorInstances(CIS1,Comp,CI1);
ConnectorInstance(CI1);
ConnectorInstance.type(XI1,CI1,Connec);
Connector(Connec);
Connector.role(Ro,Connec,Role);
ConnectorRole(Rele);
check (name(Rele)=="LocalConnector");
Connection.members(M1,Conn,CI1);
ComponentInstance.connectorInstances(CIS2,CDorig,CI2);
ConnectorInstance(CI2);
Connection.members(M2,Conn,CI2);

neg pattern negp(Comp)= {
ComponentInstance(Comp);
CommNode(Node);
ComponentInstance.node(X,Comp,Node);
}

}
postcondition pattern post(Comp,Node)= {
ComponentInstance(Comp);
CommNode(Node);
ComponentInstance.node(X,Comp,Node);
}

gtrule connectNodeToLink(in G, in Link, in Conn, out Node) = {
precondition pattern pre(G,Link,Conn,Node)= {
CommGraph(G);
CommLink(Link);
CommNode(Node);
CommGraph.links(Li,G,Link);
CommGraph.nodes(No,G,Node);

Connection(Conn);
Connection.members(Mem,Conn,CI);
ConnectorInstance(CI);
ComponentInstance(Comp);
ComponentInstance.node(Nod,Conn,Node);
}

B.10 DATA ELEMENT TO MESSAGE MAPPING CONSTRAINTS

The constraint checking pattern definition (noMixedMessage):

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ETMessage</strong></td>
<td>( \text{Message}(\text{Msg}) )</td>
</tr>
<tr>
<td><strong>DSMessage</strong></td>
<td>( \text{Message}(\text{Msg}) )</td>
</tr>
<tr>
<td><strong>TTMessage</strong></td>
<td>( \text{Message}(\text{Msg}) )</td>
</tr>
</tbody>
</table>

```plaintext
postcondition pattern post(Node,Link) = {
  CommLink(Link);
  CommNode(Node);
  CommNode.connection(L,Node,Link);
}
}

rule main(in M) = let Main = ref(M), G = undef in seq {
  println("Root HW composition " + fqn(Main));
  // create comm graph root
  new(CommGraph(G));
  // create nodes
  let Node = undef in
  forall Exe with apply createGraphNode(Main,G,Exe,Node) do seq {
    // expand nodes through local connections
    iterate choose Comp,Conn with
      apply addCompToNode(Node,Conn,Comp) do skip;
  }
  let Link = undef in
  forall Con with apply createGraphLink(Main,G,Con,Link) do seq {
   forall No with apply connectNodeToLink(G,Link,Con,No) do skip;
  }
}
)
```

B.10 Data element to message mapping constraints

The constraint checking pattern definition (noMixedMessage):

```plaintext
pattern ETMessage(Msg) = {
  Message(Msg);
  Message.dataElements(D,Msg,MapET);
  DataElementToMessageMapping(MapET);
  DataElementToMessageMapping.dataElement(X,MapET,DEET);
  pim.Interface(SrcET);
  pim.Interface.containedElements(X1,SrcET,DEET);
  pim.DataElement(DEET);
  // Interface has message debounce time attached
  pim.Interface.debounceTime(DebET,SrcET,MsgDeb);
  pimtime.DebounceTime(MsgDeb);
}

pattern DSMessage(Msg) = {
  Message(Msg);
  Message.dataElements(D,Msg,MapDS);
  DataElementToMessageMapping(MapDS);
  DataElementToMessageMapping.dataElement(X,MapDS,DEDS);
  pim.Interface(SrcDS);
  pim.Interface.containedElements(X1,SrcDS,DEDS);
  pim.DataElement(DEDS);
  // Interface has message debounce time attached
  pim.Interface.bandwidth(DebDS,SrcDS,MsgBW);
  pimtime.DebounceTime(MsgBW);
}

pattern TTMessage(Msg) = {
  Message(Msg);
  Message.dataElements(D,Msg,MapTT);
  DataElementToMessageMapping(MapTT);
  DataElementToMessageMapping.dataElement(X,MapTT,DETT);
  pim.Interface(SrcTT);
  pim.Interface.containedElements(X1,SrcTT,DETT);
  pim.DataElement(DETT);
  // Interface has message period attached
  pim.Interface.period(Per,SrcTT,MsgPer);
}
pattern noMixedMessage(Msg) = {
    Message(Msg);
    find TTMessage(Msg);
    neg find ETMessage(Msg);
    neg find DSMessage(Msg);
} or {
    Message(Msg);
    find DSMessage(Msg);
    neg find ETMessage(Msg);
    neg find TTMessage(Msg);
} or {
    Message(Msg);
    find ETMessage(Msg);
    neg find TTMessage(Msg);
    neg find DSMessage(Msg);
}
Appendix C

Algorithms and models from Chapter 3

C.1 Design pattern definitions

C.1.1 Job with interfaces pattern

The following pattern is the precondition (checks the uniqueness of the name of the new job).

```
pattern uniqueNameElement (Name) = {
    PIMFunctionalElement (El);
    neg pattern negp(El,Name)= {
        PIMFunctionalElement (El);
        check (Name==name(El));
    }
}
```

The following ASM rule implements the design pattern (above DECOS PIM language):

```
rule jobWithInterfaces (in JobName, in JobType, 
                          in HasSPLIF, in HasSRLIF, in HasCOI)=
    let J= undef in seq {
        choose with apply createJob (JobName, JobType, J) do skip;
        if (HasSPLIF==true) choose with apply createSPLIFForJob (J) do skip;
        if (HasSRLIF==true) choose with apply createSRLIFForJob (J) do skip;
        if (HasCOI==true) choose with apply createCOIForJob (J) do skip;
    }

gtrule createJob (in Name, in Type, out J)= {
    precondition pattern pre (Type,T)= {
        entity (T);
        entity (JobSup);
        // find Job meta element
        check (fqn(JobSup)=="DECOS.PIM.Functionality.Job");
        // Type is subtype of Job
        supertypeOf (JobSup, T);
        // name matches
        check (name (T)==Type);
    }
    postcondition pattern post (T,J)= {
        entity (J);
        entity (T);
        instanceOf (J, T);
    }
    action {
        rename (J, Name);
    }
}

gtrule createSPLIFForJob (in J)= {
```

163
precondition pattern pre(J)= {
    Job(J);
}

postcondition pattern post(J,I)= {
    Job(J);
    SPLIF(I);
    Job.ownedInterface(X,J,I);
    Interface.interfaceOwnerJob(Z,I,J);
}

action {
    rename(I,name(J)+"_SPLIF");
}

{}
entity(EClass);
supertypeOf(EClassifier,EClass);

datatype(EDataType);
supertypeOf(EClassifier,EDataType);
datatype(datatypes)
{
datatype(EEnum);
supertypeOf(EDataType,EEnum);
datatype(EEnumLiteral);
typeOf(EEnum, EEnumLiteral);

datatype(EBigDecimal){
datatype(name) -> "EBigDecimal";
typeOf(datatypes.String, name);
relation(nameRel, EBigDecimal, name);
typeOf(nemf.ecore.ENamedElement.name, nameRel);
}
typeOf(EDataType,EBigDecimal);
datatype(EBigInteger){
datatype(name) -> "EBigInteger";
typeOf(datatypes.String, name);
relation(nameRel, EBigInteger, name);
typeOf(nemf.ecore.ENamedElement.name, nameRel);
}
typeOf(EDataType,EBigInteger);
datatype(EBoolean){
datatype(name) -> "EBoolean";
typeOf(datatypes.String, name);
relation(nameRel, EBoolean, name);
typeOf(nemf.ecore.ENamedElement.name, nameRel);
}
typeOf(EDataType,EBoolean);
datatype(EByte){
datatype(name) -> "EByte";
typeOf(datatypes.String, name);
relation(nameRel, EByte, name);
typeOf(nemf.ecore.ENamedElement.name, nameRel);
}
typeOf(EDataType,EByte);
datatype(EChar){
datatype(name) -> "EChar";
typeOf(datatypes.String, name);
relation(nameRel, EChar, name);
typeOf(nemf.ecore.ENamedElement.name, nameRel);
}
typeOf(EDataType,EChar);
datatype(EDate){
datatype(name) -> "EDate";
typeOf(datatypes.String, name);
relation(nameRel, EDate, name);
typeOf(nemf.ecore.ENamedElement.name, nameRel);
}
typeOf(EDataType,EDate);
datatype(EDouble){
datatype(name) -> "EDouble";
typeOf(datatypes.String, name);
relation(nameRel, EDouble, name);
typeOf(nemf.ecore.ENamedElement.name, nameRel);
}
typeOf(EDataType,EDouble);
datatype(EFloat){
datatype(name) -> "EFloat";
typeOf(datatypes.String, name);
    relation(nameRel, EFloat, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}

typeOf(EDataType, EFloat);

entity(EInt){
    entity(name) -> "EInt";
    typeOf(datatypes.String, name);
    relation(nameRel, EInt, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}

typeOf(EDataType, EInt);

entity(ELong){
    entity(name) -> "ELong";
    typeOf(datatypes.String, name);
    relation(nameRel, ELong, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}

typeOf(EDataType, ELong);

entity(EShort){
    entity(name) -> "EShort";
    typeOf(datatypes.String, name);
    relation(nameRel, EShort, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}

typeOf(EDataType, EShort);

entity(EBooleanObject){
    entity(name) -> "EBooleanObject";
    typeOf(datatypes.String, name);
    relation(nameRel, EBooleanObject, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}

typeOf(EDataType, EBooleanObject);

entity(EByteObject){
    entity(name) -> "EByteObject";
    typeOf(datatypes.String, name);
    relation(nameRel, EByteObject, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}

typeOf(EDataType, EByteObject);

entity(EByteArray){
    entity(name) -> "EByteArray";
    typeOf(datatypes.String, name);
    relation(nameRel, EByteArray, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}

typeOf(EDataType, EByteArray);

entity(ECharacterObject){
    entity(name) -> "ECharacterObject";
    typeOf(datatypes.String, name);
    relation(nameRel, ECharacterObject, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}

typeOf(EDataType, ECharacterObject);

entity(EDoubleObject){
    entity(name) -> "EDoubleObject";
    typeOf(datatypes.String, name);
    relation(nameRel, EDoubleObject, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
C.2. ECORE METAMODEL VPM REPRESENTATION

```plaintext
}
typeOf(EDataType, EDoubleObject);
entity(EGenericType)(
    entity(name) -> "EDoubleObject";
    typeOf(datatypes.String, name);
    relation(nameRel, EDoubleObject, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}
typeOf(EDataType, EFloatObject);
entity(EGenericType)(
    entity(name) -> "EFloatObject";
    typeOf(datatypes.String, name);
    relation(nameRel, EFloatObject, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}
typeOf(EDataType, EIntegerObject);
entity(EGenericType)(
    entity(name) -> "EIntegerObject";
    typeOf(datatypes.String, name);
    relation(nameRel, EIntegerObject, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}
typeOf(EDataType, ELongObject);
entity(EGenericType)(
    entity(name) -> "ELongObject";
    typeOf(datatypes.String, name);
    relation(nameRel, ELongObject, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}
typeOf(EDataType, EShortObject);
entity(EGenericType)(
    entity(name) -> "EShortObject";
    typeOf(datatypes.String, name);
    relation(nameRel, EShortObject, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}
typeOf(EDataType, EString);
entity(EGenericType)(
    entity(name) -> "EString";
    typeOf(datatypes.String, name);
    relation(nameRel, EString, name);
    typeOf(nemf.ecore.ENamedElement.name, nameRel);
}
entity(EPackage);
supertypeOf(ENamedElement, EPackage);
relation(nsUri, EPackage, datatypes.String);
relation(nsPrefix, EPackage, datatypes.String);
entity(EReource);
supertypeOf(ENamedElement, EResource);
relation(uri, EResource, datatypes.String);
relation(EStructuralFeature, EClass, EClassifier);
relation(name, EClass.EStructuralFeature, datatypes.String);
relation(changeable, EClass.EStructuralFeature, datatypes.Boolean);
relation(volatile, EClass.EStructuralFeature, datatypes.Boolean);
relation(transient, EClass.EStructuralFeature, datatypes.Boolean);
relation(unsettable, EClass.EStructuralFeature, datatypes.Boolean);
relation(many, EClass.EStructuralFeature, datatypes.Boolean);
relation(EReference, EClass, EClass);
supertypeOf(EClass.EStructuralFeature, EClass.EReference);
relation(containment, EClass.EReference, datatypes.Boolean);
relation(container, EClass.EReference, datatypes.Boolean);
```
C.3 MOF metamodel VPM representation

```plaintext
entity(cmof) {
  entity(metamodel) {
    entity(Argument) {
      relation(name, Argument, datatypes.String);
      relation(value, Argument, cmof.metamodel.Object);
    }
    entity(Association) {
      relation(endType, Association, Type);
      relation(isDerived, Association, datatypes.Boolean);
      relation(memberEnd, Association, Property);
      relation(navigableOwnedEnd, Association, Property);
      relation(ownedEnd, Association, Property);
    }
    entity( BehavioralFeature) {
      relation(ownedParameter, BehavioralFeature, Parameter);
      relation(raisedException, BehavioralFeature, Type);
    }
    entity(Class) {
      relation(isAbstract, Class, datatypes.Boolean);
      relation(ownedAttribute, Class, Property);
      relation(ownedOperation, Class, Operation);
      relation(superClass, Class, Class);
    }
    entity(Classifier) {
      relation(attribute, Classifier, Property);
      relation(feature, Classifier, Feature);
      relation(general, Classifier, Classifier);
      relation(inheritedMember, Classifier, NamedElement);
    }
    entity(Comment) {
      relation(annotatedElement, Comment, Element);
      relation(body, Comment, datatypes.String);
    }
    entity(Constraint) {
      relation(constrainedElement, Constraint, Element);
      relation(context, Constraint, Namespace);
      relation('namespace', Constraint, Namespace);
      relation(specification, Constraint, ValueSpecification);
    }
    entity(DataType) {
      relation(ownedAttribute, DataType, Property);
      relation(ownedOperation, DataType, Operation);
    }
    entity(DirectedRelationship) {
      relation(source, DirectedRelationship, Element);
      relation(target, DirectedRelationship, Element);
    }
    entity(Element) {
      relation(ownedComment, Element, Comment);
      relation(ownedElement, Element, Element);
    }
  }
}
```
relation(owner, Element, Element);
}
entity(ElementImport) {
    relation(alias, ElementImport, datatypes.String);
    relation(importedElement, ElementImport, PackageableElement);
    relation(importingNamespace, ElementImport, Namespace);
    relation(visibility, ElementImport, VisibilityKind);
}
entity(Enumeration) {
    relation(ownedLiteral, Enumeration, EnumerationLiteral);
}
entity(EnumerationLiteral) {
    relation enumeration, EnumerationLiteral, Enumeration);
}
entity(Exception) {
    relation(description, Exception, datatypes.String);
    relation(elementInError, Exception, Element);
    relation(objectInError, Exception, Element);
}
entity(Expression) {
    relation operand, Expression, ValueSpecification);
}
entity(Extent);
entity(Factory) {
    relation package, Factory, Package);
}
entity(Feature) {
    relation featuringClassifier, Feature, Classifier);
}
entity(Link) {
    relation association, Link, Association);
    relation firstElement, Link, Element);
    relation secondElement, Link, Element);
}
entity(MultiplicityElement) {
    relation(isOrdered, MultiplicityElement, datatypes.Boolean);
    relation(isUnique, MultiplicityElement, datatypes.Boolean);
    relation lower, MultiplicityElement, datatypes.Integer);
    relation upper, MultiplicityElement, datatypes.Integer);
}
entity(NamedElement) {
    relation name, NamedElement, datatypes.String);
    relation visibility, NamedElement, VisibilityKind);
}
entity(Namespace) {
    relation(elementImport, Namespace, ElementImport);
    relation(importedMember, Namespace, PackageableElement);
    relation member, Namespace, NamedElement);
    relation ownedRule, Namespace, Constraint);
    relation(packageImport, Namespace, PackageImport);
}
entity(Object);
entity(OpaqueExpression) {
    relation body, OpaqueExpression, datatypes.String);
    relation language, OpaqueExpression, datatypes.String);
}
entity(Operation) {
    relation bodyCondition, Operation, Constraint);
    relation class, Operation, Class);
    relation datatype, Operation, DataType);
    relation(isQuery, Operation, datatypes.Boolean);
    relation postcondition, Operation, Constraint);
    relation precondition, Operation, Constraint);
    relation redefinedOperation, Operation, Operation doctr
entity(Package) {
    relation(nestedPackage, Package, Package);
    relation(ownedMember, Package, PackageableElement);
    relation(ownedType, Package, Type);
    relation(packageMerge, Package, PackageMerge);
    relation(uRI, Package, datatypes.String);
}

entity(PackageImport) {
    relation(importedPackage, PackageImport, Package);
    relation(importingNamespace, PackageImport, Namespace);
    relation(visibility, PackageImport, VisibilityKind);
}

entity(PackageMerge) {
    relation(mergedPackage, PackageMerge, Package);
    relation(receivingPackage, PackageMerge, Package);
}

deep(PackageableElement);
deep(Parameter) {
    relation(default, Parameter, datatypes.String);
    relation(direction, Parameter, ParameterDirectionKind);
    relation(operation, Parameter, Operation);
}

deep(ParameterDirectionKind) {
    ParameterDirectionKind(in)->"in";
    ParameterDirectionKind(inout)->"inout";
    ParameterDirectionKind(out)->"out";
    ParameterDirectionKind(return)->"return";
}

deep(PrimitiveType);
deep(Property) {
    relation(association, Property, Association);
    relation(class, Property, Class);
    relation(datatype, Property, DataType);
    relation(default, Property, datatypes.String);
    relation(isComposite, Property, datatypes.Boolean);
    relation(isDerived, Property, datatypes.Boolean);
    relation(isDerivedUnion, Property, datatypes.Boolean);
    relation(isID, Property, datatypes.Boolean);
    relation(isReadOnly, Property, datatypes.Boolean);
    relation(opposite, Property, Property);
    relation(owningAssociation, Property, Association);
    relation(redefinedProperty, Property, Property);
    relation(subsettedProperty, Property, Property);
}

deep(RedefinableElement) {
    relation(redefinedElement, RedefinableElement, RedefinableElement);
    relation(redefinitionContext, RedefinableElement, Classifier);
}

deep(ReflectiveCollection);
deep(ReflectiveSequence);
deep(Relationship) {
    relation(relatedElement, Relationship, Element);
}

deep(StructuralFeature);
deep(Tag) {
    relation(element, Tag, Element);
    relation(name, Tag, datatypes.String);
    relation(value, Tag, datatypes.String);
    relation(package, Type, Package);
    relation(type, TypedElement, Type);
}
deep(Type);
deep(TypedElement);
C.4 ECore semantics definition

import nemf.ecore;

machine estoreMethods {
    /* INTERNAL PATTERNS */
    pattern isContainmentFeature(Feat) = {
        nemf.ecore.EClass.EAttribute(Feat,A,B);
    }
}
APPENDIX C. ALGORITHMS AND MODELS FROM CHAPTER 3

entity(A);
entity(B);
}
or {
    nemf.ecore.EClass.EReference(Feat,A,B);
    nemf.ecore.EClass.EReference.containment(CC,Feat,BV);
    datatypes.Boolean(BV);
    check(value(BV)=="true");
    entity(A);
    entity(B);
}
or {
    nemf.ecore.EClass.EAttribute(Feat,A,A);
    entity(A);
}
or {
    nemf.ecore.EClass.EReference(Feat,A,A);
    nemf.ecore.EClass.EReference.containment(CC,Feat,BV);
    datatypes.Boolean(BV);
    check(value(BV)=="true");
    entity(A);
}

pattern findFeature(Obj,Feat,Value)= {
    find findFeatureInt(Obj,Feat,X,Value);
}

pattern findFeatureInt(Obj,Feat,X,Value)= {
    entity(Obj);
    relation(Feat,A,B);
    entity(A);
    entity(B);
    entity(Value);
    relation(X,Obj,Value);
    instanceof(X,Feat);
}
or {
    entity(Obj);
    relation(Feat,A,A);
    entity(A);
    entity(Value);
    relation(X,Obj,Value);
    instanceof(X,Feat);
}
or {
    entity(Obj);
    relation(Feat,A,B);
    entity(A);
    entity(B);
    entity(Value);
    Value=Obj;
    relation(X,Obj,Value);
    instanceof(X,Feat);
}
or {
    entity(Obj);
    Value=Obj;
    relation(Feat,A,A);
    entity(A);
    entity(Value);
relation(X,Obj,Value);
instanceOf(X,Feat);
}

pattern findFirstFeature(Obj,Feat,X)= {
find findFeatureInt(Obj,Feat,X,Val);

neg pattern negp(X,Obj)= {
  entity(Obj);
  entity(Value);
  entity(Value2);
  relation(X,Obj,Value);
  relation(X2,Obj,Value2);
  EObject.orderedRelation.next(NX,X2,X);
}

neg pattern negp2(X,Obj)= {
  entity(Obj);
  entity(Value2);
  relation(X,Obj,Obj);
  relation(X2,Obj,Value2);
  EObject.orderedRelation.next(NX,X2,X);
}

neg pattern negp3(X,Obj)= {
  entity(Obj);
  relation(X,Obj,Value);
  relation(X2,Obj,Obj);
  EObject.orderedRelation.next(NX,X2,X);
}

neg pattern negp4(X,Obj)= {
  entity(Obj);
  relation(X,Obj,Obj);
  relation(X2,Obj,Obj);
  EObject.orderedRelation.next(NX,X2,X);
}

neg pattern negp5(X,Obj)= {
  entity(Obj);
  relation(X,Obj,Value);
  relation(X2,Obj,Value);
  EObject.orderedRelation.next(NX,X2,X);
}

}

pattern findNextFeature(Obj,Feat,Prev,X,NX)= {
  entity(V);
  entity(Obj);
  entity(VV);
  relation(Prev,Obj,V);
  relation(X,Obj,VV);
  EObject.orderedRelation.next(NX,Prev,X);
}
or {
  entity(Obj);
  entity(VV);
  relation(Prev,Obj,Obj);
  relation(X,Obj,VV);
  EObject.orderedRelation.next(NX,Prev,X);
}or {

}
APPENDIX C. ALGORITHMS AND MODELS FROM CHAPTER 3

```java
entity(V);
entity(Obj);
relation(Prev,Obj,V);
relation(X,Obj,Obj);
EObject.orderedRelation.next(NX,Prev,X);
}
or {
entity(V);
entity(Obj);
relation(Prev,Obj,V);
relation(X,Obj,V);
EObject.orderedRelation.next(NX,Prev,X);
}
or {
entity(Obj);
relation(Prev,Obj,Obj);
relation(X,Obj,Obj);
EObject.orderedRelation.next(NX,Prev,X);
}

pattern getParentFeature(Obj, MetaF) = {
entity(Obj) in P;
entity(P);
relation(F,P,Obj);
EClass.EReference(MetaF,A,B);
entity(A);
entity(B);
instanceOf(F, MetaF);
}
or {
entity(Obj) in P;
entity(P);
relation(F,P,Obj);
EClass.EReference(MetaF,A,A);
entity(A);
instanceOf(F, MetaF);
}

pattern getParent(Obj,P) = {
entity(Obj) in P;
entity(P);
}

rule internalGet(in Obj, in Feat, in Idx, out FeInst) = seq {
if (Idx==0)
choose FI with find findFirstFeature(Obj,Feat,FI) do update FeInst=FI;
else
let Prev=undef in
seq {
   call internalGet(Obj,Feat,Idx-1,Prev);
   choose FI,NX with find
   findNextFeature(Obj,Feat,Prev,FI,NX) do update FeInst=FI;
}
}

//EStore interface implementation/

rule add(in Obj, in Feat, in Idx, in Value) =
let IsSet=false, Rel=undef, Prev=undef, N=undef,Next=undef in
seq {
call isSet(Obj,Feat,IsSet);
}```
C.4. ECORE SEMANTICS DEFINITION

try choose with find isContainmentFeature(Feat) do move(Value, Obj);
if (IsSet) seq {
    if (Idx==0) seq {
        call internalGet(Obj, Feat, 0, Prev);
        new(relation(Rel, Obj, Value));
        new(instanceOf(Rel, Feat));
        new(EObject.orderedRelation.next(N, Prev, Rel));
    }
    else seq {
        call internalGet(Obj, Feat, Idx-1, Prev);
        try call internalGet(Obj, Feat, Idx, Next);
        if (Next! = undef)
            choose NX with find
            findNextFeature(Obj, Feat, Prev, Next, NX)
            do delete(NX);
            new(relation(Rel, Obj, Value));
            new(instanceOf(Rel, Feat));
            new(EObject.orderedRelation.next(N, Prev, Rel));
        if (Next! = undef)
            new(EObject.orderedRelation.next(N, Rel, Next));
    }
    else seq {
        new(relation(Rel, Obj, Value));
        new(instanceOf(Rel, Feat));
    }
}

rule clear(in Obj, in Feat)= seq {
    forall V with find findFeature(Obj, Feat, V) do delete(V);
}

rule create(in Cls, out Obj)= seq {
    new(entity(Obj));
    new(instanceOf(Obj, Cls));
}

rule get(in Obj, in Feat, in Idx, out Val)= let FI= undef in seq {
    call internalGet(Obj, Feat, Idx, FI);
    update Val=target(FI);
}

rule getContainer(in Obj, out Cont)= seq {
    try choose P with find getParent(Obj, P) do update Cont=P;
}

rule getContainingFeature(in Obj, out Cont)= seq {
    forall F with find getParentFeature(Obj, F) do seq {
        try choose with find isContainmentFeature(F)
        do update Cont=F;
    }
}

rule indexOf(in Obj, in Feat, in Value, out Idx)=
let I=0, FI=undef in seq {
    update Idx=-1;
    try iterate seq {
        call internalGet(Obj, Feat, I, FI);
        if (target(FI)==Value) seq {
            update Idx=I;
            fail;
        }
        update I=I+1;
    }
}
rule isEmpty (in Obj, in Feat, out Empt) =
  let XX= undef in seq {
    try seq {
      call internalGet (Obj, Feat, 0, XX);
      update Empt=false;
    }
    else update Empt=true;
  }

rule isSet (in Obj, in Feat, out Set) = seq {
  try choose V with find findFeature (Obj,Feat,V) do seq {
    update Set=true;
  }
  else update Set=false;
}

rule lastIndexOf (in Obj, in Feat, in Val, out Idx) =
  let I=0,FI= undef in seq {
    update Idx=-1;
    try iterate seq {
      call internalGet (Obj, Feat, I, FI);
      if (target (FI)== Val) seq {
        update Idx=I;
      }
      update I=I+1;
    }
  }

/* move is a keyword in VTCL*/
rule movee (in Obj, in Feat, in Trg, in Src, out Val) =
  let SI= undef ,TI= undef , Val2 = undef in seq {
    call internalGet (Obj, Feat, Src, SI);
    call internalGet (Obj, Feat, Trg, TI);
    update Val=target (SI);
    call remove (Obj,Feat,Src,Val2);
    setTo (TI, Val);
  }

rule remove (in Obj, in Feat, in Idx, out Val) =
  let Act= undef ,Next= undef,Pre= undef, N= undef in seq {
    if (Idx==0) seq {
      call internalGet (Obj,Feat,0,Act);
      update Val=target (Act);
      delete (Act);
    }
    else seq {
      call internalGet (Obj,Feat,Idx-1,Pre);
      call internalGet (Obj,Feat,Idx,Act);
      try call internalGet (Obj,Fea,Idx+1,Next);
      update Val=target (Act);
      delete (Act);
      if (Next!=undef) seq {
        new (EObject.orderedRelation.next(N,Pre,Next));
      }
    }
  }

rule set (in Obj, in Feat, in Idx, in NewVal, out Val) =

let SI=undef in seq {
    call internalGet(Obj, Feat, Idx, SI);
    update Val=target(SI);
    setTo(SI,NewVal);
    try choose with find isContainmentFeature(Feat) do move(NewVal, Obj);
}

rule size(in Obj, in Feat, out Size)=
    seq {
        update Size=0;
        forall Ref, Value with find findFeatureInt(Obj, Feat, Ref, Value) do update Size=Size+1;
    }

rule unset(in Obj, in Feat)=
    seq {
        forall Ref, Value with find findFeatureInt(Obj, Feat, Ref, Value) do delete(Ref);
    }
Appendix D

PIM-PSM mapping example

This appendix gives an introduction to the PIM-PSM mapping tools through a simple example that is developed for the DECOS project. The actual tool implementation that is used for the screenshots has been implemented using the generative framework discussed in Section 3.5.

D.1 Input models

The intended functionality of the system is a simple climate control application where the IT system is controlling a fan and a radiator element in order to control the temperature of a room. The software model (see Figure D.1) consists of five software components. Keypad reader is responsible for controlling a user input device, and Temp reader reads the current temperature value from a sensor. Temp regulator implements a simple control algorithm that calculates the commands for the actuators. The actuators (fan and heater) are controlled by jobs fan control and radiator control, respectively.

Figure D.1: Software model of the example

Figure D.2: Platform model of the example
The platform model (see Figure D.2) is simple, containing two nodes. The first node contains the input (sensor) peripherals and the second node is connected to the actuators.

D.2 The PIM-PSM mapping

The PIM-PSM mapping can be started by generating a mapping file through the new file wizard of the mapping tool. After the file is created, the tool starts and an introduction page is shown (see Figure D.3).

Figure D.3: Start page of the PIM-PSM tool

The next step is the import of the PIM and platform models. After these steps are completed, the first mapping step – data type mapping – can be started. The user interface of this step is illustrated by Figure D.4. As it can be seen in the Figure, the tool issues consistency check errors if the mapping is incomplete. These messages are displayed using the standard mechanisms of the tool environment.

The next steps of the mapping include the job and interface type mappings. In our example, these are completed by marking all elements to native (DECOS). After that, the PIM level sensor/actuator requirements are mapped to hardware elements. In our example, the sensors are mapped to I/O devices on node 1, and the actuators to output peripherals on node 2.

Job compatibility is specified after the resource allocation. Figure D.5 illustrates the mapping of the example. After this marking step, the automatic allocation takes place (see Figure D.6). The results of the allocation is the mapping of software components to hardware peripherals. The final step of the PIM-PSM mapping is (after the automatic, hidden synthesis of the communication matrix) the generation of middleware configuration files and scheduler input files.
Figure D.4: Data type mapping

Figure D.5: Component compatibility mapping
Figure D.6: Result of the allocation