UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCIO FERREIRA DA SILVA OLIVEIRA

Model Driven Engineering Methodology for Design Space Exploration of Embedded Systems

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Prof. Dr. Flávio Rech Wagner
Advisor

Prof. Dr. ret. nat. Franz-Joseph Rammig
Co-advisor

Porto Alegre, October 2013

CIP – CATALOGING-IN-PUBLICATION

Oliveira, Marcio Ferreira da Silva

Model Driven Engineering Methodology for Design Space Exploration of Embedded Systems / Marcio Ferreira da Silva Oliveira. – Porto Alegre: PPGC da UFRGS, 2013.

198 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–Brazil, 2013. Advisor: Flávio Rech Wagner; Co-advisor: Franz-Joseph Rammig.

1. Embedded systems. 2. Design space exploration. 3. Model-driven engineering. 4. UML. 5. Platform-based design. I. Wagner, Flávio Rech. II. Rammig, Franz-Joseph. III. Título.

ACKNOWLEDGEMENT

Again, I reached the end of another journey and I would like to thank many people and institutions that I had met in the way and contributed directly or indirectly to this work. I start with my thanks to “Conselho Nacional de Pesquisa” (CNPq), and “Deutscher Akademischer Austausch Dienst” (DAAD) for the partial support to my PhD research.

This journey started in the Embedded System Laboratory at the Federal University of Rio Grande do Sul (UFRGS), Brazil, where I found more friends than colleagues. We studied or worked together for four years, and from them I learned the fundamentals that guided me to the end of this journey. Now they are all spread from the south to the north of Brazil, but I hope we stay close some how. I am grateful to all people in the Embedded System Laboratory. I should mention some people for the discussions, contributions, and friendship: Bixo (a.k.a Ronaldo Ferreira), Francisco Nascimento, Marco Wehrmeister, Lisane Brisolara, and Gustavo Girão, thank you. Special thanks to Bixo for your friendship and support.

At some point in time, my journey has yawed to the “Old World”(Europe), where I met new people and new culture. What should be a one year stay became an unlimited time residence and now Germany is also my country. Thanks to the people from Rammig A.G. and C-LAB at University of Paderborn (UPB) for the great time we stayed together. Thanks to Henning Zabel and Alexander Krupp, for your welcome and support when I joined the group. Thanks to other colleagues from the ADT Group at C-LAB, Wolfgang Müller, Mabel Joy, Markus Becker, Kay Klobedanz, Tao Xie, Da He, Fabian Mischkalla, Gilles Defo (Bertrand), Chistoph Kuznik, Diana Riemer, and Jan Jatzkowski.

I would like to acknowledge my supervisors, Prof. Flávio Wagner (UFRGS) and Prof. Franz Rammig (UPB). Thanks for sharing your time and knowledge. Thanks for your unbounded patience and correct advices during all this time.

Special thanks to my relatives and my family, which got expanded since I left Brazil. There are no words to express my gratitude to my wife, Daniele Avila, who followed me all these years. Thanks for your support, encouragement and empathy. I want to thank my son Henrique, who was born during this journey and “donated” part of his time with me, so that I could spend more time at work (It is also an apology). Thanks to Avilas Family for your support and allowing me to kidnap your daughter and bring her to the other side of the Atlantic. Thanks to my parents, brother, and grand parents for understanding my dreams and accept the distance. I love you all.

Model Driven Engineering Methodology for Design Space Exploration of Embedded Systems

ABSTRACT

Nowadays we are surrounded by devices containing hardware and software components. These devices support a wide spectrum of different domains, such as telecommunication, avionics, automobile, and others. They are found anywhere, and so they are called Embedded Systems, as they are information processing systems embedded into enclosing products, where the processing system is not the main functionality of the product. The ever growing complexity in modern embedded systems requires the utilization of more components to implement the functions of a single system. Such an increasing functionality leads to a growth in the design complexity, which must be managed properly, because besides stringent requirements regarding power, performance and cost, also time-to-market hinders the design of embedded systems. Design Space Exploration (DSE) is the systematic generation and evaluation of design alternatives, in order to optimize system properties and fulfill requirements.

In embedded system development, specifically in Platform-Based Design (PBD), current DSE methodologies are challenged by the increasing number of design decisions at multiple abstraction levels, which leads to an explosion of combination of alternatives. However, only a reduced number of these alternatives leads to feasible designs, which fulfill non-functional requirements. Moreover, each design decision influences subsequent decisions and system properties, hence there are inter-dependencies between design decisions, so that the order decisions are made matters to the final system implementation. Furthermore, there is a trade-off between heuristics for specific DSE, which improves the optimization results, and global optimizers, which improve the flexibility to be applied in different DSE scenarios.

In order to overcome the identified challenges an MDE methodology for DSE is proposed. For this methodology a DSE Domain metamodel is proposed to represent relevant DSE concepts such as design space, design alternatives, evaluation method, constraints and others. Moreover, this metamodel represents different DSE problems, improving the flexibility of the proposed framework. Model transformations are used to implement DSE rules, which are used to constrain, guide, and generate design candidates. Focusing on the mapping between layers in a PBD approach, a novel design space abstraction is provided to represent multiple design decisions involved in the mapping as a single DSE problem. This abstraction is based on Categorical Graph Product, decoupling the exploration algorithm from the design space and being well suited to be implemented in automatic exploration tools. Upon this abstraction, the DSE method can benefit from the MDE methodology, opening new optimization opportunities, and improving the DSE integration into the development process and specification of DSE scenarios.

Keywords: embedded systems, design space exploration, model-driven engineering, UML, platform-based design.

Metodologia de Engenharia Dirigida por Modelos para Exploração do Espaço de Projeto de Sistemas Embarcados

RESUMO

Atualmente dispositivos contendo hardware e software são encontrados em todos os lugares. Estes dispositivos prestam suporte a uma varieadade de domínios, como telecomunicações, automotivo e outros. Eles são chamados “sistemas embarcados”, pois são sistemas de processamento montados dentro de produtos, cujo sistema de processamento não faz parte da funcionalidade principal do produto. O acréscimo de funções nestes sistemas implica no aumento da complexidade de seu projeto, o qual deve ser adequadamente gerenciado, pois além de requisitos rigorosos em relação à dissipação de potência, desempenho e custos, a pressão sobre o prazo para introdução de um produto no mercado também dificulta seu projeto. Exploração do espaço de projeto (DSE) é a atividade sistemática de gerar e avaliar alternativas de projetos, com o objetivo de otimizar suas propriedades.

No desenvolvimento de sistemas embarcados, especialmente em Projeto Baseado em Plataformas (PBD), metodologias de DSE atuais são desafiadas pelo crescimento do número de decisões de projeto, o qual implica na explosão da combinação de alternativas. Porém, somente algumas destas resultam em projetos que atedem os requisitos não-funcionais. Além disso, as decisões influenciam umas às outras, de forma que a ordem em que estas são tomadas alteram a implementação final do sistema. Outro desafio é o balanço entre flexibilidade da metodologia e seu desempenho, pois métodos globais de otimização são flexíveis, mas apresentam baixo desempenho. Já heurísticas especialmente desenvolvidas para o cenário de DSE em questão apresentam melhor desempenho, porém dificilmente são aplicáveis a diferentes cenários.

Com o intuito de superar os desafios é proposta uma metodologia de projeto dirigido por modelos (MDE) adquada para DSE. Um metamodelo do domínio de DSE é definido para representar conceitos como espaço de projeto, métodos de avaliação e restrições. O metamodelo também representa diferentes problemas de DSE aprimorando a flexibilidade da metodologia. Regras de transformações de modelos implementam as regras de DSE, as quais são utilizadas para restringir e guiar a geração de projetos alternativos. Restringindo-se ao mapeamento entre camadas no PBD é proposta uma abstração para representar o espaço de projeto. Ela representa múltiplas decisões de projeto envolvidas no mapeamento como um único problema de DSE. Esta representação é adequada para a implementação em ferramentas automática de DSE e pode beneficiar o processo de DSE com uma abordagem de MDE, aprimorando a especificação de cenários de DSE e sua integração no processo de desenvolvimento.

Keywords: sistemas embarcados, exploração espaço de projeto, engenharia dirigida por modelos, UML, projeto baseado em plataformas.

Modellgetriebene Entwicklungsmethodik für die Entwurfsraumexploration von Eingebetteten Systeme

ZUSAMMENFASSUNG

Heutzutage sind wir von Geräten umgeben, die sowohl Hardware wie auch Software-Komponenten beinhalten. Diese Geräte unterstützen ein breites Spektrum an verschiedenen Domänen, so zum Beispiel Telekommunikation, Luftfahrt, Automobil und andere. Derartige Systeme sind überall aufzufinden und werden als Eingebettete Systeme bezeichnet, da sie zur Informationsverarbeitung in andere Produkte eingebettet werden, wobei die Informationsverarbeitung des eingebetteten Systems jedoch nicht die bezeichnende Funktion des Produkts ist. Die ständig zunehmende Komplexität moderner eingebettete Systeme erfordert die Verwendung von mehreren Komponenten um die Funktionen von einem einzelnen System zu implementieren. Eine solche Steigerung der Funktionalität führt jedoch ebenfalls zu einem Wachstum in der Entwurfs-Komplexität, die korrekt und effizient beherrscht werden muss. Neben hohen Anforderungen bezüglich Leistungsaufnahme, Performanz und Kosten hat auch Time-to-Market-Anforderungen großen Einfluss auf den Entwurf von Eingebetteten Systemen. Design Space Exploration (DSE) beschreibt die systematische Erzeugung und Auswertung von Entwurfs-Alternativen, um die Systemleistung zu optimieren und den gestellten Anforderungen an das System zu genügen.

Bei der Entwicklung von Eingebetteten Systemen, speziell beim Platform-Based Design (PBD) führt die zunehmende Anzahl von Design-Entscheidungen auf mehreren Abstraktionsebenen zu einer Explosion der möglichen Kombinationen von Alternativen, was auch für aktuelle DSE Methoden eine Herausforderung darstellt. Jedoch vermag üblicherweise nur eine begrenzte Anzahl von Entwurfs-Alternativen die zusätzlich formulierten nicht-funktionalen Anforderungen zu erfüllen. Darüber hinaus beeinflusst jede Entwurfs-Entscheidung weitere Entscheidungen und damit die resultierenden Systemeigenschaften. Somit existieren Abhängigkeiten zwischen Entwurfs-Entscheidungen und deren Reihenfolge auf dem Weg zur Implementierung des Systems. Zudem gilt es zwischen einer spezifischen Heuristik für eine bestimmte DSE, welche zu verbesserten Optimierungsresultaten führt, sowie globalen Verfahren, welche ihrerseits zur Flexibilität hinsichtlich der Anwendbarkeit bei verschiedenen DSE Szenarien beitragen, abzuwägen.

Um die genannten Herausforderungen zu lösen wird eine Modellgetriebene Entwicklung (englisch Model-Driven Engineering, kurz MDE) Methodik für DSE vorgeschlagen. Für diese Methodik wird ein DSE-Domain-Metamodell eingeführt um relevante DSE-Konzepte wie Entwurfsraum, Entwurfs-Alternativen, Auswertungs- und Bewertungsverfahren, Einschränkungen und andere abzubilden. Darüber hinaus modelliert das Metamodell verschiedenen DSE-Frage- stellungen, was zur Verbesserung der Flexibilität der vorgeschlagenen Methodik beiträgt. Zur Umsetzung von DSE-Regeln, welche zur Steuerung, Einschränkung und Generierung der Ent- wurfs-Alternativen genutzt werden, finden Modell-zu-Modell-Transformationen Anwendung.

Durch die Fokussierung auf die Zuordnung zwischen den Schichten in einem PBD-

Ansatz wird eine neuartige Entwurfsraumabstraktion eingeführt, um multiple Entwurfsentscheidungen als singuläres DSE Problem zu repräsentieren. Diese auf dem Categorial Graph Product aufbauende Abstraktion entkoppelt den Explorations-Algorithmus vom Entwurfsraum und ist für Umsetzung in automatisierte Werkzeugketten gut geeignet. Basierend auf dieser Abstraktion profitiert die DSE-Methode durch die eingeführte MDE-Methodik als solche und ermöglicht nunmehr neue Optimierungsmöglichkeiten sowie die Verbesserung der Integration von DSE in Entwicklungsprozesse und die Spezifikation von DSE-Szenarien.

Keywords: Eingebetteten Systeme, Entwurfsraumexploration, Modellgetriebene Entwicklung, UML, Platform-based Design.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ACO	Ant Colony Optimization
API	Application Programming Interface
ATL	Atlas Transformation Language
AWM	Atlas Model Weaving
CASE	Computer Aided Software Engineering
CDFG	Control and Data Flow Graph
CGP	Categorical Graph Product
CPACO-MO	Crowding Population-based Ant Colony Optimization for Multi-Objective
CPU	Central Processing Unit
DSE	Design Space Exploration
DSED	Design Space Exploration Domain
DSL	Domain Specific Language
DSMDET	Domain Specific Model-Driven Engineering Tools
DSML	Domain Specific Modeling Language
E3S	Embedded System Synthesis Benchmarks Suite
EBNF	Extended Backus-Naur Form
EEMBC	Embedded Microprocessor Benchmark Consortium
EMF	Eclipse Modeling Framework
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FR	Functional Requirement
ILP	Integer Linear Programming
IP	Intellectual Property
ISA	Instruction Set Architecture
KPN	Kahn Process Network
LTA	Labeled Time Automata

MARTE	UML profile for Modeling and Analysis of Real-Time and Embedded System
MDA	Model-Driven Architecture
MDD	Multi-valued Decision Diagrams
MDE	Model-Driven Engineering
MIC	Model Integrated Computing
MoC	Model-of Computation
MODES	MOdel-Driven engineering for Embedded System
MOF	Meta-object Facility Specification
MPSoC	Multi-Processors System-on-Chip
NFR	Non-Functional Requirement
NoC	Network-on-Chip
OBDD	Ordered Binary Decision Diagram
OCL	Object Constraint Language
OMG	Object Management Group
PBD	Platform-Based Design
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query, View and Transformation Language
RTL	Register Transfer Level
SAT	Satisfiability Problem
SoC	System-on-Chip
SysML	Systems Modeling Language
TGFF	Task Graph for Free
TLM	Transaction Level Modeling
UML	Unified Modeling Language
VLSI	Very Large Scale Integration
WCE	Worst Case Execution
XML	Extensible Markup Language
YAPI	Y-Chart Application Programming Interface
YML	Y-Chart Modeling Language

LIST OF SYMBOLS

G	is a directed graph $G = \langle V_G, E_G, \delta_s, \delta_t \rangle$.
$G(E)$	is the set of edges $E_G = e_0, e_1, \ldots, e_n \in G$, also denoted by $G(E)$.
E_G	is the set of edges $E_G = e_0, e_1, \ldots, e_n \in G$, also denoted by $G(E)$.
V_G	is the set of vertices $V_G = v_0, v_1, \ldots, v_n \in G$, also denoted by $G(V)$.
$G(V)$	is the set of vertices $V_G = v_0, v_1, \ldots, v_n \in G$, also denoted by $G(V)$.
$u, v, u\prime, v\prime$	are variable used do represent a vertices.
$e, e\prime$	are variables used to represent a edges.
δ_s	is the function $\delta_s : E_G \rightarrow V_G$, which returns the source vertices of an edge.
δ_t	is the function $\delta_t : E_G \rightarrow V_G$, which returns the target vertices of an edge.
Mo	is a model $Mo = \langle G, \omega, \mu \rangle$.
ω	is reference model of Mo, associated to a graph $G_\omega = \langle V_\omega, E_\omega, \delta_s, \delta_t \rangle$.
μ	is a function $\mu : N_G \cup E_G \rightarrow N_\omega$, which associates elements (vertices and edges) of G to nodes of G_ω (metamodel).
$DSED$	is a Design Space Exploration Domain model.
$D(Dg)$	is the set of design graphs.
$Sl(d)$	is the set of solutions, such that design decisions $d \in Sl$ solve a problem P.
Rm	are the required metric used to qualify solution $s \in Sl$.
Ro	is the set of required objectives to be optimized, such that $Ro \subseteq Rm$.
Rg	is the set of required metrics used to guide the DSE, such that $Rg \subset Rm$.
Cs	is a set of constraints to be applied, when solving a DSE problem P.
$D\prime(Dg)$	is a subset $D\prime \subseteq D$ of design graphs which are required to model a DSE Problem P.
P	is the set of DSE problems to be solved for a $D\prime(Dg)$.
Sc	is a DSE scenario consisting of Dg and P, which can be solve for all $p \in P$.
So	is the available solver methods to optimize the problems $p \in Sc$.
ϕ	is a function $\phi : P \rightarrow Sl$ that generates the solutions $s \in Sl$.
Ev	is the available evaluation methods to qualify solutions $s \in Sl$.

γ	is a function $\gamma : Sl \rightarrow Co$ that calculates the cost $c \in Co$ for a solution $s \in Sl$.
Pa	are parameters used to configure evaluation and solver methods.
M	is a set of metrics, $m \in M$ and $\exists\ ev \in Ev \mid m$ can be provided.
c	is a variable used to represent a cost $c \in Co$ for element in Sc.
c_{max}	is a upper bound for a value that can be assigned to c.
c_{min}	is a lower bound for a value that can be assigned to c.
$c_{assigned}$	is the value that must be assigned to c.
P_{conf}	is a DSE Configuration Problem, which extends the DSED model.
Do	is the variable domain defined by the interval $[u; l]$, such that $Do \subset \mathbb{Z}$ or $Do \subset \mathbb{R}$.
Pr	is the set of configurable properties associated to a design elements $v \mid v \in Dg\,(V)$.
$Sl\,(Pr)$	is the set of configurations that solve the problem P_{conf}.
pr_{lower}	is the lower bound for a property $pr \in Pr$.
pr_{upper}	is the upper bound for a property $pr \in Pr$.
$pr_{assigned}$	is the value that must be assigned to pr.
P_{const}	is a DSE Construction Problem, which extends the DSED model.
E_m	is a set of mandatory edges.
E_o	is a set of optional edges.
E_i	is a set of cross-tree implies edges.
E_x	is the set of cross-tree excludes edges.
G_o	is the set of edges $e \in Dg\,(E)$ representing an or-group.
G_m	is the set of edges $e \in Dg\,(E)$ representing an mutex-group.
G_x	is the set of edges $e \in Dg\,(E)$ representing a xor-group.
Sd	is a label associated to building block $v \in Dg\,(V)$ which identify the selected block to constitute the system under construction.
ϵ	is a labeling function $\epsilon : Sl \rightarrow Dg\,(V)$ to annotate the selected block to constitute the system under construction.
$Sl\,(Sd)$	is the set of building blocs that solve the problem P_{conf}.
$P_{mapping}$	is a DSE Mapping Problem, which extends the DSED model.
Mp	is a pair $\langle u, v\rangle$, which represents the mapping between vertex u into vertex v of different design graphs.
$Sl\,(Mp)$	is the set of mappings which solve the problem $P_{mapping}$, and is subject of the constraint set Cs.
P_{sched}	is a DSE Scheduling Problem, which extends the DSED model.

Dg_t	is a task graph.
Dg_p	is a processor graph.
Dg_r	is a resource graph.
T_i	is the period of task τ_i.
r_i	is the release time of task τ_i.
C_i	is the computational time of τ_i.
d_i	is the absolute deadline of task τ_i.
D_i	is the relative deadline of task τ_i.
s_i	is the start time of task τ_i.
f_i	is the finish time of task τ_i.
St	is the tuple of properties that configures a task for the DSE Scheduling Problem.
$Sl\,(St)$	is the set of St, which solve the DSE Scheduling Problem.
$\tau_i \prec \tau_j$	determines precedence between between tasks τ_i and τ_j.
$s_{assigned}$	is the start time to be assigned to a task.
$f_{assigned}$	is the finish time to be assigned to a task.
Occ_{max}	is the maximum occupation of a resource, so that a schedulability test can be satisfied.
$\otimes$	is the categorical graph product operator.
Ds	is a design space graph.
π_n	is a projection function, which returns the graph $G_n \in G_i \otimes G_{i+1} \cdots \otimes \cdots Gn$.
ϕ_{ij}	is the current amount of pheromone in the edge $\langle i, j \rangle$.
$\Delta\phi^s_{ij}$	is the amount of pheromone value to adjust ϕ_{ij} by the ant s.
s_{rank}	is an integer rank assigned by the *Fast Non-dominated Sort* procedure.
p_{ij}	is the probability of ant k selecting the edge connecting vertex i and j.
α	is the magnitude of pheromone influence on probabilistic decision.
h	is the number of objectives.
η^d_{ij}	is the heuristic value for edge connecting vertex i and j.
β	is the magnitude of heuristic influence on probabilistic decision.
λ	is the heuristic exponent weighting factor.
N^k_i	is the set of design alternatives that ant k has not yet visited.
V_{shared}	is the number of shared vertices between s^{new}_j and s_k.
Cd	is a candidate design graph, such that $Cd \subseteq Ds$.
Q	is a first-in first-out queue.
A	is a list of alternative design decisions.

LIST OF ALGORITHMS

LIST OF LISTINGS

CONTENTS

1 INTRODUCTION

This chapter gives an introduction into this thesis. It starts presenting a motivation and describes the purpose. Following it identifies the contribution and present the structure of this thesis.

1.1 Motivation

Nowadays we are surrounded by devices containing hardware and software components. These devices support a wide spectrum of different domains, such as telecommunication, avionics, automobile, space, military, medical care and others. They are inserted into our day-by-day lives, in the cell phone, in the car as controllers for multiple subsystems (e.g. Ant-Block System - ABS, Electronic Power Steering - EPS, etc), electronic toys, blood pressure measurement systems, etc. In short, they are found anywhere, and so they are called Embedded Systems, as they are information processing systems embedded into enclosing products, where the processing system is not the main functionality of the product (MARWEDEL, 2003).

The ever growing complexity in modern embedded systems requires the utilization of more hardware and software components to implement the functions incorporated into a single system. Such an increasing functionality leads to a growth in the design complexity, which must be managed properly, because besides stringent requirements regarding power, performance and cost, also time-to-market hinders the design of embedded systems. The presence of multiple design decisions with stringent and often conflicting requirements unveils complex design alternatives, which the design team must evaluate under reduced time-to-market. Such a systematical generation and evaluation of design alternatives is named Design Space Exploration (DSE). The purpose of DSE is to optimize one or more system properties according to some quality metrics. Each different alternative corresponds to a trade-off regarding design requirements and constraints. From the best alternatives an engineer selects one of them to follow the next steps of a development process.

In the context of embedded systems, Platform-Based Design (PBD) (FERRARI; SANGIOVANNI-VINCENTELLI, 1999) is a development process model largely employed, because PBD maximizes the reuse of pre-designed components and achieves the best customization of the design platform concerning system requirements. PBD's strategy is to apply a layered design approach and reuse components from a large library of components. In this strategy a DSE step is required in order to optimize mapping between layers, thus building a link from the initial specification until the final implementation. The increasing number of reused components, together with the complex mapping between layers, reinforces the need for adequate DSE methods, which should enable the automation and optimization of design activities. Although the PBD approach is very valuable to the design of embedded system, developing applications for the existing complex platforms is a hard task. Furthermore, developing new platforms from scratch is a big bet for companies (GOERING, 2002). Moreover, it is difficult to map multiple layers, as well as get benefits from the optimization potential at higher abstraction layers (SANGIOVANNI-VINCENTELLI, 2007).

In embedded system development processes, specifically in PBD, current DSE methodologies are challenged by the increasing number of design decisions at multiple abstraction levels, which leads to an explosion of combination of alternatives. However, only a reduced number of these alternatives leads to feasible designs, which fulfill non-functional requirements (NFRs). Moreover, each design decision influences subsequent decisions and system properties, hence there are inter-dependencies between design decisions, so that the order decisions are made matters to the final system implementation. Furthermore, there is a trade-off between heuristics for specific DSE, which improves the optimization results, and global optimizers, which improve the flexibility to be applied in different DSE scenarios.

In order to overcome the difficulty in rising the abstraction level and simultaneously improve the refinement of the design from the initial specification until the final system through an automated DSE process, the DSE methodology proposed in this thesis advocates the application of Model-Driven Engineering (MDE) (KENT, 2002) techniques. Research efforts are strongly supporting MDE for embedded systems (VANDERPERREN; DEHAENE, 2006), because MDE can improve the complexity management, by rising the abstraction, and also provides mechanisms to improve automation and reusability of artifacts as models. MDE is also adequate to represent and handle DSE problems, because it provides abstractions and tools to handle PBD concepts such as orthogonalization of concerns, layer refinement, layer and mapping representation. Moreover, MDE plays an important role on architecture design by providing mechanisms to represent and map problem and solutions spaces, as highlighted in (HAAN, 2008):

> *"Model-Driven Engineering (MDE) in its essence is constructing a model of a problem space (e.g. a business process) and transform that model into a model of a solution space (e.g. a software system)." (HAAN, 2008)*

The purpose of this thesis is to improve the flexibility, reusability, and productivity in the DSE process. Specifically the methodology endeavors to integrate easily DSE methods into a development process. Moreover, it attempts to identify and represent different DSE problems in a concise and uniform way and provide a mechanism that allows an engineer to define DSE constraints according to the specificity of the problem to be solved. Another goal of this methodology is to automate development steps related to DSE, so that DSE artifacts can be generated from other development artifacts. Because the mapping of layers is the central issue in PBD approaches, it is a goal of this work to propose new abstractions to represent the design alternatives in the PBD context and a new method to generate candidates design based on the new proposed representation.

1.2 Contribution

1.2.1 Major Contributions

Definition of a DSE methodology as an MDE process: The increasing application of MDE as development technology for embedded systems requires alignment of the development process, which includes DSE, in order to completely exploit the productive gains promised by MDE. This means to go beyond the tool integration through transformation of input/output models between tools, but including engineering expertise in the (meta)models and transformation.

The DSE methodology was completely defined for an MDE process, allowing the integration of the DSE activities into a Domain Specific MDE process. First the methodology defines a DSE process, which is integrated in the development process by using automatic model transformation and model weaving, alleviating the development effort. Moreover, model weaving allows the orthogonal specification of development models, whose elements are woven with DSE model elements at the beginning of the DSE process. The process identifies methods, tools, and artifacts required, promoting an easy deployment of the methodology.

Exploiting MDE concepts in every step of DSE, a DSE Domain (DSED) metamodel is proposed to represent the important elements of a DSE process. Such elements represent available solver algorithms, available evaluation tools, and metrics to be optimized or used to guide the DSE process. Besides, the metamodel represents four different DSE problems identified from the studied literature. These problems are classified according to (SAXENA; KARSAI, 2011), namely construction, configuration, mapping, and scheduling.

Models conforming to the DSE Domain metamodel are handled in every step by model-to-model transformations, which extract required information from development models and fill the DSE models. Such model-to-model transformations are also used to implement DSE rules, which guide and prune the automatic DSE, providing the adequate mechanism to specify configurable, reusable and complex DSE rules.

The methodology was implemented using the *de facto* standards, such as ATL, ECORE, and UML, for model transformation, metamodeling and modeling respectively. Such standards are well supported by tools, such as the ones provided by the Eclipse Modeling Project, allowing easy application of the proposed methodology in real life projects.

Design Space representation for mapping in PBD: In PBD methodologies the development is strongly dependent on the mapping between layers to refine the system until the implementation. Because different design activities can be represented as mappings between graphs, which represent the mapping between layers, this work presents a method to represent the design space as a Categorical Graph Product (CGP) (WEICHSEL, 1962). The CGP method maps automatically multiple graphs, which are generated from development models, and expose element dependencies through all graphs. Such property is especially important, for example, in order to optimize the communication in different systems aspects (e.g. tasks, processors, and buses). Moreover, CGP improves the abstraction and represents multiple design decisions involved in the mapping as one single unified decision. Therefore the CGP is appropriate for representing simultaneous and interdependent design alternatives.

Stepwise alternative design generation: In the CGP-based representation of the design space, vertices of CGP represent design decisions, and edges define alternatives reachable from the last selected vertex. The proposed method to generate alternative designs based on the CGP induces a stepwise search in design space. This search is guided by constraints applied locally at vertex's adjacency as the search algorithm iterates on the graph resulting from CGP, in order to select the sub-graph which represents an alternative design. Such an approach avoids the enumeration of all possibilities, by removing alternative vertices that do not fulfill the specified constraints.

1.2.2 Minor Contributions

Domain Specific Model Driven Engineering Framework: The MOdel-Driven engineering for Embedded System (MODES) framework (NASCIMENTO; OLIVEIRA; WAGNER, 2007), consists of a set of metamodels and transformations to capture different views of embedded systems and provides support for integration of Domain Specific MDE tools. This framework was extended to include more metamodels and transformations to support the proposed methodology flow and automate some development tasks such as transformation from Unified Modeling Language (UML) to DSE domain model, Simulink models in text files to Simulink model in ECORE, extraction of data from UML, etc.

Analytic Evaluation Tool: A tool called System Performance Estimation with UML (SPEU) (OLIVEIRA et al., 2006) was implemented to provide quick evaluation of alternative designs suggested by the exploration tools. SPEU provides analytical estimates on physical system properties, which are directly obtained from system specification in UML, by using an implementation of the Implicit Path Enumeration method (LI; MALIK, 1995) and guided refinement of a symbolic instruction set.

Design Space Exploration Tool: In order to support the proposed methodology a tool called High-level Design Space Exploration (H-SPEX) (OLIVEIRA et al., 2008) was developed to orchestrate the DSE process. Moreover, this tool integrates an implementation of an optimization algorithms used to search in the design space and generate alternative designs.

Library of DSE rules: In the proposed methodology DSE constraints are model-to-model transformation rules, which guide and prune the available design space, in order to reduce the exploration time and ensure the feasibility of a candidate solution. Even if a design is feasible, it can be invalidated when checked against NFRs, which must be satisfied by the system. In this fashion, these rules avoid the violation of requirements such as task deadlines, maximum delays, and maximum energy consumption. Moreover, designs usually start with pre-defined design decisions and previously developed components, and the selected platform may impose restrictions, which an engineer must respect. Furthermore, an engineer, with his experience, may influence how the automated DSE process proceeds.

Observing the common constraints and rules used to implement DSE tools, this work provides a library of common DSE rules. This library reduces the effort during the DSE by reusing DSE rules and makes easy the adaptation of the DSE tool to be applied for different DSE scenarios. The library can also be extended by implementing additional rules by using standard transformation languages.

Automatic DSED model extraction from UML models: In order to improve the productivity during DSE process, a tool for automatic extraction of design graphs was implemented. It transform UML models into different design graphs in the DSED model. Because DSE is essentially a requirement-driven activity, it is expected that the requirements are defined in such a way it can be used to guide the DSE. In this fashion, this tool also extracted automatically constraints from UML models. These constraints are represented in the DSED model and implemented by the library of DSE rules, which bridges the UML and DSED models

1.3 Organization

The remainder of this thesis is organized as follows:

Chapter 2 presents an overview on embedded system development models and processes. It introduces the Y-chart model and PBD, which are the basis of the state-of-the-art DSE methodologies. The most common phases of embedded system development are present, as well as the context where DSE is performed.

Chapter 3 provides a background on DSE. It starts presenting a motivation and some challenges faced by DSE tools. After, it defines DSE and shows a general activity flow for DSE based on the studied literature. The following sections present the state-of-the-art methodologies for DSE, considering different views, such as language used for specification, application, platform, constraints modeling. It also surveys different evaluation methods and automatic exploration mechanisms. This chapter finishes with a discussion on the related work.

Chapter 4 introduces the main concepts of MDE. Following, a basic technological framework to support MDE is identified. This chapter also complements the Chapter 3 by presenting additional methodologies that explicitly apply MDE technologies for embedded system development and for DSE. A brief discussion on the studied methodologies ends this chapter.

Chapter 5 presents the proposed MDE methodology for DSE. The DSE methodology and process are defined. All steps of the methodology process are presented in detail, starting by the method for system modeling. Afterwards, the DSE domain metamodel representing the relevant DSE concepts is defined and the formal representation of four DSE problems are specified and mapped to the DSE Domain metamodel. Following, the method for weaving design models with DSE Domain models is presented. The specification of DSE rules to guide the DSE process are presented. It is also presented how optimization and evaluation methods can be integrated in the process. Finishing this chapter, a discussion provides a comparison of the proposed approach with others found in the literature.

Chapter 6 describes the proposal to improve the mapping in PBD methodologies. It starts defining the CGP. Then the CGP definition is used to build a design space abstraction, which is adequate to deal with the DSE challenges in PBD and improve the flexibility. The algorithms developed to generate design candidates considering the proposed design abstraction model are presented. Finishing this chapter developed and existing algorithms for generating design candidates are discussed.

Chapter 7 presents the implemented tools to support the methodology presented in Chapters 5 and 6. It also presents the required tools to support DSE considering a specific technology. Such tools and development flow are considered for the evaluation discussed in Chapter 8.

Chapter 8 shows the evaluation of the methods proposed in this work. First, the CGP method to represent the design space and the procedure to generate design alternatives are evaluated by applying the methods to synthetic graphs representing DSE problems with different complexities. It is also presented a realistic DSE scenario for a real-life application, in order to provide a complete example of the DSE flow presented in Chapter 5.

Chapter 9 summarizes the thesis and gives an outlook to further research.

2 DEVELOPMENT OF EMBEDDED SYSTEMS

Embedded Systems show special characteristics that do not appear in conventional ones, such as time, power, and size constraints, integration of software and hardware components (e.g. accelerators), as well as heterogeneous models of computation (MoC). As the embedded systems are not the final product, their costs are an important constraint too. Developing quality systems, within restricted time-to-market and affordable price, is a challenge. Therefore, many efforts are spent to define or improve development processes, methods, models and tools for embedded systems.

This chapter presents a short introduction on embedded system development. It starts presenting some relevant development process models. Afterwards it presents the common development phases found in many embedded systems development processes. Such background helps understanding of state-of-the-art of the DSE methodologies and identifying the context of DSE activities in a comprehensive development processes. The chapter finishes with a summary.

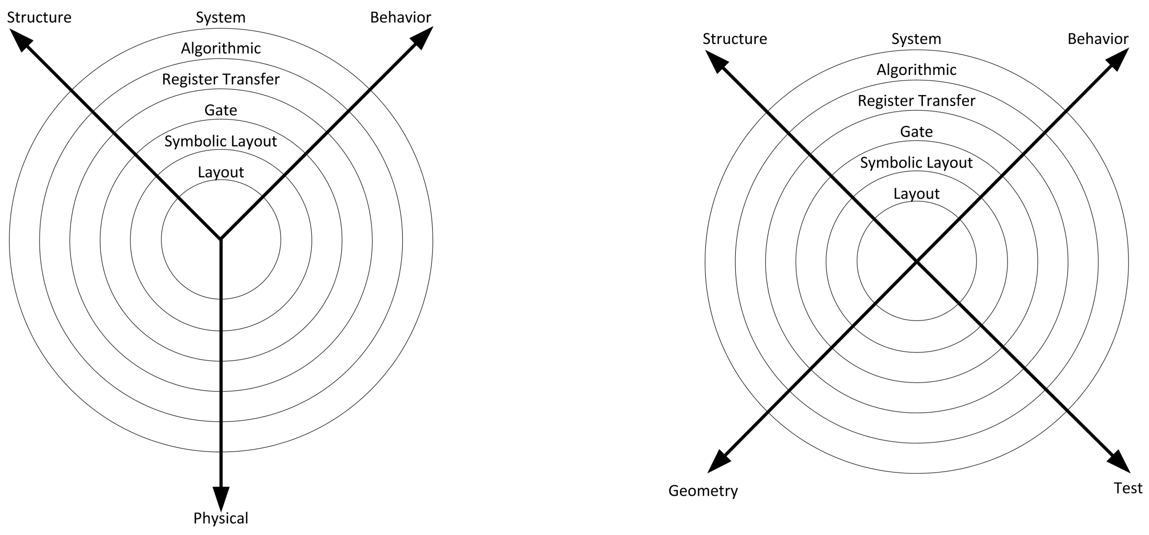
2.1 Development Process Models

A development process model (or development model for short) is an abstraction of a development process. It provides a particular perspective and abstraction level, helping the understanding of a development process (SOMMERVILLE, 2004). Three development models are specially studied in software engineering, namely the waterfall, the evolutionary, and the component-based software engineering (SOMMERVILLE, 2004). A development process may combine more than one model, which usually happens during complex system development. In embedded systems, three development models are highlighted, first the Gajski's Y-Chart (GAJSKI et al., 1994), later Kienhuis' Y-Chart (KIENHUIS et al., 1997) and PBD (FERRARI; SANGIOVANNI-VINCENTELLI, 1999).

Gajski's Y-Chart proposed a system model composed of behavioral, structural and physical representation. The behavioral representation provides a view of the design as a function of its input values and expired time. The structural representation defines a set of components and their connections. The physical representation adds physical information about components on the model, such as spatial distribution, size, heat dissipation and position of input/output pins. Each of these views identifies an axis, which are divided in several different abstraction levels, namely Circuit, Logic, Register transfer, Algorithmic, and System. Later, the Rammig's X-Chart propose an extension of Gajski's Y-Chart, by including the test view (RAMMIG, 1989). Figure 2.1 illustrates both models.

Improvements in the synthesis tools and the increasing transistor integration allowed the production of Systems-on-Chip and enforced the reuse of hardware IP in large scale. Therefore the Gajski's Y-Chart was replaced by new proposals (VAHID; GIVARGIS,

Figure 2.1: Gajski's Y-chart (left) and Rammig's X-Chart (right).



2001).

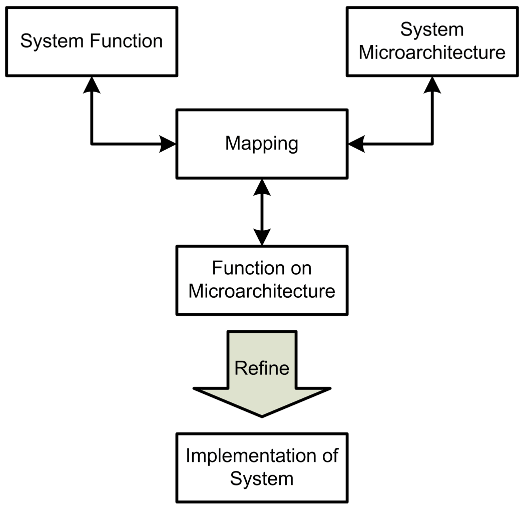
One of such proposals is a new Y-Chart (KIENHUIS et al., 1997). It identifies three key concepts, which now is the base for the embedded systems development: Architecture, Application and Mapping. The Architecture describes a hardware architecture composed by programmable processing elements (processors), buses and memories as parameterized templates. The Application is a behavioral description to be executed in a defined architecture. The Mapping represents the map from Application to Architecture, identifying which processor must execute each piece of software and where the software must be stored. Independently, another Y-Chart was proposed in the context of POLIS methodology (BALARIN et al., 1997), which is similar to Kienhuis' proposal.

PBD (FERRARI; SANGIOVANNI-VINCENTELLI, 1999) is a development model consisting of the refinement of a model at the highest abstraction level to meet an abstraction layer at a lower level. Such abstraction layer is named platform and defined to be *"a library of components that can be assembled to generate a design at that level of abstraction."* (SANGIOVANNI-VINCENTELLI, 2007).

The meet-in-the-middle approach advocated by PBD combines the refinement of models, by mapping it into an instance of platform and propagating constraints as in a top-down manner, with the bottom-up flow, by building up a platform from selected components available in the library. In this approach, a DSE step is required in order to optimize each mapping between layers, thus building a link from the initial specification to the final implementation. Such an approach includes the co-design of software and hardware, furthers the reuse of components and favors the use of higher levels of abstraction. However, these benefits depend on the establishment of the number, location and components of platform for a specific project. These trade-offs influence the design space to be explored and the accuracy of estimations to evaluate each map.

An improved proposal based on the previous two Y-Charts is presented in (KEUTZER et al., 2000), which combines Y-Chart with PBD. The Keutzer's model introduced more concepts of software development and strong focus on software and hardware reuse. The System Function (similar to the Application in the Kienhuis' model) describes the system behavior and must not include any architectural implication, e.g. the definition of what must be hardware or software. The Microarchitecture (similar to the Architecture in the Kienhuis' model) is the set of components used to implement a function. Although Keutzer states that the most important microarchitecture for embedded systems design consists of microprocessors, peripheral, dedicated logic blocs and memories, his definition left open space for other kinds of microarchitectures, including physical or abstract components. As the definition of microarchitecture is wider in Keutzer's model, the mapping could be more than the definition of which part of the function must be executed in each processor (as defined in the Kienhuis' model), so that the mapping could define many different design decisions, such as the definition of software and hardware, computation models and interconnection. Therefore, mapping is a refinement step, which may be performed interactively, and is required to determine the performance and cost of a system. It is highlighted that the more fixed the microarchitecture is, the easier is the mapping process. However, it limits the design alternatives, thus the design optimality. Figure 2.2 illustrates the Keutzer's Y-Chart development model. Since the pubication of the Y-Chart, in 1997 by Kienhuis, Y-Chart is interpreted in many different ways. Henceforth, this text refers to Keutzer's Y-Chart development model.

Figure 2.2: Keutzer's Y-chart.



2.2 Development Processes

A development process is the set of activities systematically performed to transform requirements into the products. A development process must provide well defined activities, containing a systematic view and traceable refinement. Moreover, the existence of tools to automate or support the development activities is important. A list of common activities can be identified in different development processes of embedded systems, such as RUP-SE (RATIONAL, 2002), ROPES (DOUGLASS, 2002), HASoC (GREEN; EDWARDS, 2002), SHE (GEILEN, 2001), MIDAS (FERNANDES; MACHADO; SANTOS, 2000), and Ptolemy (KALAVADE; LEE, 1992). These activities are requirements engineering, analysis and design, implementation/synthesis, verification, and test.

2.2.1 Requirements Engineering

During requirements engineering the services and operational constraints of a systems are established according to stakeholders (e.g. development team and customers). The output of this process are requirements, which can be defined in different levels of detail, ranging from abstract textual description in natural language to a formal definition. In (SOMMERVILLE, 2004) two requirement levels are distinguished:

- *User requirements* are statements in natural languages and diagrams, to describe expected services and constraints under which the system must operate.
- *System requirements* are precise descriptions of system's functions, services and constraints. These requirements contain metrics and units to qualify the expected system. It is also called functional specification and is used as a contract between the customers and developers.

Besides the level of detail, the requirements are also classified in (SOMMERVILLE, 2004):

Functional Requirements (FR) define which services a system must provide and how it must behave or not in some particular situation.

Non-Functional Requirement(NFR) are constraints on services or functions. They can also be constraints on processes and methods used to develop, forcing standards or tools to be used. Moreover, NFR may arise from users' special needs, budget constraints, technological availability, quality characteristics and others. In embedded system development typical NFRs to be met are the resource and timing constraints.

2.2.2 System Analysis and Architectural Design

The analysis activity defines what the system is and what it should do. The output of this activity is an analysis model. The analysis model represents concepts from the problem domain (e.g. wheels, steer, and transmission in automotive systems), for which the FRs and NFRs are defined. As much as possible the analysis model specification should have no architectural implications, which is left to the refinement process. During this refinement, the analysis model should be mapped to some architecture in the solution domain (e.g. scheduler, processors, classes, and objects in an embedded system), represented by the design model. During the design specification, computational and engineering issues (e.g. scheduling, memory, communication mechanism, processors, instructions, and others) arise as the relevant problems. In order to find the appropriate software and hardware architecture, different design activities must be performed and decisions between alternative designs must be taken. A non exhaustive list of design activities found in the literature (BERGER, 2001; JERRAYA et al., 2003; MARWEDEL, 2003; VAHID; GIVARGIS, 2001) is highlighted and some references to methods which implement or describe such activities are given:

- *Responsibility distribution and interaction between components* - This activity identifies independent portions of the system, whose functionality can be encapsulated and made available through ports and interfaces. These portions are called capsules (GEILEN, 2001) or processes (DOUGLASS, 2002). In order to identify these capsules it is important to identify how the components interact with each other, which functions will be provided and the required data for each component. Some methods are described in ROPES (DOUGLASS, 2002), MIDAS (FERNANDES; MACHADO; SANTOS, 2000) and SHE (GEILEN, 2001);
- *Algorithm and data structure selection* - This activity selects, among different algorithms and data structures, which ones will be used to implement a specific function

(MATTOS et al., 2004). It is important to highlight that the quality of an algorithm depends on its implementation as software or hardware (GRATTAN; STITT; VAHID, 2002). The selection of scheduling algorithms, which impacts on the final system characteristics (BECKER; WEHRMEISTER; PEREIRA, 2004), is an example of such an activity;

- *Hardware-Software partitioning* - Identifies the ideal portion of hardware and software to implement the system's function. The software portion requires a processing unit hardware. Compared to a hardware implementation, the software is more flexible, though it may consume more energy and processing time. Some recognized methods for partitioning are described in (KALAVADE; LEE, 1992; VAHID; GAJSKI; GONG, 1994; GRATTAN; STITT; VAHID, 2002). Notable two approaches are identified. One advocates that the partitioning must be defined as early as possible and the software and hardware are developed separated. This approach is recommended in the processes ROPES (DOUGLASS, 2002), COMET (GOMAA, 2000) and RUP-SE (RATIONAL, 2002). The second approach recommends the co-design of hardware and software, postponing the partition to as late as possible, when specialized algorithms perform automatic partition and synthesis. Such an approach is described in the processes HASoC (GREEN; EDWARDS, 2002), MIDAS (FERNANDES; MACHADO; SANTOS, 2000) and SHE (GEILEN, 2001);

- *Communication mechanisms and structures* - The system components may require communication between different interfaces, such as hardware-hardware, software-software or hardware-software. This communication can be implemented in different way, considering protocols, media and structures (e.g. buses and Network-on-Chip (NoC)). Each implementation offers trade-offs to be considered, in order to achieve an optimized system. There are methods able to generate interfaces automatically (LAHIRI; RAGHUNATHAN; DEY, 2000) and others to determine the protocols and configure the required parameters (ORTEGA; BORRIELLO, 1998);

- *Task mapping* - The specified system tasks (processes or capsules) must be mapped to processing units to be executed. This activity is related with task scheduling and influences directly the communication and real parallel processing in the systems. Automatic methods to perform task mapping are described in (BLICKLE; TEICH; THIELE, 1998; LIEVERSE et al., 2001). In (DOUGLASS, 2002) a manual heuristic is described and uses UML to specify the mapping, while in (KANGAS et al., 2006) the method called Koski receives UML models as input and generates the mapping automatically;

- *Hardware allocation* - The hardware constituting the system must be connected to a communication structure such as buses or a NoC as described in (BLICKLE; TEICH; THIELE, 1998; LAHIRI; RAGHUNATHAN; DEY, 2000);

- *Task scheduling* - The system tasks must be scheduled if they share some resources, such as processing units or a bus. The scheduling can be statically defined at design time, reducing overhead and optimizing some system characteristics (BLICKLE; TEICH; THIELE, 1998; KANGAS et al., 2006), dynamically defined at run-time (BECKER; WEHRMEISTER; PEREIRA, 2004), or in adaptive fashion (LU et al., 2000);

- *Voltage and frequency scaling* - The clock frequency and voltage are related to processing speed and power dissipation, as such should be adjusted to the system requirements. Design time heuristics to define the frequency and voltage are found in (VARATKAR; MARCULESCU, 2003), while (SHIN; CHOI, 1999; SHIN; CHOI; SAKURAI, 2000) describes methods to define them at run-time, adapting the systems to exploit the free scheduling time to save energy.

As a single problem (analysis model) can be solved by (mapped to) alternative solutions (design models), a design space is composed by all these available alternatives. During the design phase, an engineer searches manually or automatically in the design space for the best candidate design according to some design goals and constraints (e.g. measured by performance and energy consumption). Therefore, the design space is the set of all alternative designs for all required design activities, and DSE is the systematical method for searching and evaluating different candidates in the design space. Due to the different abstraction levels, design activities and DSE can be performed at each level of refinement, until achieving the final implemented system.

2.2.3 Implementation

Implementation is the phase where the design models are translated manually or automatically into software or hardware components. These components can be implemented at different abstraction levels, and coded in many different languages. Hardware components are usually written at the Register Transfer Level (RTL) using languages such as VHDL and Verilog, which are synthesized into gate-level models. The software implementation may range from application to real-time operating systems, and they are usually coded in C/C++ or Java. Synthesis tools and compilers can automate at least parts of the implementation effort, e.g. by generating glue code to integrate multiple components, configuring the operating system or translating the code to a lower abstraction level. Behavioral synthesis tools can generate hardware modules from behavioral descriptions, usually specified in C or derivate languages, such as Cynthesizer[1], CatapultC[2], and CyberWorkBench[3]. Some domain specific tools are able to generate a full system from design models described in Simulink[4] or UML (WEHRMEISTER et al., 2008).

2.2.4 Verification, Validation and Test

Verification, validation and test are activities used to check if an artifact meets the requirements based on some metrics. The validation determines the correctness of the final product with respect to the specified requirements. The validation can be accomplished by verifying the product at the end of each development step. Verification is the demonstration of consistency, completeness and correctness of functions, components or products by using different techniques, such as model checking (LARSEN; PETTERSSON; YI, 1997) and testing (HABIBI; TAHAR, 2004). Testing is the examination of the parts of, or the complete system, by executing them on sample data sets. These activities aim to insure the quality of the product in different development phases, in order to identify errors and improve the product (ADRION; BRANSTAD; CHERNIAVSKY, 1982).

[1]http://www.forteds.com
[2]http://www.mentor.com
[3]http://www.cyberworkbench.com
[4]http://www.mathworks.com/products/simulink-coder/index.html

2.3 Summary

This chapter presented a brief introduction to the embedded system development process. First it introduces different development models. This work is based on the Y-Chart development model combined with PBD as proposed in (KEUTZER et al., 2000). Afterwards, it described the common development phases and identified some design activities. An informal definition of DSE was presented. The context of DSE in a development process was identified. DSE can be earliest performed during the transformation of analysis to design. Then DSE can also applied to refine the design models at different abstraction levels until the final implementation is achieved.

3 DESIGN SPACE EXPLORATION FOR EMBEDDED SYSTEMS

The first section of this chapter defines DSE and presents a general activity flow. The following sections are organized based on the Y-Chart approach. As such, this chapter discusses about modeling languages, which is the requirement to model the application and platform represented in the Y-Chart. Following, this chapter presents the proposed solutions for modeling the application, the architectural platform, the mapping of the application to a platform and the requirements/constraints. Moreover, this chapter discusses methods for evaluation of design alternatives and presents the automatic mechanisms implemented to search for solutions within the design space. The chapter ends with a discussion on the current state of DSE methodologies.

3.1 Design Space Exploration

The great majority of current electronic products, such as mobile telephones, DVD players, microwave ovens, automotive system controls, and so far, contains an embedded system. During the development of these systems, a wide range of design alternatives arises from different design activities. For example, by observing a design example where 15 tasks executing on a 4-processor platform with 4 different voltage settings for each processor, over 100,000 design alternatives are found. The trends show that the number of embedded processors increases by 1.4/year and the amount of software doubles every 10 month (SEMICONDUCTORS, 2011). Dealing with this ever-growing challenge only by designer's expertise is not feasible, therefore new automated tools for DSE are required, in order to cope with the estimated growth of complexity.

Definition 3.1 (Design Space Exploration (DSE))**:**
It is the systematic process of generating and evaluating design alternatives, in order to find the best design solution for a development problem. In the context of PBD, DSE is also defined as the process of mapping from an application into a platform, such that design requirements and constraints are met (KEUTZER et al., 2000).

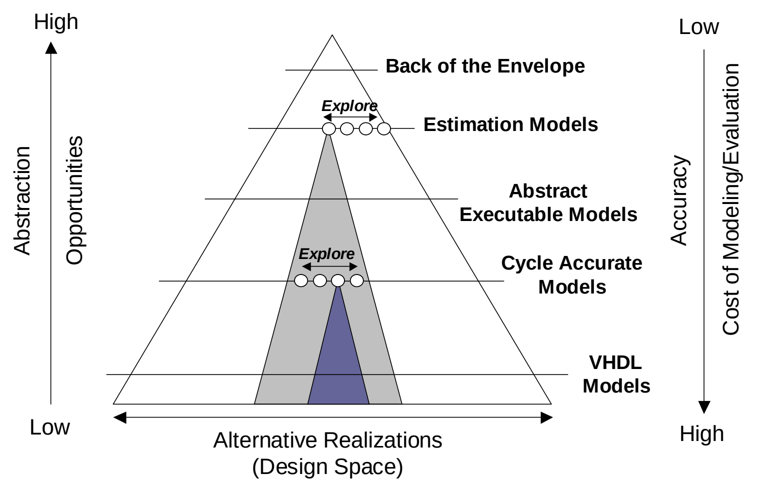
Definition 3.2 (Mapping)**:**
It corresponds to a set of design decisions required to refine a layer to the next abstraction level. The set of decisions depends on the degrees of freedom that are given by the mapped layers (e.g. application and platform in the Y-Chart approach) (KEUTZER et al., 2000).

Most tools for DSE are oriented towards a given subset of design decisions, such as task mapping and resource allocation, cache and pipeline size definition. The applied tool chain and the abstraction level define the supported set of design decisions and the design space. The main impact of the abstraction level regarding DSE concerns the flexibility and the easier evaluation of alternative designs. Higher abstraction levels reduce the evaluation effort and wide the design space, at the cost of reduction of estimation accuracy.

Definition 3.3 (Design Space)**:**
It is the set of all potential design alternatives that represent a solution for a DSE problem.

Providing both high abstraction level and an estimation method with an adequate balance between speed and accuracy is one of the main challenges for new DSE methods. Figure 3.1 illustrates this trade-off between abstraction and accuracy regarding the flexibility of DSE, based on the abstraction pyramid (KIENHUIS et al., 2002).

Figure 3.1: Abstraction Pyramid: The trade-off between abstraction and accuracy impacts on DSE.

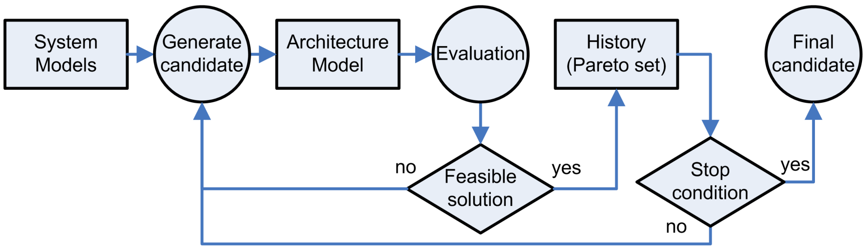


(KIENHUIS et al., 2002)

Independently of the abstraction level, the DSE process should be guided in order to avoid infeasible system solutions and optimize the system following some design quality criteria, usually named objectives. The first mean to guide the exploration is the specification of system NFR or design constraints. The NFRs are related to the quality of the system (performance, energy consumption, area and other) or related to constraints on how the functions should be implemented, such as the usage of specific technology, manufacturer, component versions, and etc. Beyond guiding the DSE process, the NFRs allow to compare each candidate solution properties with the system requirements. Thus, it is also necessary to provide methods and tools to extract system metrics in order to evaluate the alternative solutions and to verify their feasibility and quality. Usual metrics are related to performance, such as throughput or execution time; other are related to energy consumption and power dissipation; further, some metrics are related to the financial cost or to resource utilization, such as area, memory footprint or communication channel occupation.

Figure 3.2 presents a general DSE process (MARWEDEL, 2003; ASCIA et al., 2011). The inputs of this flow are the system models with open design decisions, such as the mapping of an application into a platform. NFRs and constraints can be provided as different models or embedded in these models. An exploration mechanism generates candidate designs using these models, concerned with the constraints provided, so that only feasible candidates are generated or after the generation a repair stage must be performed. Then the evaluation step extracts quality metrics from the candidate designs. After the evaluation, another repair step may be required, in order to remove undesired candidates from the process. A history registers the candidates, and from this history an engineer can select the final solution after the process has reached a stop condition.

Figure 3.2: General activity flow for design space exploration (MARWEDEL, 2003; ASCIA et al., 2011).



3.2 Modeling Languages

Application descriptions should have no architectural implications, thus the languages should raise the abstraction level in order to allow maximal flexibility in the DSE. Moreover, the application model language, used as interface between designers and Computer Aided Design (CAD) tools, should provide both, a precise semantic to be handled by computational algorithms and high expressiveness to allow suitable system specifications.

The utilization of an appropriate language for high-level design ensures the reduction of time-to-market, efficient communication between design team members and provides the mechanism to improve reusability and DSE process. Unfortunately, the appropriate language is not a consensus, due the fact that the selection of a language is strongly related to the system domain and the MoC used to represent the system. Some instances of high-level model languages are Simulink, used by DESERT (NEEMA et al., 2003) and (REYNERI et al., 2001); UML as proposed by Metropolis (UML AND PLATFORM-BASED DESIGN, 2003), DaRT (BONDé; DUMOULIN; DEKEYSER, 2005), and Koski (KANGAS et al., 2006); SysML, an extension of UML for system modeling, starting to be used for design space exploration in (PREVOSTINI; GANESAN, 2006). The work proposed in (SCIUTO et al., 2002), and StepN (PAULIN; PILKINGTON; BENSOUDANE, 2002) use SystemC. As the MILAN framework (BAKSHI; PRASANNA; LEDECZI, 2001) uses model translators, it allows modeling the application with different languages, including SystemC.

Using Simulink[1]-based environments the designer represents data-flow functions, usually found in digital signal processing applications such image, video or audio systems. The Simulink (functional block) language implicitly represents two MoCs, continuous time and discrete time. Simulink is able to represent periodic timing using a clock constructor. Modeling control algorithms is possible using Simulink and it is widely used in industry. However, there is not a suitable way to express timing requirements such as deadline, period, delay, jitter and other information usually required for instance to perform schedulability analysis. The possibility to express the behavior as equations and

[1]http://www.mathworks.de/products/simulink/

reusing pre-defined components is an important characteristic of Simulink. A disciplined Simulink model can be simulated using special design environments such as Matlab, and tools such as the Simulink Coder[2] are able to generate source code.

UML (OMG, 2007) is a highly expressive language to model a large range of systems. The expressiveness offered by UML is due to the fact that the model is separated into different aspects (structural and behavioral), and these aspects can be represented in different views such as classes, objects, components and deployment for structural aspects; and state, sequence, actions for behavioral aspects. Furthermore, profiles allow strong extensions on main UML constructs. Some of these profiles are standardized by Object Management Group (OMG)[3] and present important rules in embedded systems design, such as the UML profile for Modeling and Analysis of Real-Time and Embedded System (MARTE) (OMG, 2011), which is used to model resource concurrence and real-time requirements and includes hardware and software co-design. However, UML does not present precise semantics, which harms automatic simulation, verification and system code generation/synthesis.

SystemC (ACCELLERA, 2011) is a proper language to be used in hardware and software co-design, due to the possibility of hardware and software co-simulation and multiple MoC representation. Using the same representation for the hardware and software allows for a general design before partitioning, improving DSE. The imperative constructs inherited from C++ allow model execution and evaluation at different abstraction levels, and each design unit can represent a simple black box or a complex component specified at RTL-level. Moreover, it can be combined with graphical modeling languages, such as UML and Simulink, in order to improve the abstraction and the communication between design teams.

3.3 Modeling the Application

Beyond the languages used as interface with the designers to express the system structure, behavior or requirements, the utilization of appropriate internal representations is required to automate computational analysis. A DSE method should use a representation based on Models of Computation that are adequate for the target system domain and DSE activities. We can highlight some internal representation models such as Kahn Process Networks (KNP) (KAHN, 1974) used by ARTEMIS (PIMENTEL; ERBAS; POLSTRA, 2006), (DWIVEDI; KUMAR; BALAKRISHNAN, 2004) and SPADE (LIEVERSE et al., 2001); Signal Flow Graphs (LEDECZI et al., 2003) proposed in MILAN (BAKSHI; PRASANNA; LEDECZI, 2001; MOHANTY; PRASANNA, 2002); Control and Data Flow Graphs (CDFG) as used in H-SPEX (OLIVEIRA et al., 2007) and (ZIVKOVIC et al., 2003); other data-flow models used in DaRT (BONDé; DUMOULIN; DEKEYSER, 2005); Network of Concurrent Processes used by Metropolis (BALARIN et al., 2003); and Task Graphs (BLICKLE; TEICH; THIELE, 1998).

As proposed in ARTEMIS, the application is represented by a KPN, which is a MoC for modeling distributed systems by representing deterministic sequential processes that communicate through unbounded First-In-First-Out (FIFO) channels. The KPN is extracted from an application specified in C, C++ or Matlab language. It represents a set of concurrent processes. Each process performs sequential computation tasks and communicates with other processes through uni-directional channels, organized as unbounded

[2]http://www.mathworks.de/products/simulink-coder/index.html

[3]http://omg.org/

FIFOs. The KPN is deterministic, thus the result does not depend of the process execution order. Some extensions (COFFLAND; PIMENTEL, 2003) on the KPN model, such as the YAPI (KOCK et al., 2000), introduce non-determinism and allow modeling resource scheduling. Another extension adds timing information allowing time-dependent behavior analysis.

A CDFG is used in H-SPEX and Archer for DSE. In the CDFG, the data flow part specifies the concurrence presented in the application, whereas the control flow part determines the synchronization of data flow and dynamic path decision at run-time. The CDFG is a highly precise representation of the application, and usually it is used in code generation approaches. As suggested in (ZIVKOVIC et al., 2003), the CDFG is more complex than a Trace Driven representation to be handled and the simulation could be slower. However, static analysis as implemented in (OLIVEIRA et al., 2007) can speedup the system evaluation (see Section 3.7).

The main purpose of the MILAN framework is to represent embedded systems designed for signal processing. Thus, an enhanced hierarchical Signal Flow Graph model was proposed to represent the application. The hierarchical characteristic is proposed to handle system complexity. The meta-model, which defines the Signal Flow MoC, also supports explicit alternative implementations. Some additional formalisms have been included to extend the basic model for DSE purposes. The Signal Flow supports synchronous and asynchronous data-flow semantic, as well as the composition of both models. The proposed model is strongly typed and the model elements are parameterized to enable data-flow configuration. The formalism related to data-flow is tailored to support hardware implementation of the model on Field-Programmable Gate Array (FPGA) or Application-Specific Integrated Circuit (ASIC). In this model, each element is a hierarchical component, which contains well-defined interfaces represented by input and output ports. The signal flow specifies the partial order of execution through the communicating components. Object Constraint Language (OCL) (OMG, 2006) constraints are used to define the precise static semantic of data-flow connections.

The Network of Processes model adopted in Metropolis represents a set of communicating concurrent processes, where each process contains a sequential program, called thread. The communication is done through ports, and interfaces declare methods that processes can use through the ports. A media should implement the interface. The use of media and threads allow the separation of concerns, where computation and communication are specified separately. The process behavior is modeled as a sequence of events. The model supports non-deterministic execution of these events, thus constraints can be specified as logic formulas in order to restrict the execution.

3.4 Modeling the Architectural Platform

For any formal processing (analysis, mapping of an application, synthesis, etc.), the architectural platform must be explicitly and formally modeled. This model must express the architectural components of the platform - Central Processing Units (CPUs), global memories, hardware intellectual property (IP) components, communication infrastructure - as well as their specific properties. Examples are processor type and frequency and size of local memories, in the case of CPUs, or size and access time, in the case of global memories. Furthermore, the architecture can represent the software layer used to support the application, such as communication protocols, Operating System, Application Programming Interfaces (API) and other components. Usually the architectural model is based on previously designed components, such as ARM processors, AMBA buses, specific memory modules, and specific hardware or software IP components, such as an MPEG4 decoder, VxWorks OS and Dot Net Framework.

Most architectural modeling proposals focus on simulation and evaluation rather than on search mechanisms, hence not many details can be analyzed from the DSE process point of view. When architectural models are described in the context of search mechanisms, most proposals are strongly coupled with the search mechanism adopted. For example, some approaches use a list of alternatives, which represents architectural components, e.g. list of processors, list of buses, list of memories, which are used in candidate generation functions encoded in the search mechanism, such as Genetic Algorithms (AXELSSON, 1997; ASCIA; CATANIA; PALESI, 2004; DICK; JHA, 1998; ERBAS; ERBAS; PIMENTEL, 2003), Simulated Annealing (AXELSSON, 1997; KANGAS et al., 2006; OLIVEIRA et al., 2008) and Tabu Search (AXELSSON, 1997). Other approaches use a tree-based representation encoded in Ordered Binary Decision Diagram (OBDD) (NEEMA et al., 2003; MOHANTY; PRASANNA, 2002; PIMENTEL; ERBAS; POLSTRA, 2006).

Few works propose architectural representations independent of the search mechanism. There is proposal for an architectural graph (BLICKLE; TEICH; THIELE, 1998; OLIVEIRA et al., 2009; SCHLICHTER et al., 2006), which represents an architecture template. Such a template integrates in one graph multiple alternative architectures, into which an application can be mapped. Each architecture in the template represents the resources available, such as processors, buses and memories. Because this representation is more abstract, architectural information, such as cost, cost functions, and architectural parameters, are annotated directly in the graph (BLICKLE; TEICH; THIELE, 1998) or associated to another more detailed architectural model (SCHLICHTER et al., 2006; OLIVEIRA et al., 2009)

From the design point of view, there are many approaches for architecture modeling. Most of them represent an architecture by adopting different types of data-flow models, whose vertices represent processors and memories and edges represent buses (ASCIA; CATANIA; PALESI, 2004; BAKSHI; PRASANNA; LEDECZI, 2001; BALARIN et al., 1997, 2003; ERBAS; ERBAS; PIMENTEL, 2003; LIEVERSE et al., 2001; LEDECZI et al., 2003; MOHANTY; PRASANNA, 2002; NEEMA et al., 2003; PIMENTEL; ERBAS; POLSTRA, 2006; PIMENTEL, 2008). Other approaches represent only architectural resources with costs or parameterized functions associated to them, in order to perform static evaluation of the system (ASCIA et al., 2011; BLICKLE; TEICH; THIELE, 1998; BONTEMPI; KRUIJTZER, 2002; DICK; JHA, 1998; ERBAS; ERBAS; PIMENTEL, 2003; MATTOS et al., 2004; MOHANTY; PRASANNA, 2002; REYNERI et al., 2001).

3.5 Modeling the Mapping

The Y-Chart-based exploration approach allows separation of concerns and enhances the design space. However, to perform DSE, the application must be mapped onto the platform and the candidate solution must be represented in some fashion. Alternatives for reaching the final solution, resulting from this mapping, can be: i) intermediate layer representing an architecture in multiple abstraction levels (ERBAS; ERBAS; PIMENTEL, 2003; MIHAL et al., 2002; PIMENTEL; ERBAS; POLSTRA, 2006); ii) synchronization of traces from the application event with architecture events, as used in Metropolis (BALARIN et al., 1997); iii) explicit mapping from application elements into elements from architecture(BAKSHI; PRASANNA; LEDECZI, 2001; BLICKLE; TEICH; THIELE, 1998; LEDECZI et al., 2003; SCHLICHTER et al., 2006).

The mapping by successive refining architectural layers is implemented in the Sesame tool of the ARTEMIS (PIMENTEL; ERBAS; POLSTRA, 2006) project. The final refinement of the mapping layer with virtual processors represents the final solution. The event traces generated by Kahn processes of the application model are mapped onto virtual processors on the mapping layer. Both the Kahn's buffers and channels are also mapped onto size-restricted buffers and channels on the mapping layer. Sesame is able to automatically generate the mapping layer from the explicit mapping specification in the Y-Chart Modeling Language (YML) (COFFLAND; PIMENTEL, 2003). The mapping of virtual processors onto the architecture is specified in YML and can be freely adjustable in order to perform the DSE. For new mappings, modifications in the application is not required, thus the same application can be used for many architectures. When virtual processor models are refined, the mapping model layer is also refined through data-flow analysis and trace transformation. The application model is not changed during the refinement of the mapping layer and of the architectural model, so that it can be reused for alternative refinements. At the end of the process, the mapping layer and the architectural model together correspond to the final solution.

The Metropolis method uses a network, which encapsulates the network of the application events and the network of provided architectural services. Then, this new network is the mapping model, which synchronizes the events of the application with events of the architecture generated from decomposed services, for which the application components were mapped. The synchronization mechanism is made of constraints, which are written as logic formulas.

The most common alternative to perform the mapping and represent the final solution is by direct mapping between application and architecture elements. In the approach proposed in (BLICKLE; TEICH; THIELE, 1998), the explicit mapping is specified by a specification graph and "activations". The specification graph consists of compositions of dependence graphs, which represent the application and architecture models. In order to compose these graphs, mapping edges are manually specified, in such a way that they connect vertices from one graph into vertices of another one. The specification graph defines the design space and user constraints for allocation, binding and scheduling. The term "activations" specifies the set of nodes and edges active in the specification graph. A set of active nodes and edges represents an implementation, i.e., a candidate design.

A similar approach for direct mapping is implemented in MILAN (BAKSHI; PRASANNA; LEDECZI, 2001), where the mapping model specifies the available mappings of each data-flow component in the application model onto components at the resource model (architectural model). The mapping is represented by one or more references between components in both application and architecture models. These multiple references represent the alternative designs available during the DSE process. Additionally, values for performance and power can be attached in the references in order to guide the exploration algorithm. The mapping model also determines the channel, which implements the communication between data-flow tasks.

3.6 Modeling the Requirements and Constraints

Differently from the FR, which imply the explicit activities that a computing system must handle, NFR (which include design constraints) are somewhat hidden from the main application, though being critical for the correct system functioning. Hence, NFR encapsulate the system quality, its attributes and restrictions (FREITAS, 2007). The DSE process is motivated by attempting to find an optimal architecture to support the functional requirements and that architecture must meet the NFR. Therefore, applying an efficient requirements analysis method and best specification practices is imperative to successful DSE.

In the DSE methods, NFR are rarely argued, concerning the influence of requirement analysis and specification on the DSE. However, most of them present some mechanism to specify at least the constraints in order to prune the design space and avoid infeasible designs.

The most common and simple approach is annotating the constraints over the models. Metropolis, Koski (KANGAS et al., 2006) and Milan (BAKSHI; PRASANNA; LEDECZI, 2001) use OCL from OMG to specify the design constraints annotated over a UML models. In (THEELEN; PUTTEN; VOETEN, 2004), OCL is used together with an UML profile called UML-SHE to annotate requirement formulas and specify the system model. By using the UML-SHE profile the resulting models can be translated into the POOSL language, which provides the formalism required for system execution, requirement validation and design space pruning. Following the DESERT method (NEEMA et al., 2003), designers express the compatibilities, inter-aspect and resource constraints for refining a design element into one from the set of possibilities specified in the design space model. The constraints are expressed in OCL formulas specified not on the functional model, but instead on the design space model. These approaches are more flexible, because they offer a rich language for constraints definition, although the usage of OCL still limits these approaches to special types of constraints, such as limiting the number of instances or composition of elements.

In other approaches, constraints are defined by the specification of edges (BLICKLE; TEICH; THIELE, 1998; SCHLICHTER et al., 2006). The edges define the possible allocation and mapping, so that tasks in an application graph can only be mapped into processors if there are specification edges connecting them. Therefore, additional alternatives are defined by including more specification edges.

A more general way to specify NFRs and constraints is by using model-to-model transformations rules, such that the rules can be used to operate a model, in order to guide and prune the design space (OLIVEIRA et al., 2009; JACKSON et al., 2009; SCHATZ; HOLZL; LUNDKVIST, 2010). Other approaches define proprietary languages for specification of the design space and constraints (SAXENA; KARSAI, 2010; SILVANO et al., 2010).

3.7 Evaluating Design Alternatives

DSE must be performed at a high-level of abstraction in order to be efficient. If assessing each candidate solution requires its detailed synthesis and cycle-accurate simulation, design time will be prohibitive. Automatic exploration tools, based for instance on genetic algorithms or simulated annealing (see Section 3.8), must rely on fast evaluations in order to explore dozens of solutions in a few seconds or minutes, finding sub-optimal solutions that can be later refined, either when post-synthesis, cycle-accurate simulations are possible or by performance simulation tools.

The evaluation approaches differ widely concerning the DSE objectives. The most used approach is based on simulation, such as the Trace-based simulation used by ARTEMIS (PIMENTEL; ERBAS; POLSTRA, 2006) and SPADE (LIEVERSE et al., 2001). Instructionlevel simulation is used by (ASCIA; CATANIA; PALESI, 2004) for platform tuning. By using cycle-accurate simulation, PLATUNE (GIVARGIS; VAHID, 2002) evaluates solutions also for platform tuning. The approach proposed by the MILAN framework and its tools (BAKSHI; PRASANNA; LEDECZI, 2001; LEDECZI et al., 2003; MOHANTY; PRASANNA, 2002) is simulation at several abstraction levels, starting with data sheets and rough estimates or pre-characterized components' information, after that trace-driven simulation and then specific simulation using cycle-accurate simulation. The integration of MILAN and HiperE (MOHANTY; PRASANNA, 2002) allows hierarchical evaluation. The HiperE approach is divided into two steps. In the first step, HiperE uses one of the evaluation mechanisms provided by MILAN to simulate and extract the components' individual properties, then in the second step it performs system-level estimation by using the information extracted from the previous step. The first step exploits the hierarchical representation of the application, where each component can have a behavioral script. These scripts can be highly abstract or completely detailed.

Some other approaches are employed, such as symbolic programs used in Archer (ZIVKOVIC et al., 2003) or different types of high-level analytical estimates as composition functions used in ARTEMIS (NEEMA et al., 2003) and curve fitting used in (REYNERI et al., 2001). More simple approaches are also used to quickly extract system properties and perform DSE such as fixed cost (AXELSSON, 1997; BLICKLE; TEICH; THIELE, 1998; DICK; JHA, 1998; ERBAS; ERBAS; PIMENTEL, 2003) or analysis based on previously characterized library, as in (MATTOS et al., 2004; DICK; JHA, 1998; REYNERI et al., 2001). Other methods combine different approaches in order to improve the estimation accuracy and reduce the estimation time and/or use a different approach for each step of DSE, such as in (MOHANTY; PRASANNA, 2002; ERBAS; ERBAS; PIMENTEL, 2003; OLIVEIRA et al., 2006).

SESAME (PIMENTEL; ERBAS; POLSTRA, 2006) implements the trace-based simulation approach to compose the ARTEMIS (PIMENTEL, 2008) environment. In the trace-based simulation, each process of the Kahn Process Network, extracted from the application model, is executed for a virtual processor. The process execution generates the trace, a sequence of application events for each process, which is used during simulation. The virtual processor is a mapping layer, which reads the application trace and dispatches the events to the architectural platform model. During the design exploration process, an exploration tool or designer can change the mapping to evaluate another candidate solution without changes in the application trace.

The Platune environment (GIVARGIS; VAHID, 2002) is a platform-tuning framework used to select appropriate architectural parameter values, for a given application mapped onto the parameterized System-on-Chip (SoC) platform, in order to meet performance and power objectives. Platune is composed of tightly integrated cycle-accurate simulation models for SoC components (e.g. processors, memories and buses). Then power models based on gate switching activities for each component must be parameterized according to the parameterization of the respective component. The behavioral simulator collects the consumed cycles and detailed statistics on the internal activity. The processor power model uses an instruction-based approach. The power consumption is calculated considering all executed instructions and register file accesses, and a preview gate-level simulation is used to calibrate the instruction-level information. These simulation models capture dynamic information essential for computing power and performance metrics.

An analytical approach based on a non-linear method is presented in (BONTEMPI; KRUIJTZER, 2002) to estimate execution time. The proposed method consists of two phases: i) using the lazy learning algorithm in order to build a model for curve-fitting (training phase); ii) use the produced model to estimate properties from a system (test phase). During the training phase a profiler extracts a functional signature vector for a virtual processor from a benchmark set. The function signature vector contains the instruction types that appear in the code and the number of times each instruction type is executed. This functional signature is theoretically independent of the target architecture, so it can be reused for estimation with different processors. An architectural signature of the target processor is also extracted, which uses the number of memory wait cycles and the ratio between the Central Processing Unit (CPU) clock and bus clock as parameters. The functional and architectural signatures and the number of clock cycles needed to execute each application in the benchmark set are the inputs for the model-training phase. During the test phase the profile for the application is used as input for the trained model.

HiPerE (MOHANTY; PRASANNA, 2002) is a high-level performance estimator proposed to guide performance evaluation and mapping in SoC architectures. The input for the HiPerE simulator is an architecture and application described in Generic Model (GenM). A GenM description models the SoC architecture capabilities that will be used to optimize the application mapping. The SoC architecture consists of three components: a processor, reconfigurable logic, and memory. GenM describes the different architecture configurations, such as voltage operations of the processor, power states for the memory, and reconfiguration cost for the reconfigurable logic. In GenM, an application is described as a task graph. For each task, a set of performance parameters is given by the designer, for instance, the amount of input and output data to/from memory and the time and energy for executing the task at a given voltage. The initial estimations can be obtained by analytic methods. The authors show an example describing performance and energy as a function of the operational frequency. To improve the accuracy of these initial estimations, the authors propose linking GenM with a simulation-based framework in order to estimate the performance of an individual task with more accuracy. This framework, called MILAN, takes the task description (in C) and generates the scripts as well as the configuration files necessary to launch the simulator and to obtain the performance and power estimation. Using a symbolic simulator, HiPerE can verify the performance (latency in completing the task graph execution) and the energy for a given mapping. This fast symbolic simulation enables system optimization in terms of power consumption or performance.

A combined approach is described in (REYNERI et al., 2001). This approach provides an IP library, and for each hardware components there is a model based on a second order curve-fitting process from real implementation data in order to estimate chip area, byte size, clock cycles, energy per operation and other system properties. For software components estimation the method proposed in (LI; MALIK, 1995) and (TIWARI; MALIK; WOLFE, 1994) is applied, which provides the worst case execution time and power estimation by formulating an Integer Linear Programming (ILP) from a CDFG. This CDFG is extracted from basic blocks after the compilation for the target platform. The simulation on the Matlab Simulink Environment provides the overall estimation for the system, computed based on individual figures estimated from each system component by using their associated models.

3.8 Automatic Exploration Mechanisms

The design space is usually too large, the decision problem is often NP-hard and evaluating all design alternatives would require too much time. An efficient way of pruning the design space and evaluate only interesting solutions is needed. This requires adequate search mechanisms. Usually an engineer can not freely choose from all alternative design decisions, so that one restricts the search according to previous design decisions or/and constraints - e.g. number and type of processors or/and cost and performance.

The search for solution of two important synthesis tasks, namely allocation (selection of components) and mapping between selected components from an architectural template are NP-hard (BLICKLE; TEICH; THIELE, 1998). Besides the allocation and mapping, the determination of configuration parameters (knobs) of an architecture is also NP-hard (MOHANTY; PRASANNA, 2002).

Because the design space presents a large number of alternative designs, usually the design decisions are taken in two different steps in order to manage the complexity. The first step focuses on system level design, which has a coarse grain DSE dedicated to global decisions such as hardware allocation, hardware and software partitioning, task mapping, and scheduling. Most DSE proposals concentrate on this kind of DSE, such as (BAKSHI; PRASANNA; LEDECZI, 2001; BALARIN et al., 2003; BLICKLE; TEICH; THIELE, 1998; BONDé; DUMOULIN; DEKEYSER, 2005; DICK; JHA, 1998; DWIVEDI; KUMAR; BALAKRISHNAN, 2004; ERBAS; ERBAS; PIMENTEL, 2003; KANGAS et al., 2006; LEDECZI et al., 2003; MIHAL et al., 2002; PIMENTEL; ERBAS; POLSTRA, 2006). These methods usually perform DSE at a high abstraction level, where the design space has more alternatives and the optimization provides better results (MATTOS et al., 2004; THEELEN; PUTTEN; VOETEN, 2004), and rely on heuristic methods to search in the design space. In the other step, platform tuning is performed, by which the established system architecture is fine-tuned according to system requirements. Platform tuning is a special type of DSE, by which a system is usually evaluated using lower abstraction level models in order to establish values for local configuration parameters, for instance cache configuration, bus width and buffer size. The methods proposed in (GIVARGIS; VAHID, 2002; ASCIA; CATANIA; PALESI, 2004) are examples of platform tuning environments. Few tools cover both high-level DSE and platform tuning, such as the work presented in (MOHANTY; PRASANNA, 2002).

We can distinguish two classes of search methods, the exact and the heuristic methods (ROTHLAUF, 2011). The former methods guarantee that the optimal solution can be found. The latter efficiently sample some point in the design space, however, without guaranteeing that the optimal solution can be found.

Exact methods enumerate all possible design points in the space. Some tools uses dynamic programming (MOHANTY; PRASANNA, 2003; DWIVEDI; KUMAR; BALAKRISHNAN, 2004), others use exhaustive enumeration with special methods to prune the design space, by specifying parameter dependencies (GIVARGIS; VAHID, 2002; GIVARGIS; VAHID; HENKEL, 2002). Symbolic techniques are also used, such as OBDD (NEEMA et al., 2003; MOHANTY; PRASANNA, 2002; PIMENTEL; ERBAS; POLSTRA, 2006) and Satisfiability Problem (SAT) solvers (ANDERSEN et al., 2012). As their effort to solve NP-problems increases exponentially with the problem size, they are not applied to explore large design spaces. However, these methods find their utilization in specific types of problems.

The heuristic methods exploit properties of the problem to sample some points in the design space, following some search strategy. In order to provide adequate coverage strategies they may incorporate design knowledge and combine multiple algorithms, such as global and local search. Some examples of this approach uses Genetic Algorithms (AXELSSON, 1997; ASCIA; CATANIA; PALESI, 2004; BLICKLE; TEICH; THIELE, 1998; DICK; JHA, 1998; ERBAS; ERBAS; PIMENTEL, 2003), Simulated Annealing (AXELSSON, 1997; KANGAS et al., 2006; OLIVEIRA et al., 2008) and Tabu Search (AXELSSON, 1997). One work has also investigated the application of OBDDs, Multi-valued Decision Diagrams (MDD) and SAT solvers, in order to improve an heuristic algorithm called Strength Pareto Evolutionary Algorithm (SPEA2)(SCHLICHTER et al., 2006).

3.9 Discussion

After many years of research in this field, there is still a lack of information to evaluate and compare these methods (GRIES, 2004). The main problem for the evaluation is the lack of standard benchmarks and metrics to allow extracting concrete measurements related to DSE. Moreover, the different languages, models and abstraction levels make complex the experimental setup required to produce metrics for comparison. Such a situation makes it difficult to point out different contributions and evolution of the methodologies.

The pressure to reduce the time-to-market and the ever growing design complexity enforce the adoption of languages and MoCs at higher abstraction levels. Languages such as Simulink and UML are extended, such as by Simulink CAAM, MARTE and SysML, in order to comprise the new requirements for embedded system design, which must deal with multiple systems aspects. Simultaneously, exploitation of executable models and virtual prototypes bridges the gap between abstraction levels, such as by using models described in xtUML or SystemC.

The richest diversity among DSE methods are methods adopted for evaluation of design candidates. Each approach focuses on specific abstraction levels and system information to perform the exploration activity. Thus, a large set of tools is necessary to comprise the entire system design. Furthermore, the design space search mechanisms rely on system properties provided by the current evaluation approaches using simulation at several levels or imprecise information provided by the designer by using *ad hoc* methods. This results in time consuming automatic DSE or waste of effort by manual integration of tools.

Most of all automatic design space exploration methods rely on a heuristic search. That leads to no guarantee to find the best solution. However, due to the current complexity of the design space, the time to apply exact methods is still prohibitive, until one finds more efficient methods to prune and guide the process. Earlier methods for DSE do not provide methods to apply constraints defined by an engineer, and their limited constraints are embodied inside the tools and in the search method. Newer methods use more abstract modeling languages, such as UML, and allow limited specification of constraints, e.g. by using OCL. However, few works allow some integration between search methods and constraint specification. The works that allow such integration apply MDE techniques and are discussed in Chapter 4.

An item missing in the discussion of DSE is the guarantee that the solution found is correct and satisfies all requirements. Although some methods rely on few requirements to prune the design space, none work presents integration of DSE to the full requirements of the system and the verification process. Only the work in (SCHLICHTER et al., 2006) presents an investigation on the improvement of the solution generated by heuristics, in order to improve feasibility and satisfaction of requirements.

Finally one can highlight that most of the DSE methodologies focus on System-Level design decisions, so that the usually supported DSE activities are architectural allocation (considering memories, processor and buses), task mapping to the architectural template and task scheduling. However, other design decisions must be made during the development process. Some works focus on other activities, such as configuration of platform parameters, composition of components selected from a library, etc. However, few works present a proposal to improve the flexible support for exploration of different design activities. Moreover, the generation of design candidates is coded inside the tools and cannot be changed, because most of the DSE tools are black box tailored for a specific domain. Only few tools present limited mechanisms for user-defined constraints or support the inclusion of the user expertise to guide the DSE. The metrics required to evaluate a design candidate may vary according to design activities to be performed. In this way, different evaluation tools may be required too. However, many approaches for DSE are tightly coupled to an evaluation tool, and make assumptions on the domain of the application and how the design is specified. They are bound to design languages, MoCs and design activities. The focus of DSE methodologies is the automation of design activities and computational support for design decisions. However, few methodologies provide means to integrate the DSE methodology into a comprehensive development process. In this way they either leave a gap between the design artifacts and the DSE inputs or are restricted to specific artifacts and languages.

The first works on DSE were published by the end of the 80's. DSE has been further investigated after the emergence of hardware and software co-design in the 90's (WOLF, 2003). After more than 20 years, DSE methods still represent an important research topic, as observed in reviews on DSE (GRIES, 2004; VEGA-RODRIGUEZ, 2013), and co-design (DENSMORE; PASSERONE; SANGIOVANNI-VINCENTELLI, 2006). However, there are still many challenges to be overcome and tools start finding their way in the industry, such as modeFrontier[4], System Architecting[5], and Platform Architect[6].

[4]http://www.esteco.com/modefrontier

[5]http://www.cofluentdesign.com/index.php/en_US/Products_Services/cofluent-studio/system-architecting.html

[6]http://www.synopsys.com/Systems/ArchitectureDesign/pages/PlatformArchitect.aspx

4 MODEL DRIVEN ENGINEERING OF EMBEDDED SYSTEMS

This chapter introduces the basic concepts of MDE and some tools to provide the technological framework to support an MDE methodology. It also presents the state-of-the-art on available MDE methodologies for Embedded Systems in general and on DSE in particular. Finishing this section a discussion on the state-of-the art is presented.

4.1 Motivation

In order to overcome the difficulty in rising the abstraction level and to improve the automation of the design from the initial specification until the final system, research efforts look for modeling methods, formalisms, and suitable abstractions to specify, analyze, verify, and synthesize embedded systems in a fast and precise way.

The main motivation for using models in the design of embedded systems is abstraction. Abstraction helps us to understand a complex system, hiding irrelevant information to solve a specific problem. However, abstraction alone does not improve the development. Accuracy is required, so that models truly represent a specific system view. A model must clearly communicate its intent and must be easy to understand and to develop, in order to be effective (SELIC, 2003).

A prominent effort that attempted to use models in order to raise the abstraction and automate development tasks resulted in the Computer Aided Software Engineering (CASE) tools. CASE tools provide graphical representations for fundamental programming concepts and automatically generate implementation code from them. The main purpose of these tools was to reduce the effort of manually coding, debugging and porting programs. However, due to the limited platforms existing at that time, the code to be generated was too complex for the available technology. Moreover, the graphical representations were too generic and poorly customizable, and thus they could not support many application domains. Nowadays, these limitations have been drastically reduced, due to object-oriented languages and development frameworks, which make the reuse of software components easier. However, these development frameworks and platforms are extremely complex and evolve quickly, causing a fragmented view due to multiple tool integrations required for developing new applications (SCHMIDT, 2006).

Although models are used in any engineering domain, only recently they start playing a central role in the development embedded systems (SELIC, 2003). MDE (KENT, 2002) has been proposed to improve the complexity management and also the reusability of previously developed artifacts. MDE raises the design abstraction level and provides mechanisms to improve the portability, interoperability, and maintainability of models.

4.2 Model-Driven Engineering

The MDE approach was proposed to overcome the limitation of the object technology to raise the abstraction and to deal with the increasingly more complex and rapidly evolving systems we are developing today. It is defined as:

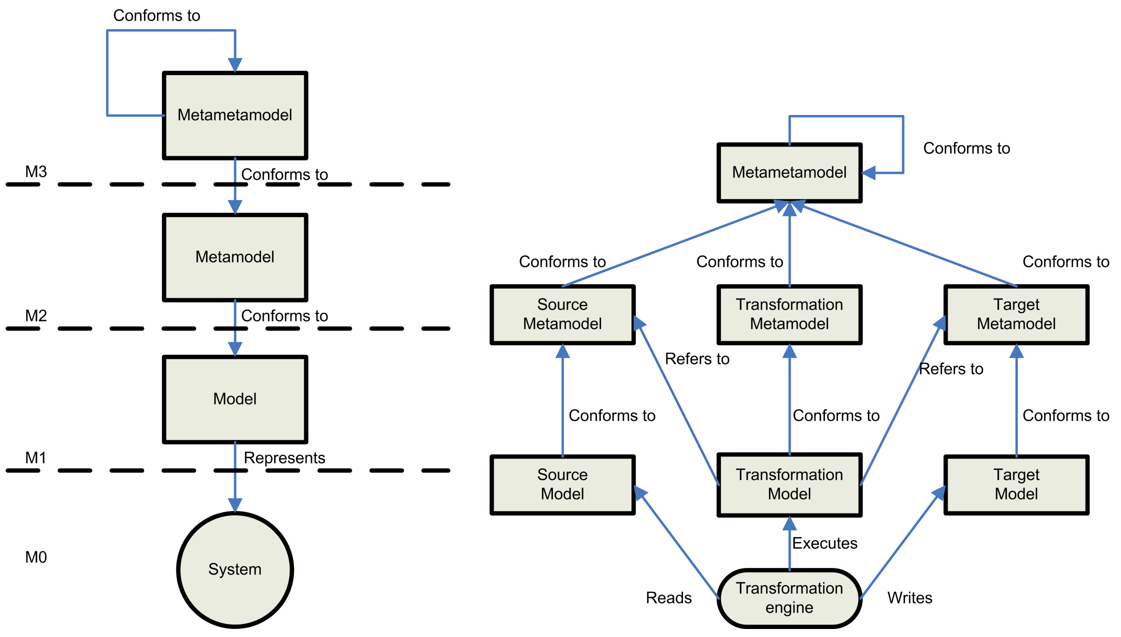
> *"A set of well defined practices based on tools that use at the same time metamodeling and model transformation to achieve some automation goal in the production, maintenance or operation of software-intensive systems." (BEZIVIN, 2005)*

Proposing that *"Everything is a model"*, MDE promotes the paradigm shift required to the necessary evolution (BEZIVIN, 2005). Although the central concept of this proposal - model - still has multiple definitions, a consensual definition of model and modeling is:

> *"Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something simpler, safer or cheaper than reality instead of reality by some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simpler manner, avoiding the complexity, danger and irreversibility of reality." (ROTHENBERG, 1989)*

Since the main principle of MDE is that *"Everything is a model"*, models play a central role in the development process, thus defining the scope of MDE proposed in (KENT, 2002). The basic concepts to support the MDE principle are system, model, metamodel, and the relations between them, so that a model represents a system and conforms to a metamodel (BEZIVIN, 2005). Such concepts were organized in 3+1 layers (BEZIVIN, 2005) and are illustrated in Figure 4.2(a).

Figure 4.1: MDE Basic concepts: (a) Layered organization; (b) Model transformation.



Formally, a model in MDE is a graph composed of elements (vertices and edges), where each element corresponds to a concept in a reference graph (metamodel) as defined below:

Definition 4.1 (Directed Graph)**:**
$G = \langle V_G, E_G, \delta_s, \delta_t \rangle$ is a directed graph defined by:

V_G is the set of vertices $V_G = v_0, v_1, \ldots, v_n \in G$, also denoted by $G\,(V)$.

E_G is the set of edges $E_G = e_0, e_1, \ldots, e_n \in G$, also denoted by $G\,(E)$.

δ_s is the function $\delta_s : E_G \rightarrow V_G$, which returns the source vertices of an edge.

δ_t is the function $\delta_t : E_G \rightarrow V_G$, which returns the target vertices of an edge.

Definition 4.2 (Model)**:**
$Mo = \langle G, \omega, \mu \rangle$ is a model defined by:

G is a directed graph as stated in Definition 4.1.

ω is itself a model, named reference model of Mo, associated to a graph $G_\omega = \langle V_\omega, E_\omega, \delta_s, \delta_t \rangle$.

μ is a function $\mu : N_G \cup E_G \rightarrow N_\omega$, which associates elements (vertices and edges) of G to nodes of G_ω (metamodel).

A metamodel is a model, which is a reference model for other models, so that it defines classes of models that can be produced conforming to it. It is an abstraction, which collects concepts of a certain domain and the relations between these concepts.

Models are operated through transformations, aiming at the automation of some development activity. Such transformations define clear relationships between models (BEZIVIN, 2005) and usually are specified in a specialized language to operate on models. Following the description in (GASEVIC; DJURIC; DEVEDZIC, 2009), a model transformation means converting one or more source models to a target model, where all models conform to some metamodel, including the model transformation itself, which is also a model. Figure 4.2(b) illustrates the concept of model transformation in the MDE context.

Model transformation plays a key role in MDE and has many applications, such as: generating low-level models from high-level ones, generating development artifacts and source code, mapping and synchronizing models, creating query-based views of a system, model refactoring, reverse engineering, model verification, and others (CZARNECKI; HELSEN, 2006). Based on the possible applications of model transformations, they can be classified in:

- Model-to-Model, when the source and target of the transformation are models, e.g. transformation from UML to a Relational Data Base (RDB) schema or from a Platform Independent Model (PIM) to a Platform Specific Model (PSM);

- Model-to-System, characterizing a generation from model to system, which can include program code or any other artifact, e.g. UML to Java or Simulink to C++;

- System-to-Model, meaning a reverse engineering, such as from Java code to a UML model or from Java code to a business model.

4.3 Technological Frameworks

Technological frameworks (FRANCE; RUMPE, 2007) are tools to support different operations and common tasks for MDE independently from the application domain. Such tools rely on standards, such as Model-Driven Architecture (MDA) (OMG, 2003), Model Integrated Computing (MIC) (SZTIPANOVITS; KARSAI, 1997), and Software Factories, in order to generalize the manipulation of models, providing facilities such as persistence, repository management, copy, etc. They are the technological support of MDE principles. An overview on some standards and tools are presented in the next subsections.

4.3.1 MDE Standards

4.3.1.1 Model-Driven Architecture

MDA (OMG, 2003) is a standard proposed by OMG for software development. The main purpose of MDA is the abstraction of platforms, so that the business models can be reused as the technological platform evolves. MDA integrates different OMG standards, such as Meta-object Facility Specification (MOF) for metamodeling, UML for system modeling, Software Process Engineering Metamodel (SPEM) for process modeling, and Query/View/Transformation (QVT) for model transformation. In order to separate business and application models from the underlying platform, MDA advocates three modeling dimensions (view points):

- The Computation Independent Model (CIM) focuses on the required features of the system and on the environment where it must operate;
- Platform Independent Model (PIM) focuses on business functionality and behavior, which are unlikely to change from one platform to another;
- Platform Specific Model (PSM) describes platform specific details integrated with elements of PIM.

The relationship between PIM and PSM in MDA can be established by automatic or semi-automatic mechanisms, specifying a mapping between these models. MDA suggests that this mapping can be specified by using QVT, so that a transformation engine can generate the automatic transformation from PIM to PSM. The languages used to express these models are defined by means of metamodels using MOF, which are able to represent abstract and concrete syntax, as well as the operational semantics of the modeling language. Originally, MDA was proposed for enterprise architectures that use platforms, such as Java2EE, CORBA, VisiBroker, and WebSphere. However, as using the MDA approach the development of systems can be focused on aspects that do not involve implementation details, many other domains start considering the MDA approach, such as real-time and embedded systems. Therefore, MDA and the experience with OMG standards are in the origins of MDE.

4.3.1.2 Model Integrated Computing

MIC (SZTIPANOVITS; KARSAI, 1997) is an initiative from Vanderbilt University. In this approach, models representing different views capture the designer's understanding of the computer-based system, including information process, physical architecture, and operating environment. A formal specification of the dependencies and constraints among these models allows for the generation of tools to solve an entire class of problems. MIC proposes a two-steps development process. In the first step, a domain-independent abstraction is used to formally define a domain specific environment and the required models, languages and tools. In the second step, three typical components delivered from the previous phase are used for system engineering:

- A graphical model builder is used to specify domain specific models. Constraints explicitly defined at meta-level allow model testing;
- A model database stores domain specific multi-view models using a multi-graph architecture;
- Model Interpreters are used to synthesize executable programs from the domain specific models and generate data structures for the tools.

MIC has a strong influence on the principles of MDE as it has a wider basis on engineering of systems than MDA. Moreover, the two-step process advocated by MIC is close to the idea of Technological Frameworks as a basis of development for Domain Specific Engineering Tools present in the MDE approach.

4.3.1.3 Microsoft Software Factories

The main idea behind the Software Factories (GREENFIELD; SHORT, 2003) is to introduce patterns of industrialization in the software development. It is *"a software product line that provides a production facility for the product family by configuring extensible tools using a software template based on a software schema*" [1]

A Software Factory Schema describes the artifacts that comprise a software product. It is represented by a graph, where vertices are viewpoints and edges are relationships between viewpoints (mapping). Each viewpoint defines the tools and materials required by a concern in a specific abstraction level. Attached to a viewpoint, a microprocess is defined for producing the artifacts described in the viewpoint. Such a process is constrained by preconditions, post-conditions and invariants that must hold when the view is stabilized.

A Software Factory Template is the collection of Domain Specific Languages (DSL), patterns, frameworks and tools described in the Software Factory Schema, which is made available to developers, in order to create a specific software product.

4.3.2 MDE Tools

The MDE approach has a practical relevance only if it can produce and transform models bringing considerably more benefit than the current practices. Therefore, to enhance the value of models, they must become tangible artifacts, which can be simulated, verified, transformed, and so on, and the burden for maintaining these models in synchronization with the produced system must be reduced (KENT, 2002).

[1]http://msdn.microsoft.com/en-us/library/ms954811.aspx

Supporting tools are essential to provide all benefits of MDE. This section describes some MDE tools, focusing on tools supported by the Eclipse Modeling Project [2]. Eclipse Modeling Project provides a unified set of modeling frameworks, tooling, and standards implementations.

4.3.2.1 Metamodeling/Abstract Syntax

As the model is the most important artifact in MDE, defining the class of models an MDE process must work on is one of the first steps. This is done by metamodeling, which defines the structured data types used to represent a system (abstract syntax). In the Eclipse Modeling Project, metamodels are defined conforming to ECORE, a metametamodel (layer 3 in Figure 4.1) defined by the Eclipse Modeling Framework (EMF). EMF is a projection of ECORE and of the models conforming to it, into Java API. It provides code generation facilities and tools to build model editors and to compare, query, persist and validate models. As most tools in Eclipse Modeling Project are based on ECORE and EMF, and many other projects make use of EMF, ECORE is a *de facto* standard.

Besides Ecore metametamodels and EMF, other metamodeling tools are found. Kermeta [3] is based on the OMG standard Essential MOF (EMOF), which was originated from ECORE and kernel Meta Meta Model (KM3), a metametamodel proposed in (JOUAULT; BéZIVIN, 2006). MetaGME is a metamodeling tool, which implements the metamodeling concepts for MIC. Originally, its metametamodel was called Multigraph Architecture. Newest versions use UML class diagrams notation and OCL for metamodeling.

4.3.2.2 Concrete Syntax

A concrete syntax for a Domain Specific Modeling Language (DSML) can be defined using the tools from the Eclipse Graphical Modeling Project. It provides tools, such as GMF Notation and Graphiti, to specify the concrete syntax and to generate an editor to express models graphically.

The definition of the concrete syntax of languages expressed as text is also possible by using tools such as Xtext. It provides a simple EBNF language, which is used to define grammars, a generator to create a parser, an AST-metamodel (implemented in EMF), and a Eclipse text editor for the defined language.

4.3.2.3 Model Development

For common general purpose and domain specific languages, there is no need to build new editors as good tools can be found, such as Magic Draw, Enterprise Architecture and Rhapsody for modeling with UML. Simulink and Scade are DSML's commonly used for control engineering and signal processing and specialized tools for that are also provided. Eclipse Model Development tools provide model editors for some standards such as UML, Extensible Markup Language (XML), and OCL.

4.3.2.4 Model Transformation

Since model transformation is the key operation for MDE, many transformation engines and languages were proposed. However, after the experience with first languages, a discussion on classification (CZARNECKI; HELSEN, 2006) and quality metrics (AMSTEL; LANGE; BRAND, 2008) is starting to take place in the research agenda, so that a

[2]http://www.eclipse.org/modeling/

[3]http://www.kermeta.org/

standard with high adoption may rise.

Eclipse Modeling Project had many model-to-model transformation languages, but now the efforts concentrate on ATL and in a reference implementation of QVT, the QVT Operational. Other languages are provided as Eclipse projects or Eclipse plug-ins, such as VIATRA II and GReAT.

Model-to-text (Model-to-System) transformation is provided by EMF through three different template-based languages: Java Emitter Template (JET); Acceleo, which is an implementation of an OMG standard, named MOF to Text Language; and Xpand, which was initially an openArchitecturalware component.

4.4 MDE Methodologies for Embedded Systems

In (FRANCE; RUMPE, 2007) two classes of MDE tools were identified. One was called MDE Technology Framework, which supports the MDE process by providing tools for different operations and common tasks, independently from development domain, such as metamodeling, transformation engines and languages, debugger, tracing and other facilities. These tools rely strongly on standards. Some of them were presented in the previous section, such as the tools provided by the Eclipse Modeling Project. The other class of tools adopts an MDE framework to provide Domain Specific Application Development Environments (DSAEs), which aggregate domain specific knowledge to define relations between models and how these models could be refined. In this thesis the term Domain Specific Model-Driven Engineering Tools (DSMDET) is used, in order to highlight the fact that some tools rely on MDE technological framework to engineer not only software, but systems, which may be also composed hardware, electrical, mechanics parts. This section presents some DSMDETs for embedded system development.

The adoption of platform-independent design and executable UML has been vastly investigated. For example, xtUML (MELLOR; BALCER, 2002) defines an executable and translatable UML subset for embedded real-time systems, allowing the simulation of UML models and C code generation oriented to different microcontroller platforms. The Model Execution Platform (MEP) (SCHATTKOWSKY; MUELLER; RETTBERG, 2005) is another approach based on MDA, oriented to code generation and model execution, as well as the Framework for UML Model Behavior Simulation (FUMBeS) (WEHRMEISTER; PACKER; CERON, 2012).

Other approaches improve the integration of the design tools into an MDE environment, by defining meta-models, and the transformations on them include some refinement. This approach includes the DaRT (Data Parallelism to Real Time) project (BOULET et al., 2003; BONDé; DUMOULIN; DEKEYSER, 2005), whose evolution produced the Gaspard2 framework. It proposes an MDA-based approach that has many similarities with our approach in terms of meta-modeling concepts. DaRT defines MOF-based metamodels to specify application, architecture, and software/hardware associations and uses transformations between models to refine an association model. In the Gaspard2 framework (PIEL et al., 2008) UML/MARTE models are used as input and transformation to other tools, providing support for co-synthesis, simulation and formal verification, by translating a model into synchronous reactive languages. However, no automated DSE (Design Space Exploration) strategy based on these transformations is implemented, and the main focus is code generation for simulation at Transaction Level Model (TLM) and RTL levels. In this approach, each candidate solution is simulated at a different abstraction level, thus guiding the designer in the DSE activities.

The Aspect-oriented Model-Driven Engineering for Real-Time systems (AMoERT) methodology (WEHRMEISTER et al., 2008) proposes an automated integration of design phases for distributed embedded real-time systems, focusing on automation systems. The proposed approach uses MDE techniques together with Aspect-Oriented Design and previously developed (or third party) hardware and software platforms to design the components of distributed embedded real-time systems. The Aspect-Oriented Design concepts allow a separate handling of functional and NFRs, improving the modularization of the produced artifacts. In addition, the methodology is supported by the GenERTiCA code generation tool (WEHRMEISTER et al., 2008), which uses mapping rules for the automatic transformation of UML models into source code for software and hardware components, which can be compiled or synthesized by other tools, thus obtaining the realization/implementation of the distributed embedded real-time system. During the generation process, the tool includes the required implementation code to handle the specified aspects for NFRs (model weaving).

Metropolis (BALARIN et al., 2003) is an infrastructure for electronic system design, in which tools are integrated through an API and a common metamodel. Following the platform-based approach (FERRARI; SANGIOVANNI-VINCENTELLI, 1999), the Metropolis infrastructure captures application, architecture and mapping using a proposed UML-platform profile (UML AND PLATFORM-BASED DESIGN, 2003). Furthermore, its infrastructure is general enough to support different Models of Computation and to accommodate new ones. No automatic support for Design Space Exploration is provided by Metropolis, which proposes an infrastructure to integrate different tools. Nevertheless, the current simulation and verification tools integrated into Metropolis and the proposed refinement process can be used to manually perform some architectural explorations (task mapping, scheduling, hardware/software partitioning) and component configuration. Moreover, the refinement process allows the explicit exploration of application algorithms, which implement a higher level specification.

Koski (KANGAS et al., 2006) is a UML-based framework to support MPSoC design. It is a library-based method, which implements a platform-based design. Koski provides tools for UML system specification, estimation, verification, and system implementation on FPGA. The Koski design flow starts with a requirement analysis, which specifies the application or architecture requirements and design constraints. Following the design flow, the application, architecture and the initial mapping are specified as UML 2.0 models. A UML interface handles these models and generates an internal representation, which is used for architectural exploration. The architectural exploration is performed in two steps; the first one is static, fast and less accurate; the second one is dynamic. At the end of the design flow, the UML models are used to generate code and the selected components from the platform are linked to build the system.

Other complete environment for design space exploration is the MILAN (BAKSHI; PRASANNA; LEDECZI, 2001) framework, with two exploration tools called DESERT (NEEMA et al., 2003) and HiPerE (MOHANTY; PRASANNA, 2002). The focus of MILAN is the integrated simulation of embedded systems, so that it evaluates pre-selected candidate solutions. The hierarchical simulation provided by MILAN allows a designer to explore the design space at several abstraction levels, by using the DESERT and HiPerE tools. First, the DESERT tool uses models of aggregated system sub-components and constraints to automatically compose the embedded system through OBDD, based on a complete pre-characterization of components. Moreover, the DESERT tool performs design space pruning, reducing the number of candidate solutions. After that, HiPerE can be used for accurate system-level estimation, exploring the pruned design space. Finally, by using integrated simulation at lower abstraction levels, the designer can explore the reminder of the design space, performing then also platform tuning.

The Multi-objective Design Space Exploration of Multiprocessor SOC Architectures for Embedded Multimedia Applications (MULTICUBE) was an European research project devoted to provide methods for DSE. One of the outcomes from this project was the MULTICUBE methodology and tool for automated DSE (SILVANO et al., 2010). The MULTICUBE framework includes several optimization algorithms to identify trade-offs between objecives. In order to evaluate candidate designs, MULTICUBE provides an XML interface, so that different simulators can be plugged to it. The framework also provides a library of components for response surface modeling, which reduces the time required to evaluate candidate designs, as it replaces the evaluation by simulation.

The Generic Design Space Exploration (GDSE) (SAXENA; KARSAI, 2010) is a meta-programmable framework, whose implementation is based on MIC/GME. The framework provides a metamodel that defines the Abstract Design Space Exploration Language (ADSEL) and the Constraint Specification Language (CSL). ADSEL is used as a base metamodel, which must be composed with a domain metamodel in order to create a domain-specific DSE metamodel. The DSE specific concepts are added into the composed metamodel, tailoring the composed metamodel to the domain of the problem. The goal of ADSEL and such composition is to allow a generic representation of the design space, which can be configured to different domains. CSL was proposed to provide an expressive constraint language able to capture different types of constraints, such as arithmetic, boolean and set constraints. From the domain-specific DSE model and CSL the GDSE framework generates an intermediate description, namely Intermediate Language (IRL), which is used to generate tool specific languages, such as Minizinc [4] that can be used as input for different solvers.

A strategy for DSE by means of model transformation is present in (SCHATZ; HOLZL; LUNDKVIST, 2010). The approach allows a declarative relational definition of the design space using Prolog. An Eclipse framework for model transformation called tuProlog is used to execute the design space definition and generate candidate designs. It provides a translation from ECORE metamodels to Prolog, which improves automation and the integration with other flows. Then this Prolog model is extended with the Prolog specification of the design and the Prolog specification of the design space. The resulting model is a declarative description of the DSE problem to be solved.

4.5 Discussion

The first application of model-based engineering for embedded systems was the adoption of high-level modeling languages alone, such as UML and Simulink, which lead to the fail of CASE tools (SCHMIDT, 2006).Then the application of MDE started, consisting in the development of model-to-model transformations in order to transform output models from a tool to another, so that the existing development tools could be used (MURILLO; MURA; PREVOSTINI, 2010; BALARIN et al., 1997; KANGAS et al., 2006). Afterwards, domain specific metamodels were proposed to capture the heterogeneous nature of these systems, and syntactic transformations were use to generate systems based on these metamodels (BOULET et al., 2003; BONDé; DUMOULIN; DEKEYSER, 2005; NASCIMENTO et al., 2006). The following steps were the devel-

[4]http://www.minizinc.org/

opment of smart generators, which use transformations based on the semantics of elements, such as GenERTiCA (WEHRMEISTER et al., 2008), and development activities fully implemented using MDE concepts, such as performance evaluation (ATITALLAH et al., 2006), schedulability analysis (PERALDI-FRATI; DEANTONI, 2011) and DSE (OLIVEIRA et al., 2009; SCHLICHTER et al., 2006; SAXENA; KARSAI, 2010; SCHATZ; HOLZL; LUNDKVIST, 2010).

Although MDE provides many benefits, tools applying MDE for DSE adopt a similar pattern of the ones without the use of MDE. Most of tools are too specialized for some specific design decisions, such as task mapping and scheduling (KANGAS et al., 2006), system construction and component integration (NEEMA et al., 2003), or manual DSE with focus on abstracting modeling and simulation (PIEL et al., 2008; BALARIN et al., 2003). Moreover, the generation of candidate designs is internally implemented, usually as a function that is programmed directly in the tool. As result, no extension mechanisms are provided, requiring multiple tools to support each design activity. Except for the Koski and DESERT methods, for most approaches either the constraints set is restricted to previous constraints implemented by the tool or the method supports limited constraints constructs.

An issue scarcely discussed is the reuse of design artifacts for specification of DSE scenarios and reuse of DSE artifacts. Most approaches adopt abstract modeling of the application and platform and derive a DSE scenario from that. However, no DSE scenario specification is supported, so that there is no flexibility in the DSE without changing the application and platform models. Specific DSE models are proposed as UML diagrams (GRUTTNER et al., 2012; KANGAS et al., 2006) or a proprietary DSE language based on XML as proposed by MULTICUBE (SILVANO et al., 2010), so that without changes in the application and platform, different DSE scenarios can be configured. The approaches using UML for DSE scenario specification has the benefit of reusing design elements from the system model. However, this restricts the development process to use only UML. The DSE language proposed by MULTICUBE provides more flexibility. However, there is no link between the DSE model and development artifacts, , thus requiring a manual synchronization of the models, and the approach does take benefit of reuse information from the design models. Moreover, the DSE language provides no abstraction for design elements and optimization, thus requiring an engineer to adequately code the design elements for the optimization problem.

Recently flexible and generic support for DSE was proposed (SCHATZ; HOLZL; LUNDKVIST, 2010; SAXENA; KARSAI, 2010; KANG; JACKSON; SCHULTE, 2011). In (SCHATZ; HOLZL; LUNDKVIST, 2010) Prolog is used as language for DSE, exploiting the backtracking features of model transformation engines, which adopts formal methods to explore the design space. However, design models must be described in Prolog together with the DSE model, and no design model injection or translation to Prolog is provided. Moreover, no design abstractions are provided, so that the engineer must specify and customize everything. The approach in (KAHN, 1974) is similar to the approach in (SCHATZ; HOLZL; LUNDKVIST, 2010). They provide a proprietary language, namely FORMULA with some constructs with higher abstraction, so that transformations and constraints are easier to specify. However, no integration into another flow is provided. The great disadvantage of all these methods is that they are too generic. Actually, these approaches are better classified as MDE Technology framework, because they can be applied to any (meta)model, with no specific abstraction nor constructs for DSE. In the contrary, the approach proposed in this work is a DSMDET, which combines flexibility with special constructs for DSE. The proposal for MDE adoption for DSE is described in Chapter 5.

In (SAXENA; KARSAI, 2010) a language specific tailored for DSE is defined and enhanced with a constraint language. This approach also provides translation to Minizinc, which allows access to different solvers. However, the DSE model is enclosed into the design model by composing the Generic DSE metamodel and a DSL used to representing the design model, so that an engineer is forced to use the tools generated from the composed metamodel to specify the systems. Moreover, the Generic DSE Metamodel represents just one single type of problem.

5 MODEL-DRIVEN ENGINEERING METHODOLOGY FOR DESIGN SPACE EXPLORATION

This chapter presents an MDE methodology for DSE of embedded systems. It discusses the problem addressed and presents an overview of the MDE methodology for DSE. Thereafter, each process defined in the methodology is described. The first process describes the adopted system modeling method. Then a metamodel is defined to represent the DSE domain with its relevant concepts. A weaving method is proposed to associate design elements to the DSE domain model, so that the DSE process can be modeled independently from the design language adopted in the development process. The approach to specify DSE rules is presented, so that an engineer can use it to prune and guide the DSE process. The DSE process is described and alternative DSE methods are presented to be integrated into this methodology. Then the evaluation process is described and the adopted evaluation method is briefly explained. Finally, the methodology is discussed in the last section.

5.1 Motivation

Based on all the discussed works in Chapters 3 and 4, the surveys on DSE (GRIES, 2004), and (DENSMORE; PASSERONE; SANGIOVANNI-VINCENTELLI, 2006), and the book (GAJSKI et al., 2009) it is observed that there is a lack of industrially strong and mature tools for DSE, in spite of many academic proposals. The following limitations are identified in the studied works:

- *Restricted support to multiple DSE activities*: Most of the DSE methodologies focus on System-Level design decisions, so that the usually supported DSE activities are architectural allocation (considering memories, processors and buses), task mapping to the architectural template and task scheduling. However, other design decisions must be made during the development process. Some works focus on other activities, such as configuration of platform parameters, composition of components selected from a library, etc. However, few works present a proposal to improve the flexible support for exploration of different design activities.

- *Fixed method to generate solutions*: As most of the DSE tools are black-box tailored for a specific domain, the generation of design candidates is coded inside the tools and cannot be changed. Only few tools present limited mechanisms for user-defined constraints or to include the user expertise to guide the DSE.

- *Tightly coupled to evaluation tools*: The metrics required to evaluate a design candidate may vary according to design activities to be performed. In this way, different evaluation tools may be required too. However, many approaches for DSE are tightly coupled to an evaluation tool and make assumptions regarding the domain of the application and how the design is specified. They are bound to design languages, MoCs and design activities.

- *Lack of integration with the development process*: The focus of DSE methodologies is the automation of design activities and computational support for design decisions. However, few methodologies provide means to integrate the DSE methodology into a comprehensive development process. In this way they leave a gap between the design artifacts and the DSE inputs or they are restricted to specific artifacts and languages.

From a broader point of view, DSE is performed always when an engineer must choose between multiple design alternatives, which can arise at different abstraction levels or at different steps during the development process. Moreover, a wide range of languages and models are used to specify embedded systems. By using the state-of-the-art tools, setting up a tool chain to support all or at least some of those design activities is a complex task. These tools require different inputs, which may differ from the ones used in the development process. There are also a variety of internal models and outputs formats. Such lack of reusability and flexibility in the current DSE methodologies leads to a situation that inhibit the adoption of these methodologies in a production environment.

In Section 2.2 a list of common design activities were presented. Actually the list is not exhaustive and many other activities can be added to it. In order to reduce the DSE complexity, it is important to classify the design activities that can be modeled using similar elements. Based on the state-of-the-art, in this thesis the classification proposed in (SAXENA; KARSAI, 2011) was adopted, which groups the design activities into six DSE problems, namely Configuration, Construction, Mapping, Placement, Routing, and Scheduling. Such a classification is the starting point for a consensual understanding on DSE, which drives to a common representation of the DSE problems.

The general goal of the proposed methodology is to improve the flexibility, reusability, and productivity in the DSE process. Specifically the methodology endeavors to easily integrate DSE methods into a development process. Moreover, it attempts to represent the different DSE problems in a concise and uniform way and provide a mechanism that allows a user to define the DSE rules according to the specificity of the problem to be solved. Another goal of this methodology is to automate development steps related to DSE, so that DSE artifacts can be generated from other development artifacts.

In order to fulfill the expected goals, the methodology improves the DSE flow by defining points of integration, so that the data can be gathered from design models or stored into them. In this methodology a DSE domain metamodel is proposed to represent the DSE problem and other concepts related to DSE. Moreover, model-to-model transformation rules are proposed as a mechanism to allow a user to define DSE rules to generate design candidates for the specific problem to be solved. These rules implement the semantic of constraints defined in the DSE domain metamodel, thus increasing the abstraction and improving the reuse of transformation rules.

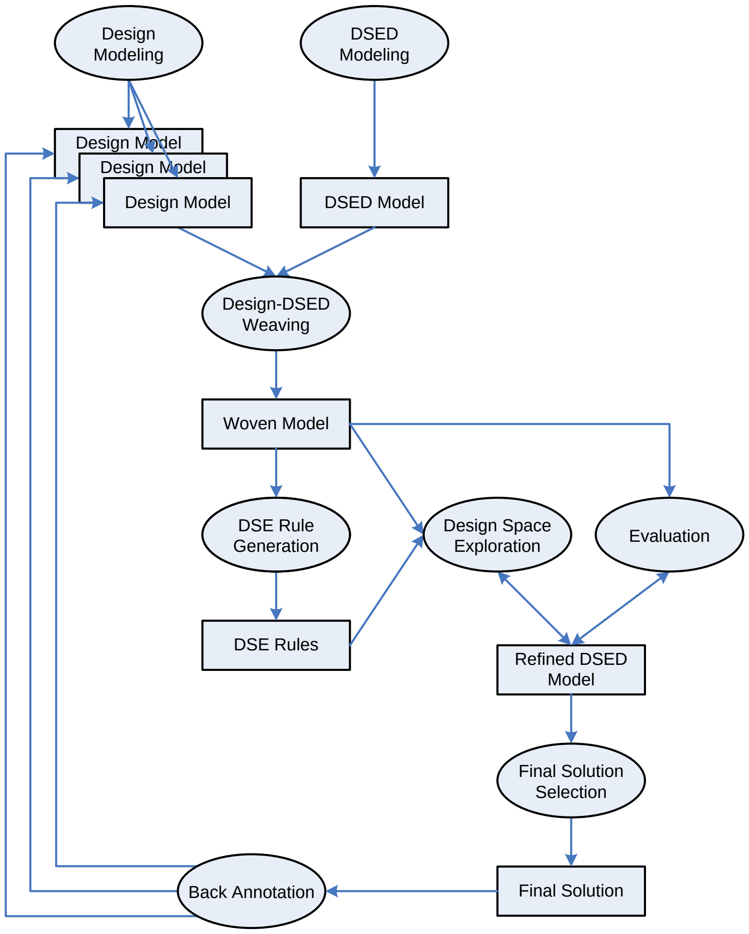
5.2 Methodology Overview

The proposed methodology applies the MDE approach, so that a metamodel is defined to represent the concepts of the DSE domain. This metamodel was named Design Space

Exploration Domain (DSED) metamodel and by using an MDE technological framework, such as the one provided by the Eclipse Modeling Project, an API is generated from this metamodel to allow manipulation of models conforming to it. In addition, a weaving technique allows the specification of DSED models independently from the design models, hence the methodology is independent of design languages. A weaving metamodel is used to represent the link between DSED and design models. It stores references to DSED elements and to elements of different design models, hence the proposed DSE methodology can combine design models specified in different languages. Moreover, model-to-model transformation rules are used to specify the rules that guide the search in the design space and prune inappropriate design alternatives, according to constraints defined in the DSED model. Such an approach allows for easy adaptation, by writing, editing, removing, or reusing transformation rules, according to the requirements of the DSE problem to be solved.

Consequently, the set of methods proposed results in a highly flexible DSE methodology, which can be integrated into the development flow at different stages, in order to handle the existing DSE problems. The proposed methodology can be applied whenever multiple design decisions must be evaluated, at earliest after the end of the analysis phase as described in Section 2.2. In Figure 5.1 the methodology flow is presented, where boxes represent artifacts and ellipses represent processes that are described briefly in the following sections.

Figure 5.1: Methodology flow for Design Space Exploration.



Design Modeling process

Before starting the DSE, a Design Modeling process must be executed. In this process engineers can use their favorite design method. Design models are inputs for the DSE methodology, which must determine the design decisions that are still open in the design models. A design modeling process is out of scope of this work, since the DSE related artifacts are specified independently from the design ones. However, technologies and methods are assumed, to provide a comprehensive context to the reader and computational tool support for the proposed methodology. More information on the design method are provided in Section 5.3.

DSED Modeling process

During the DSED Modeling process an engineer defines the DSE problems to be solved and variable domains. This process is performed manually, by using generic model editors generated by the MDE framework. Although the automation of such a process is dependent on the design language and tools used for design, by means of adequate supporting tools, is possible to extract design information, such as task graph, feature model, and communication graph. The extracted information is then used to generate the DSED model. More details on the DSED Modeling are provided in Section 5.4.

Design-DSED Weaving process

The Design-DSED Weaving process weaves design and DSED models. Such a process is important to associate design elements from different models (e.g. UML, Simulink, SystemC) to elements in the DSE model. As a result, it improves the flexibility of the DSE process, which can be applied to complex embedded systems, whose models are divided into different views and/or languages. Moreover, it allows the independence of the DSE methodology from the design method used during the development process, which is one of the problems concerning the application of DSE tools. The Design-DSED Weaving process is presented in Section 5.5.

DSE Rules Definition process

The design must meet requirements that define operations, features, constraints, standards and other rules that must be observed during the development. Likewise, design candidates generated during the DSE process must follow these rules. The rules concerning to the DSE process are named DSE rules and must be defined in a such a way that DSE tools can automatically process them during the design candidate generation.

In the proposed DSE methodology, DSE rules are model-to-model transformation rules, which may be executed by a transformation engine that receives a DSED model as input and refines this model during the DSE process. These rules are constraints to guide the search in the design space and prune inappropriate design alternatives, so that the exploration time is reduced and the feasibility of a candidate design is ensured. By using model-to-model transformation languages to define DSE rules, the flexibility of DSE is highly improved, because the DSED model can be handled according to the DSE problem to be solved, and thus the framework can be applied to different DSE problems, adapted and extended using standards tools. The specification of DSE rules is presented in Section 5.6.

Design Space Exploration process

When the DSE process is reached, all information required to execute the DSE process, such as the DSED model, woven to design models and DSE rules, are stored in a such way that a computer can process them in models that can be processed by standard APIs and transformation engines.

During the DSE process these information are handled by solvers, which generate design candidates in conformance to the DSE rules and search in the design space for adequate design alternatives. As all required information is easily accessible in two models, namely DSED and DSE rules, multiple solvers can be attached to the DSE process, such as specific heuristics, e.g. Platune (GIVARGIS; VAHID, 2002) and Koski (KANGAS et al., 2006), optimization frameworks, e.g. Opt4J (LUKASIEWYCZ et al., 2011) and Watchmaker [1], or optimization tools, e.g. modeFrontier [2] and Guimoo (LIEFOOGHE et al., 2007). The DSE process is discussed in Section 5.7.

Evaluation process

Although the search in the design space is a complex process from the computational point of view, the evaluation step is a bottleneck during DSE. As discussed in Section 3.1, there is a trade-off between time spent in the evaluation and the evaluation accuracy. The adequate abstraction level and evaluation method depend on the problem to be solved, the languages used to specify the systems and other factors.

The weaving process allows the DSE methodology to be independent from the evaluation method used, because the DSE process makes no assumptions on design artifacts specifically required for evaluation, such as modeling language and MoCs. Moreover, the DSED metamodel, together with the MDE technological framework, provides an adequate and standard API to read the required data for the evaluation method from the DSED model, as well as to write the evaluation results back into the DSED model. Therefore, the methodology is not bound to a specific evaluation method and hence allows engineers to integrate the evaluation method that is adequate to their problems into the DSE methodology.

For completeness sake, a mixed evaluation method was adopted to evaluate generated candidate designs. The evaluation method consists of a static analysis of a CDFG extracted from UML Models or C++ source code and the combination of information gathered from a platform of pre-characterized components, so that design candidates at different levels of abstraction can quickly be evaluated. More information on the evaluation process is presented in Sections 5.8 and 7.3.

Final Solution Selection process

The search for design candidates often leads to a Pareto set, whence an engineer must select one or more to proceed with the development. The trade-offs must be evaluated by an engineer. Visualization and analysis tools can be used to support this process and integrated into the methodology by means of the DSED model and the associated API or transformation. The final solution selection is out of scope of this work and is not discussed.

Back annotation

At the end of the DSE methodology, the final solution must be back annotated into the design model, so that design decisions taken during the DSE process are used to refine the design model. Such a process is partially supported by the proposed methodology, by providing the DSED metamodel, the standard API generated from it, and the Design-DSED weaving model. The DSED metamodel and API allow one to read the design

[1]http://watchmaker.uncommons.org/

[2]http://www.modefrontier.com

decisions from the DSED model, by using the API or model-to-model transformations. The Design-DSED weaving model keeps the link from the DSED model elements to elements in the design models. By using them, one can implement model transformations that automatically back annotate the design model with the design decisions.

5.3 Design Modeling

The adopted system design method uses UML 2.0 and the MARTE profile together with non-intrusive modeling guidelines to specify application, architecture, and mapping. This method tries to capture the most common approaches, so that the DSE methodology can be automated without requiring high modeling effort or a too specific modeling style. It is highly inspired in the methods presented in (FREITAS, 2007; WEHRMEISTER et al., 2008) and the MARTE2Cheddar Tool[3].In the next paragraphs examples are shown, in order to illustrate the system design modeling guidelines, considering a real-time embedded system dedicated to the automation and control of an intelligent wheelchair.

The functional requirements identified during the requirement process can be seen in the UML Use Case Diagrams, as shown in Figure 5.3(a). It defines the functional view of the system and the context the system is running. Details of such requirements are provided in textual form by using RT-FRIDA templates (FREITAS, 2007).

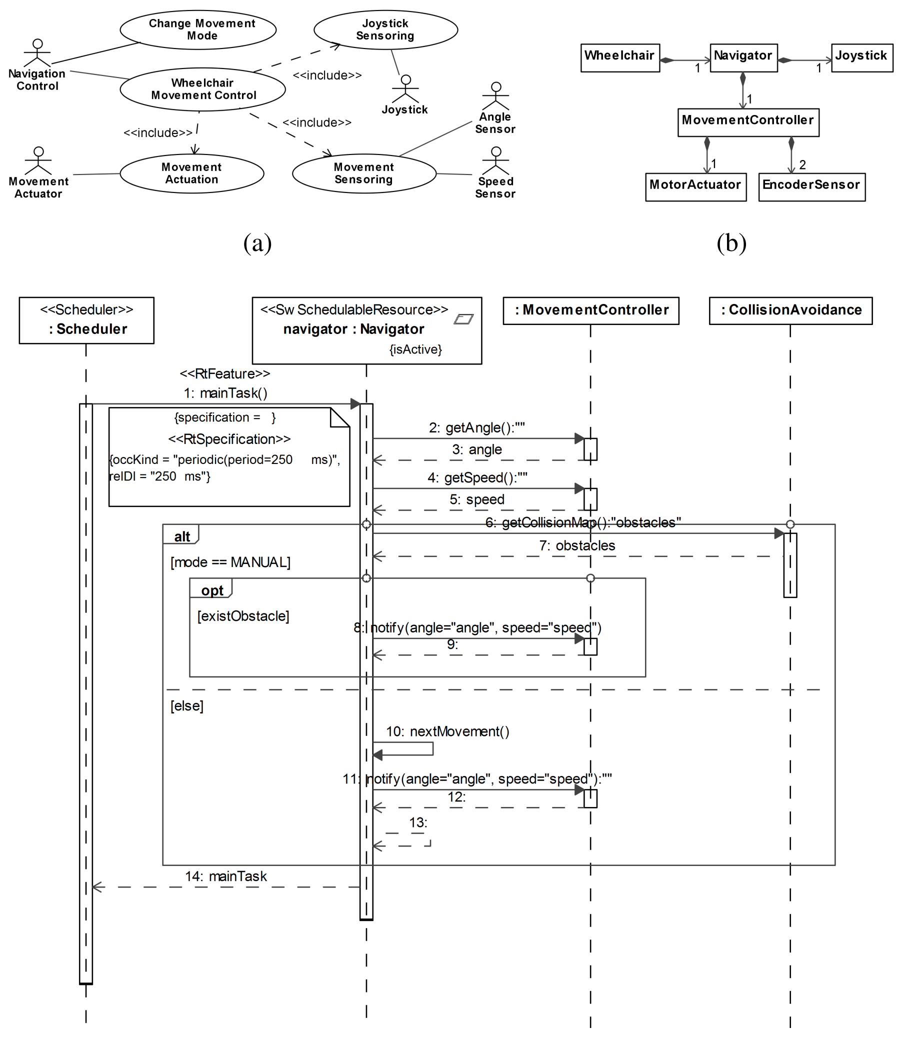
The analysis model is specified using the elements that represent concepts in the problem domain, such as actuators, sensors, and controllers of a system. The FRs and NFRs should be mapped into these elements, avoiding specific solution concepts, such as processors, task, and platform specific information. The solution model is obtained during the design phase, which searches for the best software and hardware architecture that supports the FRs and NFRs specified during the analysis phase. Following this approach, the design model becomes more abstract, thus opening more alternatives for DSE.

The application must contain structural and behavioral models. The structural model is specified by using Class diagrams. In this diagram, besides the definition of the application classes, the designer must also identify upper bounds for the multiplicity of vectors and matrix fields and objects, such that the evaluation tools can improve the estimation. The behavioral model is defined using Sequence diagrams and its fragments, such as `loop`, reference to other Sequence diagrams - `ref`, and alternative execution - `alt`. It specifies interaction between objects and dependencies between execution scenarios. Additionally, upper and lower bounds of loop fragments must be specified to improve estimation. Figure 5.3(a) shows examples of the diagrams used to specify the functional requirements and system application. Figure 5.3(b) shows a class diagram and Figure 5.3(c) shows a Sequence diagram realizing the Movement Actuation use case presented in Figure 5.3(a). Interaction of objects with actors indicates inputs/outputs from/to system devices.

Models are decorated with MARTE stereotypes to add specific design decisions or NFR information. According to the MARTE specification, many NFRs can be specified, using the MARTE «Nfp» stereotype to derive all the desired NFR requirements. The adopted method refers to `NFP_Real` data type from the MARTE library, which defines a tuple value and unit, in order to read the information required to prune the design space. In the current prototype, the «SwSchedulableResource» stereotype is adopted to identify an active object, while the «RtFeature» stereotype is used to refers to a real-time specification. Such a specification is a comment annotated with the stereotype «RtSpecification»

[3]http://beru.univ-brest.fr/ singhoff/cheddar/contribs/examples_of_use/00readme.html

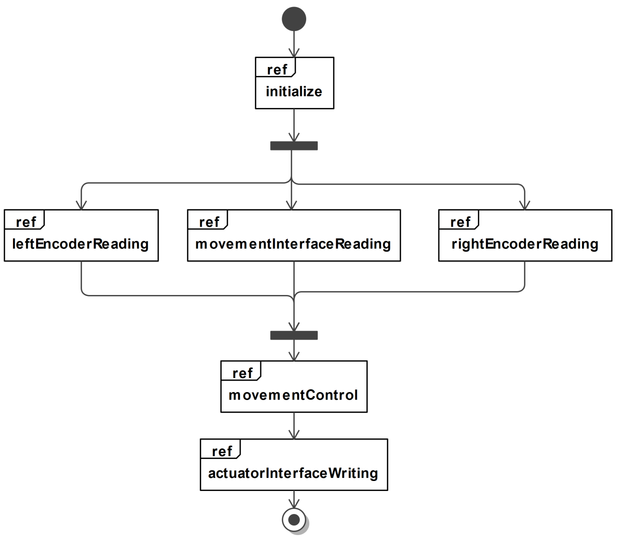
Figure 5.2: System specification using UML: (a) Use case diagram; (b) Class diagram; (c) Sequence diagram.



and provide properties to define if the feature is periodic, aperiodic, or sporadic and has attributes to allow the annotation of relative deadline `relDL`, absolute deadline `absDL`, acceptable rate of missing deadline `miss`, and occurrence `occKind`. Figure 5.3(c) also shows the application of MARTE stereotypes.

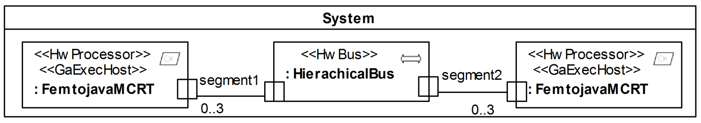
An Interaction Overview diagram identifies and links scenarios specified by using Sequence diagrams, in order to evaluate the system during the estimation process. Figure 5.3 shows an Interaction Overview diagram example. The diagrams specifies that the system execution starts by executing the `initialize` scenario, which among other actions starts different system threads. Thereafter the parallel execution of three scenarios are identified, namely `leftEncoderReading`, `movementInterfaceReading`, and `rightEncoderReading`. The specific execution order of these scenario may not be important for the evaluation, or it is open to the evaluation tool to figure out the cost of scheduling overhead based on the final mapping defined in the Component diagrams or by the DSE tool. The `movementControl` scenario depends on the execution of the three previous scenarios and must execute after them and must be followed by the `actuatorInterfaceWriting` scenario, which closes the complete evaluation scenario.

Figure 5.3: Example of an Interaction Overview diagram.

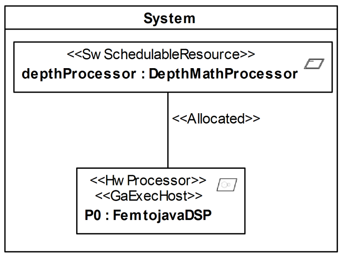


Architectural components, such as processing units, memories and communication buses, are defined in Composite diagrams and are also decorated with related MARTE stereotypes for each of these components. The Composite diagram is also used to define rules for architecture allocation and mapping of applications into the architecture. Figure 5.5(a) shows a Composite diagram specifying an architecture with up to six processors interconnected by a hierarchical bus with two segments, and Figure 5.5(b) shows an example specifying in which processing unit a software element must execute. These and others model patterns are used to create DSE Rules in the DSED model.

Figure 5.4: System specification using UML: (a) Composite diagram; (b) Processor allocation and mapping constraints.



(a)



(b)

Although limited constructs are used, such diagrams allow the extraction of many information used to automate the DSE and evaluation processes. Currently, from these diagrams interaction, task, processor, and communication graphs are extracted. Moreover constraints, configuration parameters, and DSE scenarios are also extracted with some limitation. More details on the implementation and data extracted from UML models are explained in Chapter 7. Such modeling method supports automated creation of DSE Mapping and DSE Scheduling problems and weaving of the DSED model with UML model. Other DSE problems are supported through manual Design-DSED model weaving and DSED specification.

5.4 Design Space Exploration Domain Modeling

Before starting the DSE it is necessary to determine which design activities must be executed. On these information depends the specification of the DSE scenarios, such as objectives to be optimized, the values allowed to be assigned to variables (variable domain), etc. Different design activities may require different evaluation and solver tools, and so these tools must be determined and configured accordingly.

The identification of DSE concepts commonly used in different DSE scenarios is important to create a standard representation, which can be applied to different DSE scenarios, so that the flexibility, reusability and automation of DSE process can be improved. The work presented in (SAXENA; KARSAI, 2011) suggests a classification of design activities for DSE into six problems, namely Configuration, Construction, Mapping, Placement, Routing, and Scheduling. Based on the bibliographical review presented in Chapters 2 and 3, this methodology adopted the same classification and from now on, this classification is referenced as DSE Problems. The inter-dependencies between design activities, added to the interleaving aspects of DSE Problems, for example placement and routing problems, or mapping and scheduling, require adequate methods to represent the DSE Problems. Furthermore, such relationship must also be handled adequately. Therefore, four from these six problems were selected to be defined and modeled in the proposed methodology. The selected DSE problems are Configuration, Construction, Mapping, and Scheduling. Although Placement and Routing are optimization problems which lead to DSE , these two problems are out of the scope of this thesis, because no work was found in the literature, which propose system-level development (e.g. Y-Chart or PBD) considering such problems.

The fundamental stone for the whole DSE methodology is the formal definition of the four selected DSE problems, in order to provide a strict semantic for elements that represent such problems. First, each problem is represented as a tuple of elements, which are required to model the problem in an adequate and flexible way. When required, the definition of a problem includes also constraints to assure a well-defined problem. Thereafter, a set of common constraints are identified for each DSE problem, such that an engineer can define DSE scenarios, i.e problem instances, containing special requirements according to design models, DSE goals, and the applicable constraints. In the problem model is also defined a solution place holder, to which a solution for the problem must conform by fulfilling all constraints specified for a DSE scenario.

In order to facilitate the specification of the information required to represent adequately different DSE scenarios, the DSE Domain (DSED) metamodel was defined to capture all relevant concepts of the DSE domain. It is defined based on a mapping of the formal definition of each DSE problem into a composite representation in ECORE. Therefore, it composes elements of all DSE problems into a unified model, so that it represents the four DSE problems, respective solutions, design decisions and alternatives, evaluation tools, objective metrics, constraints, and other concepts, required to provide flexible and automated DSE supported by the MDE technology.

There are different alternatives to specify a model conforming to the proposed DSED metamodel. One can manually create and edit a DSED model, by using generic model editors produced by the MDE framework from the DSED metamodel. In this way, the DSE model is created without any reference to design models, although it requires manual specification of the model and later a manual link of design and DSED models, both are error prone and counter productive tasks. Another possibility is to use model transformations to automate the creation/editing of a DSED model and the link of it to design models. The model transformation is responsible for extracting design information, such as task graph, feature model, communication graph and others from the design models used. In this approach, the automation is improved, but it is dependent on the modeling languages used in the development process, so that a transformation must be implemented for each required design language.

Another alternative would be the additional definition of a concrete syntax for the DSED metamodel, so that one could specify DSED models by writing text or even using graphical means by using a DSED language. In favor of the development of transformations from different design languages into the DSED model, and to improve automation, this work does not define a concrete syntax for the DSED metamodel. Although a concrete syntax may ease the specification of DSE rules/constraints, the proposed methodology abstains from defining one, in order to reuse the concrete syntax of a well adopted language, improving the abstraction and the reuse of development artifacts.

Moreover, in the proposed DSE methodology, common constraints applied to DSE problems are identified and their semantics are formally defined in the DSE problem models. Such constraints are mapped to elements in the DSED metamodel, which represents these constraints in an abstract way. Therefore, instead of complex metamodel defining operators and terms to specify any kind of constraints, such as OCL (OMG, 2006), FORMULA(KANG; JACKSON; SCHULTE, 2011), CSL (SAXENA; KARSAI, 2010), it was opted to use one abstract construct to represent each well-defined constraint of the DSE problem model. These abstract constraints, which have strict semantics, are later implemented in a library of model transformation rules in a specific language, for example FORMULA or VIATRA II, according to the semantics specified in the DSE problem model. In this way, such abstract constraint elements in the DSED have strict semantics, while they are independent of an implementation in a concrete language. Therefore, this approach improves the DSE process by bridging abstract constraints to a reusable library of constraints, so that an engineer is not required to master transformation languages, and implement constraints at each new DSE scenario.

The following sub-sections present formal definitions for each DSE problem and the constraints associated to these DSE problems, including the semantic to be applied when these constraints are represented in a DSED model. The equivalent representation of these definitions in the DSED metamodel is organized in seven UML Class diagrams, which are presented after each DSE problem definition to facilitate the description of its concepts. Some elements are replicated in different diagrams, so that a global view of the metamodel is made easier.

5.4.1 DSE Domain Core Model

DSE is an optimization problem focused on the design of systems, hence concepts that commonly appear in different optimization problems also appear in DSE problems, such as problem, solution, solver, evaluation of solutions, constraints and objectives to be optimized. Although different DSE problems share common concepts, each problem has its own specificity, which requires additional concepts. Therefore, a DSE Domain Core model is defined to represent the concepts shared by the four DSE problems represented. This core model is the fundamental model for the formulation of specific DSE problems and the specification global constraints, so that all other models of DSE problems extend or specialize this core model.

The DSED core model is a tuple that contains a DSE scenario Sc, a set of solvers So and another set of evaluators Ev, which are used to generate and evaluate solutions to problems defined in the DSE scenario. So and Ev have parameters Pa, which configure their algorithms. The DSED core model has also a set of available metrics M supported by the available evaluations, a function ϕ that must be implemented by solvers, and a function γ that must be implemented by evaluators. The function ϕ maps problems P to candidate solutions Sl and the function γ annotates costs Co, according to the required metrics, to solutions, which are sets of decisions d. A scenario is composed of a set D of design graphs Dg and a set of problems P. A problem P represents the optimization problem instance, which is composed of a subset $D\prime$ of D, which contains the set of design graphs defined for a scenario. P contains also a set of required metrics Rm and a set of constraints Cs, which are optionally defined by engineers to create different scenarios for the same problem. The DSED core model is defined as following:

Definition 5.1 (DSE Domain core model)**:**
$DSED = \langle Sc\left\langle D\left(Dg\right), P\left\langle D\prime, Sl\left(d\right), Rm, Cs\right\rangle\right\rangle, So\left\langle Pa\right\rangle, Ev\left\langle Pa\right\rangle, M, \phi, \gamma\rangle$ is a DSE domain model, where:

$D\left(Dg\right)$ is the set of design graphs $Dg_1 \ldots Dg_n$, such that Dg is a directed graph as presented in the Definition 4.1 and generated from design models, e.g. task graph, architectural graph, parameter dependence graph.

$Sl\left(d\right)$ is a place holder for solutions, which must fulfill the set of constraints Cs specified for a problem P, such that design decisions $d \in Sl$ can solve a problem P.

Rm are the required metrics used to qualify solution $s \in Sl$. Two set of metrics are identified, namely the set of required objectives Ro to be optimized and required metrics Rg used to guide the DSE, such that $Ro \subseteq Rm$, $Rg \subset Rm$ and $Ro \cap Rg = \emptyset$.

Cs is a set of constraints specified by an engineer to be applied, when solving a DSE problem P.

P is the set of DSE problems to be solved for a $D\prime\left(Dg\right) \mid D\prime \subseteq D$, which elements can be one of the following $\{P_{conf}, P_{const}, P_{mapping}, P_{sched}\}$.

Sc is a DSE scenario consisting of $D\left(Dg\right)$ and P, which can be solved for all $p \in P$.

So is the set of available solver methods to optimize the problems $p \in P$ that implements the function ϕ. It contains parameters Pa, which configure the solver method to be executed.

Ev is the set of available evaluation methods to qualify solutions $s \in Sl$ that implements the function γ. It contains parameters Pa, which configure the evaluation method to be execured.

M is a set of metrics, $m \in M$ and $\exists\ ev \in Ev \mid m$ can be calculated.

ϕ is a function, such that $\phi : P \to Sl$ that maps the problem $p \in P$ into solutions $s \in Sl$.

γ is a function, such that $\gamma : Sl \to Co$ that annotates the costs $c \in Co$ for a solution $s \in Sl$.

This model represents concepts shared for many optimization problems and can be applied to represent a general DSE problem to be solved by global optimizers. The set of design graphs provide an abstract representation of the design, which can be used to represent different design information, and at the same time it is easily codified to different solvers, such as Genetic Algorithm, Simulated Annealing and SAT solvers. However, it does not provide problem specific information, which could be used by heuristics in order to provide search mechanisms customized for a specific DSE problem. Therefore, other four DSE problem models are defined in the next sections.

All optimization problems may be subject of constraints, which define the search space and the conformity of found solutions. Common required constraints, such as maximum and minimum value for the metrics and objectives, are defined, so that an engineer can optionally constrain the DSE problem according to the specificity of the DSE scenario. The following constraints are defined in the DSE Domain models, so that solutions must conform to this constraints, if they are specified by a engineer to be applied to a specific DSE scenario:

- *Maximum Value*: Defines the upper bound c_{max} for a value that can be assigned to $c \in Co$. Using this rule one can deny the assignment of an out of range value after the evaluation of a system property or deny values that does not fulfills requirements:

$$c \leq c_{max} \tag{5.1}$$

- *Minimum Value*: Similar to Constraint 5.1, it defines lower bound c_{min} applied to c:

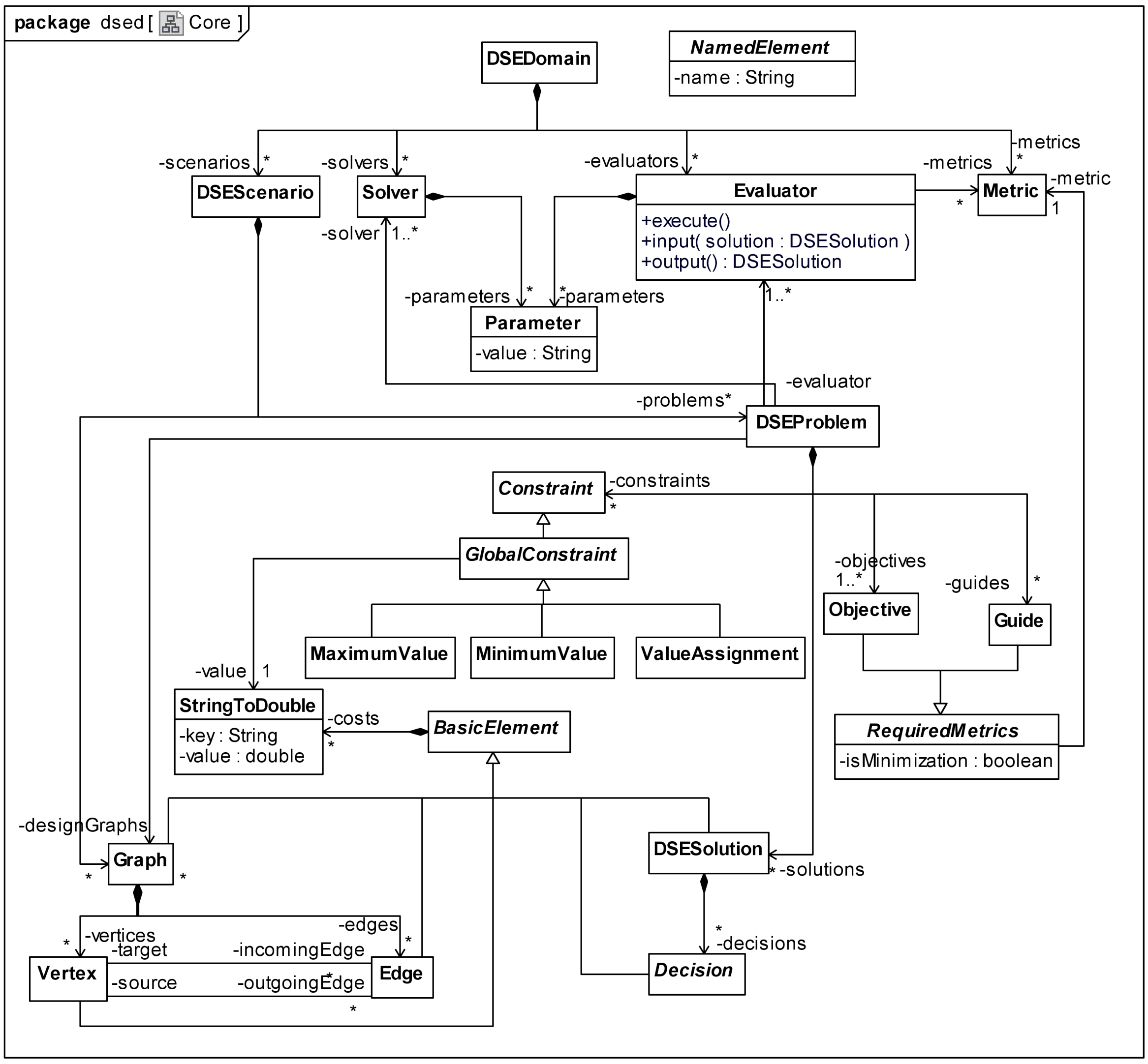
$$c \geq c_{min} \tag{5.2}$$

- *Value Assignment*: Defines the value $c_{assigned}$ to be assigned to c, in order to prune the design space or assume a specific scenarios:

$$c = c_{assigned} \tag{5.3}$$

The DSED core model proposed in Definition 5.1 is mapped into the DSED metamodel as presented in Figure 5.5. Such representation aims to capture the general DSE concepts and provides the basic elements to define more specialized DSE scenarios, as described in the next sections.

Figure 5.5: Design Space Exploration Domain Metamodel - Core.



The root container in the metamodel presented in Figure 5.5 is DSEDomain ($DSED$), which is a container for all elements related to DSE, representing the DSED model. It inherits properties from NamedElement like all other elements in this metamodel, so that these elements have a name to be identified. The generalizations to NamedElement were omitted to keep diagrams clear. The DSEDomain is composed of a set of Evaluator (Ev) elements, which represent the available tools to evaluate candidate designs that implement the function $\gamma : Sl \rightarrow Co$. Evaluator contains Parameter (Pa) that stores the information required to execute an Evaluator, such as name, path and configuration. An Evaluator is also associated to the Metrics (M) it can provide, so that multiple Evaluators may be used to evaluate the same design candidate. A Solver (So) represents the available mechanism, which implements a function $\phi : P \rightarrow Sl$ that interacts with the DSED model in order to solve a Problem (P). In the same way as in Evaluator, it also has Parameters to store the required information to execute the solver algorithm. DSEDomain also contains Scenario (Sc), which has a set of Problems (P) and a set (D) of design graphs Graphs (Dg) that represent design elements involved in the problem. A `Problem` is a general representation for a DSE problem, which is specialized in the four DSE Problems approached in this thesis. Each `Problem` specialization is described in the following sections. `Problem` contains a subset ($D\prime$) of `Graphs` contained in the `Scenario`, referencing only the design graphs that are subject of this problem. A `Problem` also contains the `RequiredMetrics` (Rm), which defines the metrics adopted to evaluate candidate designs. These metrics can be `Objectives` (Ro) to be optimized or `Guide` metrics (Rg) to be satisfied or are used to help the engineer in some trade-off analysis. Selected `Solvers` are also associated to each specific `Problem`. Each problem is subject to `Constraints` (Cs), which can be specialized for each type of problem. `Constraint` is an abstract concept, which must be specialized for the specific constraint to be addressed. `GlobalConstraints` are specialization of `Constraint`, which are applied to any `BasicElement`. Currently, the following constraints are defined: `MaximumValue`, `MinimumValue`, and `ValueAssignment`, which represent the Constraints 5.1, 5.2, and 5.3 respectively. `Solution` (Sl) represents a candidate design. A function ϕ maps `Solutions` that have `costs` (Co), annotated by the function γ in an estimation/simulation process. `Solution` also contains a list of `Decisions` (d), which identifies each individual decision that is part of a solution and has `costs` too.

The four DSE problems defined in the following sections derive from `Problems`, so that they share the common resources defined in the core of the DSED metamodel.

5.4.2 Configuration Problem

A configuration problem in embedded systems is a generalization of the platform tuning, which is also known as parametrization or parameter tuning problem explained in Chapter 3, because the configuration can be solved for any component in the system besides the platform, and at any abstraction level. Examples of configuration problems are described in (GIVARGIS; VAHID, 2002; ASCIA; CATANIA; PALESI, 2004). In this problem, the engineer must define values for configurable properties of components, such as cache size and bus width, for a hardware architecture, or sample rate and resolution for some software components. The values assigned to properties are usually restricted to a variable domain, which is the interval of values a property can have.

The Configuration Problem P_{conf} is a specialization of the Problem P defined in the $DSED$ core model. Such a specialization is achieved by the specification of constraints that provides additional semantics to elements of P, so no additional elements are included in P. These constraints Cs are used to define the values that can be assigned to configurable properties/parameters Pr of vertex contained in the design subset $D\prime$ of design graphs. The properties with assigned values that fulfill the constraints form a solution, which is stored in the set of candidate solutions Sl. Definition 5.2 presents the complete definition of Configuration problem, based on the work in (GIVARGIS; VAHID, 2002):

Definition 5.2 (DSE Configuration Problem):
$P_{conf} = \langle P \langle D\prime, Sl\,(Pr)\,, Rm, Cs \rangle \rangle$ is a DSE Configuration Problem, which specializes the Definition 5.1 as follows:

P is the general DSE problem as specified in Definition 5.1.

Cs is a set of constraints specified by an engineer to be applied, when solving P_{conf}, in order to define the variable domain Do, which is an interval $[l; u]$, such that $Do \subset \mathbb{Z}$ or $Do \subset \mathbb{R}$ and Do are the set of values which So assign to Pr.

Pr is the set of configurable properties $pr_1, pr_2..., pr_n \in Pr$ associated to a design element $v \mid v \in Dg\,(V)$.

$Sl\,(Pr)$ is a place holder for solutions, which must fulfill the set of constraints Cs specified for the problem, such that the values assigned to $Pr \in Sl$ can solve the problem P_{conf}.

The DSE Configuration problem model can represent multiple configuration problems at the same time, if it contains multiple design graphs. So that one can set up a configurable SoC architecture and application parameters simultaneously. Moreover, the graph representation can be used not only to identify components and their parameters, it can also use multiple graphs to specify interdependency between properties, as proposed in (GIVARGIS; VAHID, 2002), so that the heuristics can exploit such information when solving the DSE Configuration problem. For example, the optimization of properties of a data cache is independent of properties of an instruction cache, however they are dependent of bus' parameters.

In order to assure a finite set of values, i.e. variable domain Do, which can be assigned to a property Pr, an engineer can specify the following constraints:

- *Property Lower Bound*: Defines a lower bound to be applied to pr:

$$pr \geq pr_{lower} \tag{5.4}$$

- *Property Upper Bound*: Defines an upper bound to be applied to pr:

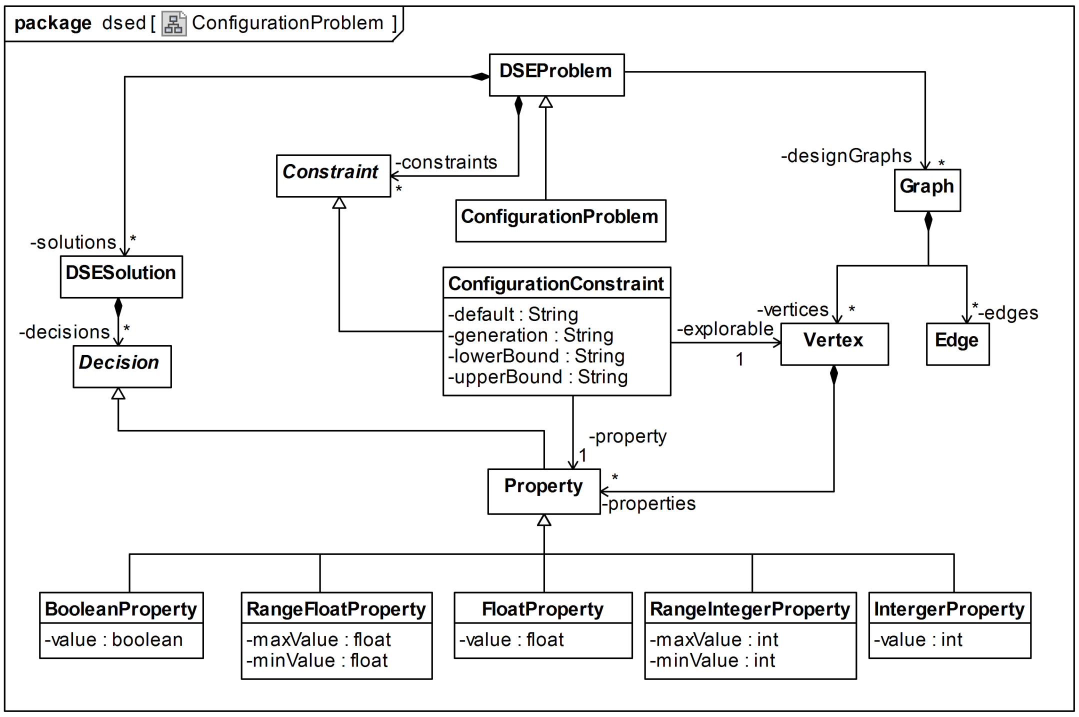
$$pr \leq pr_{upper} \tag{5.5}$$

- *Property Value*: Defines the value to be assigned to pr, in order to prune the design space or assume some DSE scenario, for example if two independent properties are to be optimized, the value of one can be arbitrarily fixed, when computing the Pareto-set of values for the other property:

$$pr = pr_{assigned} \tag{5.6}$$

The Configuration Problem model presented in Definition 5.2 is mapped to the metamodel illustrated in Figure 5.6. The `ConfigurationProblem` (P_{conf}) element represents the Configuration problem defined in 5.2. It contains a list of `Configuration-Constraints` (Cs), which define the finite set of values (variable domain) that can be assigned to a `Property` (Pr), according to the Constraints 5.4, 5.5, and 5.6, respectively. A `ConfigurationConstraint` also has a generation function for the value of a `Property`. A `Property` is associated to a `Vertex` and represents the final value to configure it.

Figure 5.6: Design Space Exploration Domain Metamodel - Configuration Problem.

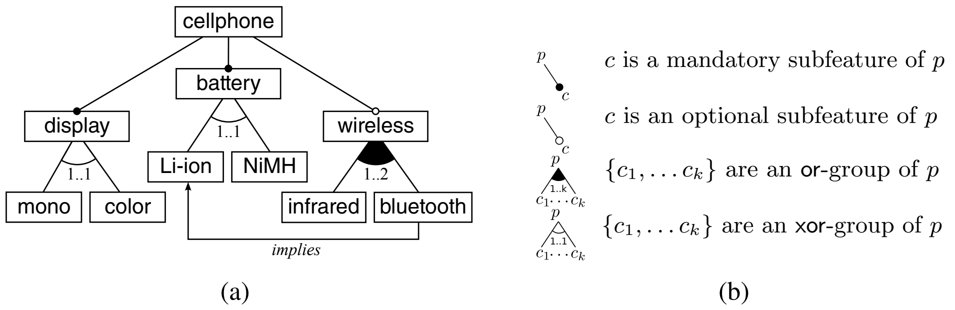


5.4.3 Construction Problem

Construction is a typical problem found in Product Family Engineering, which is the engineering process based on the variability of common features in a family of products(CZARNECKI, 1998). It is also one of the problems in the Platform-based Design, as features of the product in development are assembled from a library of components provided by the platform. An example of this problem in the context of DSE is described in (NEEMA et al., 2003; ANDERSEN et al., 2012). In a DSE Construction problem building blocks are assembled from a library of blocks, in order to build a product and optimize some product's properties. These building blocks may have dependencies, so that the selection of one block from the library may imply in the selection or exclusion of other blocks to/from the solution. These dependencies are represented in form of a tree, so that the root of the tree represents the system to be built and leafs are building blocks available in the library. Intermediate vertices of the tree represent groups of features with blocks in their leafs. Groups may have different rules for composition, for example all leafs in a group may be mandatory, optional, or mutualy exclusive. Figure 5.7 illustrates a feature model, which is a common model used to represent a construction problem in Product Family Engineering, and the concrete syntax used to define the diagram.

The Construction Problem P_{const} specializes and extends P, by providing additional semantics and elements to classify and group edges of design graphs. The Construction Problem is composed of the subset $D\prime$ of design graphs and the placeholder Sl, which is the set of selected vertices Sd of the design graphs that fulfills the constraints and solve the problem P_{const}. P_{const} is also composed of the required metrics Rm and constraints Cs specialized for this problem. It also extends P by including the sets E_m, E_o, E_i and E_x that group edges of design graphs contained in $D\prime$ according to the dependency between the vertices connected by these edges. Finally P_{const} has also the sets G_o, G_m, G_x, which are set of group of edges that imply some construction constraints between

Figure 5.7: Product Family Engineering: example of feature model: (a) Feature model; (b) Concrete syntax used in the diagram.


cellphone
battery
display
wireless
1..1
1..1
Li-ion
NiMH
1..2
mono
color
infrared
bluetooth
implies
(a)
c is a mandatory subfeature of p
c is an optional subfeature of p
{c1, ... ck} are an or-group of p
{c1, ... ck} are an xor-group of p
(b)

(ANDERSEN et al., 2012)

multiple vertices. The Construction Problem is completely defined as following, based on the definition in (ANDERSEN et al., 2012):

Definition 5.3 (DSE Construction Problem)**:**
$P_{const} = \langle P \langle D\prime, Sl\,(Sd)\,, Rm, Cs\rangle\,, \langle E_m, E_o, E_i, E_x\rangle\,, \langle G_o, G_m, G_x\rangle\rangle$ is a DSE Construction Problem, which extends the Definition 5.1 as follows:

P is the general DSE problem as specified in Definition 5.1.

$D\prime$ contains one design graph Dg, which is a rooted tree connecting all building blocks, such that $Dg\,(V)$ is a finite set of building blocks available to construct a system, and $Dg\,(E)$ is a set of directed edges indicating the relation child-parent.

E_m is a set of mandatory edges, such that $E_m \subseteq Dg\,(E)$.

E_o is a set of optional edges, such that $E_o \subseteq Dg\,(E)$.

E_i is a set of cross-tree "implies" edges, such that $E_i \subseteq Dg\,(V) \times Dg\,(V)$ and $E_i \cap Dg\,(E) = \emptyset$.

E_x is the set of cross-tree "excludes" edges, such that $E_x \subseteq 2^{Dg(V)}\ \forall\ e \in E_x,\ |e| = 2$.

G_o is the set of groups $g_{oi} \in G_o$ of edges $e_j \in Dg\,(E)$, such that $i = \{1..\,|G_o|\}$, and $j = \{1..\,|g_o|\}$. A g_o group represents an or-group, such that one or more blocks in the group can be selected.

G_m is the set of groups $g_{mi} \in G_m$ of edges $e_j \in Dg\,(E)$, such that $i = \{1..\,|G_m|\}$, and $j = \{1..\,|g_m|\}$. A g_m group represents a mutex-group, such that one block or none can be selected from the group.

G_x is the set of groups $g_{xi} \in G_x$ of edges $e_j \in Dg\,(E)$, such that $i = \{1..\,|G_x|\}$, and $j = \{1..\,|g_x|\}$. A g_x group represents an xor-group, such that one block of the group must be selected.

Sd is a label associated to building block $v \in Dg\,(V)$ through the function $\epsilon : Sl \rightarrow Dg\,(V)$ and represents the selected block to constitute the system under construction.

$Sl\,(Sd)$ is a place holder for solutions, which must fulfill the set of constraints Cs specified for the problem, such that the selected blocks $Sd \in Sl$ can solve the problem P_{const}.

A well formed Construction problem is subject to the following constraints:

$$G_o \cap G_x \cap G_m = \emptyset \tag{5.7}$$

$$E_k \cap E_j = \emptyset \mid \forall\, k, j = \{m, o, i, x\} \;\wedge\; k \neq j \tag{5.8}$$

Additionally, all edges in a group share the same parent so:

$$\textit{if}\; g \in G_i \;\forall\; i \in \{o, m, x\} \;\textit{and if}\; e = \langle b_1, b_2 \rangle\,, e\prime = \langle b_3, b_4 \rangle \in g \;\textit{then}\; b_1 = b_3 \tag{5.9}$$

The DSE Construction Problem is subject to the following constraints, which provide semantic for edges and group of edges presented in Definition 5.3 and must be fulfilled when selecting blocks to compose a solution for the problem:

- *Mandatory Element*: This constraint defines that a mandatory element $v \in Dg(V)$, which is target of edge $e \in E_m$, must be selected to build a solution Sd:

$$Sl := Sl \cup \{Sd\} \;\mid \delta_{tDg}\,(e) = v\prime \;\wedge\; e \in E_m \;\wedge\; \epsilon\,(Sd) = v \;\wedge\; v = v\prime \tag{5.10}$$

- *Optional Element*: This constraint defines that an optional element $v \in Dg(V)$, which is target of edge $e \in E_o$, may not be selected to build a solution Sd:

$$\textit{if}\, \delta_{tDg}\,(e) = v \wedge e \in E_o \wedge \epsilon\,(Sd) = v\prime \wedge v = v\prime \qquad \Rightarrow \qquad Sl \cap \{Sd\} = \begin{cases} \emptyset \\ \{Sd\} \end{cases} \tag{5.11}$$

- *Implies Element*: This constraint defines that if a block $v \in Dg(V)$ is selected to compose a solution Sl, and this elements implies another element, then this element must also compose the solution:

$$\begin{gathered} \textit{if}\,\exists\; e = \langle v, v\prime \rangle \;\wedge\; Sd_i \in Sl \mid \epsilon\,(Sd_i) = v \;\wedge\; e \in E_i \;\Rightarrow \\ \exists\; Sd_{i+1} \in Sl \mid \epsilon\,(Sd_{i+1}) = v\prime \end{gathered} \tag{5.12}$$

- *Excludes Element*: On the contrary to Constraint 5.12 this constraint defines that if a vertex $v \in Dg(V)$ is selected to compose a solution Sl, and this element excludes another element, than the excluded element must not appear in the solution:

$$\begin{gathered} \textit{if}\,\exists\; e = \langle v, v\prime \rangle \;\wedge\; Sd_i \in Sl \mid \epsilon\,(Sd_i) = v \;\wedge\; e \in E_x \;\Rightarrow \\ \nexists\; Sd_{i+1} \in Sl \mid \epsilon\,(Sd_{i+1}) = v\prime \end{gathered} \tag{5.13}$$

- *Or Group*: This constraint defines that one or more blocks in the Or-group must be selected for the solution, such that:

$$\begin{gathered} \textit{if}\, \exists\, Sd \in Sl \;\wedge\; \epsilon\,(Sd) = u \;\wedge\; e_i = \langle u, v_i \rangle \in g_o \mid i = \{1..\,|g_o|\} \;\wedge\; g_o \in G_o \;\Rightarrow \\ Sl := Sl \cup (\{Sd_1\} \;\vee\; \cdots \;\vee\; \{Sd_n\}) \mid \\ \epsilon\,(Sd_i) = v_i \;\wedge\; e_i = \langle u, v_i \rangle \in g_o \;\wedge\; n = |g_o| \;\wedge\; i = \{1..n\} \end{gathered} \tag{5.14}$$

- *Mutex Group*: This constraint defines that none or one block from the Mutex-group can be selected for the solution, such that:

$$\begin{gathered} \textit{if}\, \exists\, Sd \in Sl \;\wedge\; \epsilon\,(Sd) = u \;\wedge\; e_i = \langle u, v_i \rangle \in g_m \mid i = \{1..\,|g_m|\} \;\wedge\; g_m \in G_m \;\Rightarrow \\ \begin{cases} Sl := Sl \\ Sl := Sl \cup \{Sd\prime\} \mid \epsilon\,(Sd\prime) = v \;\wedge\; e = \langle u, v \rangle \in g_m \end{cases} \end{gathered} \tag{5.15}$$

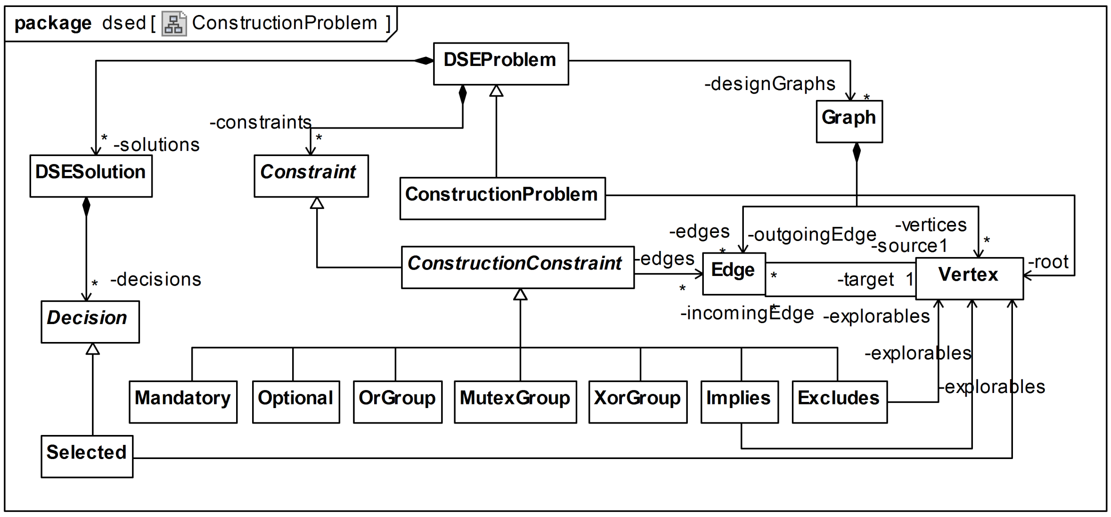
- *Xor Group*: This constraint defines that one block from the Xor-group must be selected for the solution, such that:

$$\begin{gathered} \textit{if}\, \exists\, Sd \in Sl \;\wedge\; \epsilon\,(Sd) = u \;\wedge\; e_i = \langle u, v_i \rangle \in g_x \mid i = \{1..\,|g_x|\} \;\wedge\; g_x \in G_x \;\Rightarrow \\ Sl := Sl \cup (\{Sd_1\} \;\vee\; \cdots \;\vee\; \{Sd_n\}) \;\wedge\; \neg\,(Sd_i \;\wedge Sd_j)\; \forall i \neq j \mid \\ \epsilon\,(Sd_i) = v_i \;\wedge\; e_i = \langle u, v_i \rangle \in g_x \;\wedge\; n = |g_x| \;\wedge\; i, j = \{1..n\} \end{gathered} \tag{5.16}$$

The DSE Construction Problem presented in Definition 5.3 provides elements to represent different dependencies, requirements, and alternative ways to assemble design elements. It is suitable not only to construct systems based on the application features, but also to construct systems by selecting reusable hardware and software components from a platform repository. Moreover, it is compatible to many feature models, so that solvers specialized for feature models can be integrated into the DSE methodology. Compared to other feature models found in the literature, Definition 5.3 extends such models to include not assembly of building blocks, but also optimization of systems properties. Figure 5.8 illustrates the DSED metamodel elements specified to represent a DSE Construction Problem.

The DSED metamodel represents the problem defined in Definition 5.3, by using the element `ConstructionProblem` (P_{const}), which extends `DSEProblem` (P). It contains a list of `Construction Constraints` (Cs) that are associated to `Edges` (E) of design graph `Graph` (Dg). Such constraints add the construction semantics to `Edges` and determine how a `Vertex` (V) can be selected to construct a solution. The elements `Mandatory` (E_m), `Optional` (E_o), `Implies` (E_i), `Excludes`(E_x), `OrGroup` (G_o), `MutexGroup` (G_m) and `XorGroup` (G_x) are associated to `Edges` and give them special semantics according to the Constraints 5.10 - 5.16, respectively. A `Vertex` is a building block. Finally, the `Selected` (Sd) element indicates the selected block to construct a `Solution` (Sl) for the Construction Problem.

Figure 5.8: Design Space Exploration Domain Metamodel - Construction Problem.



5.4.4 Mapping Problem

In modern embedded system development, mapping is the most common problem for DSE at the system level. As most DSE approaches are based on the Y-Chart, a mapping step is required, so that an application is mapped into a platform to define a system implementation. A mapping can define tasks which must be executed on a processor, services required by an application to components provided by a platform library, distribution of messages through a communication channel, and other possibilities.

In most of system mapping methods, the system is represented by two or more graphs, which must be mapped into each other by specifying extra edges that indicate the mapping between vertices of different graphs. In the same way the DSE Mapping Problem is defined by a set of graphs, which must be mapped by using tuples that contain vertices of different graphs. The mapping solutions are searched, so that some system metrics are optimized. The Mapping Problem $P_{mapping}$ specializes the problem P, by constraining the semantics of its elements and is defined as follows:

Definition 5.4 (DSE Mapping Problem)**:**
$P_{mapping} = \langle P \langle D\prime, Sl\left(Mp\right), Rm, Cs \rangle \rangle$ *is a DSE Mapping Problem, which specializes the Definition 5.1 as following:*

P is the general DSE problem as specified in Definition 5.1.

Mp is a pair $\langle u, v \rangle$, where $u \in Dg_i\left(V\right) \wedge v \in Dg_j\left(V\right) \mid i \neq j \wedge Dg_i, Dg_j \subset D\prime$, which represents the mapping from vertex u into vertex v of different design graphs.

$Sl\left(Mp\right)$ is a place holder for solutions that must fulfill the set of constraints Cs specified for the problem, such that the selected mappings $Mp \in Sl$ can solve the problem P_{const}.

This model can capture design elements represented in different types of MoC, such as CDFG, KNP, Signal Flow Graph, and others. Furthermore, it allows different encoding of graphs, so that different solvers can be used, for example integer representation of tasks and processors in genes of a Genetic Algorithm (DICK; JHA, 1998), activation mapping edges (0 or 1) between graphs (BLICKLE; TEICH; THIELE, 1998) or sub-graph of product of graphs (OLIVEIRA et al., 2009), which is proposed in this thesis. Moreover, it represents dependencies between elements of the same graph, so that dependencies can be used by heuristics, in order to improve generation of solutions. Such an approach is proposed in Chapter 6.

In order to specify different DSE scenarios for the DSE Mapping Problem, some constraints are defined. By using these constraints, engineers can influence solutions by including mapping they knows that are adequate, or excluding others. Moreover, constraints to specify structural restrictions, such as when a software component cannot be executed in a specific processor, all elements from a graph must be mapped, or elements cannot be mapped twice or more. Constraints that represents dependencies between elements are also defined, such as the implication of a mapping when another mapping is selected in the solution. The constraints that can be specified for DSE Mapping Problem are defined as follows:

- *Duplicated Mapping*: Avoids mapping the same vertex in a graph twice to the same vertex in another graph:

$$\begin{gathered} Mp_i = \langle u, v\rangle \neq Mp_j = \langle u\prime, v\prime\rangle \;\forall\; Mp_i, Mp_j \in Sl\,(Mp) \;\mid \\ i, j = \{1..\,|Sl\,(Mp)|\} \;\wedge\; i \neq j \end{gathered} \tag{5.17}$$

- *One To Many Mapping*: Avoids mapping one vertex in one graph to many vertices in another graph:

$$\begin{gathered} \textit{Let}\; u, u\prime \in Dg\,(V) \;\wedge\; v, v\prime \in Dg\prime\,(V) : \\ \nexists\; u = u\prime \;\wedge\; v \neq v\prime \;\forall\; Mp_i = \langle u, v\rangle, Mp_j = \langle u\prime, v\prime\rangle \in Sl\,(Mp) \;\mid \\ i, j = \{1..\,|Sl\,(Mp)|\} \;\wedge\; i \neq j \end{gathered} \tag{5.18}$$

- *Many To One Mapping*: Avoids mapping multiple vertices in one graph to the same vertex in another graph:

$$\begin{gathered} \textit{Let}\; u, u\prime \in Dg\,(V) \;\wedge\; v, v\prime \in Dg\prime\,(V) : \\ \nexists\; u \neq u\prime \;\wedge\; v = v\prime \;\forall\; Mp_i = \langle u, v\rangle, Mp_j = \langle u\prime, v\prime\rangle \in Sl\,(Mp) \;\mid \\ i, j = \{1..\,|Sl\,(Mp)|\} \;\wedge\; i \neq j \end{gathered} \tag{5.19}$$

- *Mandatory Mapping*: Defines that all vertices in a graph must be mapped to vertices in another graph:

$$\forall\; u \in Dg\,(V) \;\exists\; Mp_i = \langle u, v\rangle \;\mid i = \{1..\,|Sl\,(Mp)|\}, v \in Dg\prime\,(V) \tag{5.20}$$

- *Include Mapping*: Defines that a vertex in a graph must be mapped to a specific vertex in another graph:

$$\exists\; Mp = \langle u, v\rangle \in Sl\,(Mp) \;\mid u \in Dg\,(V) \;\wedge\; v \in Dg\prime\,(V) \tag{5.21}$$

- *Exclude Mapping*: Prevents the mapping of a vertex in a graph to another vertex in different graphs:

$$\nexists\, Mp = \langle u, v\rangle \in Sl\,(Mp) \mid u \in Dg\,(V) \;\wedge\; v \in Dg\prime\,(V) \qquad (5.22)$$

- *Imply Mapping*: If a vertex u in graph Dg is mapped to a vertex v in another graph $Dg\prime$, than another vertex $u\prime$ in graph Dg must be mapped to vertex $v\prime$ in graph $Dg\prime$:

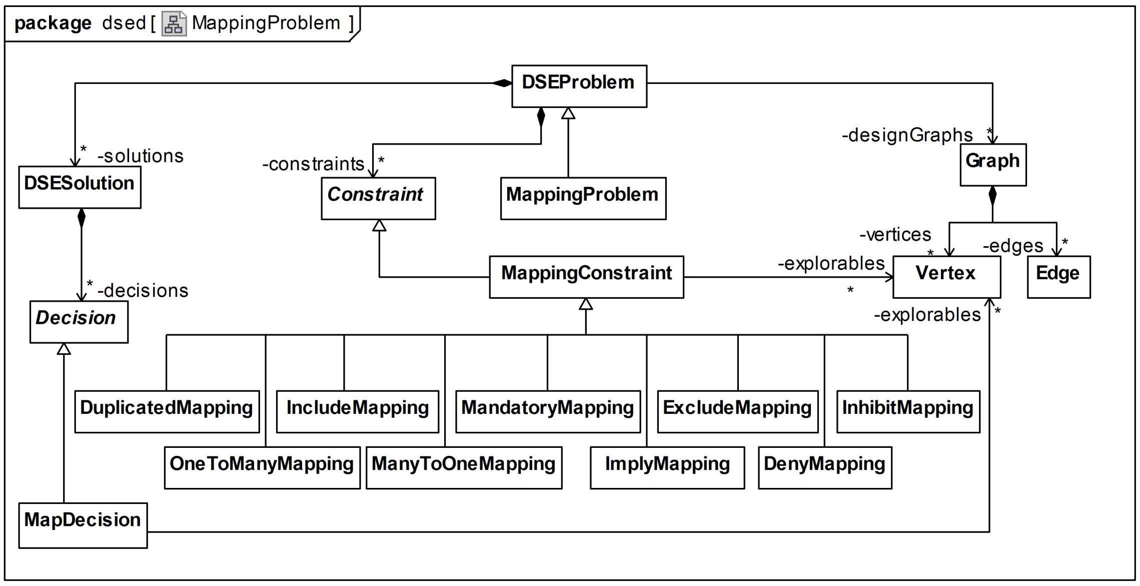
$$\textit{if}\,\exists\, Mp = \langle u, v\rangle \mid u \in Dg\,(V)\,, v \in Dg\prime\,(V) \;\Rightarrow$$
$$\exists\, Mp\prime = \langle u\prime, v\prime\rangle \mid u\prime \in Dg\,(V)\,, v\prime \in Dg\prime\,(V) \;\wedge\; Mp = \langle u, v\rangle \neq Mp\prime = \langle u\prime, v\prime\rangle \qquad (5.23)$$

- *Inhibit Mapping*: If a vertex u in graph Dg is mapped to a vertex v in another graph $Dg\prime$, than vertex $u\prime$ in graph Dg must not be mapped to a vertex $v\prime$ in graph $Dg\prime$:

$$\textit{if}\,\exists\, Mp_i = \langle u, v\rangle\, | u \in Dg\,(V)\,, v \in Dg\prime\,(V) \;\Rightarrow\; Mp_j\prime = \langle u\prime, v\prime\rangle \notin Sl\,(Mp) \qquad (5.24)$$

Definition 5.4 is represented in the DSED metamodel by the elements illustrated in Figure 5.9.

Figure 5.9: Design Space Exploration Domain Metamodel - Mapping Problem.



A `MappingProblem` ($P_{mapping}$) element represents a DSE Mapping Problem, which extends `DSEProblem`. It contains a subset ($D\prime$) of design graphs `Graph` (Dg), which contain `Vertex` (V) and `Edges` (E). The mapping between vertices of these graphs is defined in `MapDecisions` (M), which are stored in `DSESolution` (Sl). This problem is subject to `MappingConstraints` (Cs) of type `DuplicatedMapping`, `OneToManyMapping`, `ManyToOneMapping`, `MandatoryMapping`, `Include-Mapping`, `ExpludeMapping`, `ImpliesMapping`, and `InhibitMapping`, which represent Constraints 5.17 - 5.24, respectively.

5.4.5 Scheduling Problem

Scheduling is a problem where start times must be assigned to a task set, which may execute under time constraints, such as deadline and period. Messages between elements are also subject to scheduling and play an important role in real-time and networked embedded systems, such as in the automotive industry.

In the DSE Scheduling Problem P_{sched} two graphs represent the system and form the subset $D\prime$ of design graphs, namely task graph Dg_t and processor graph Dg_p. A task graph is a directed graph, according to Definition 4.1, whose vertices represent tasks and edges represent a dependency between tasks. A task has a set of attributes τ which characterize it in terms of computational effort and time, such as computational time required to execute and interval between execution. Tasks may share processors, which sequentially execute a task set. Although tasks may also share resources, such as data-structures or input/output devices, the current DSE Scheduling problem model is limited to the scheduling of tasks in shared processors. Multiple task graphs can be used to represent independent task sets. A solution Sl for the scheduling problem is a set of tuples St, whose attributes are assigned to tasks in order to define the timing attributes that meet timing and resource constraints. Definition 5.5 presents the scheduling problem based on (BUTTAZZO, 2011).

Definition 5.5 (DSE Scheduling Problem)**:**
$P_{sched} = \langle P \langle D\prime, Sl\left(St\right), Rm, Cs\rangle\rangle$ *is a DSE Scheduling Problem, which specializes the Definition 5.1 as follows:*

P is the general DSE problem as specified in Definition 5.1.

$D\prime$ is a set of directed graphs, whose graph Dg_t represents task graph and Dg_p represents a processor graph, such that $Dg_t \cup Dg_p = D\prime$ and $Dg_i \cap Dg_j = \emptyset\ \forall i \neq j$.

Dg_t is a directed graph as defined in Definition 4.1, whose vertices represent tasks, such that a tuple $\tau_i \langle T_i, r_i, C_i, d_i, D_i\rangle \in Dg_t\left(V\right)$ is a task and edge $e_{ij} \in Dg_t\left(E\right)$ are dependencies between tasks τ_i and τ_j.

T_i the period of task τ_i is the interval between two consecutive executions of a periodic task τ_i or the minimum interval in the case of aperiodic and sporadic tasks.

r_i the release time is the time at which task τ_i becomes ready for execution.

C_i the computation time is the time required to execute task τ_i without interruption.

d_i the absolute deadline is the time before which task τ_i should be completed.

D_i the relative deadline is the difference between absolute deadline and release time.

Dg_p is a direct graph as defined in Definition 4.1, whose vertex set $Dg_p\left(V\right)$ represent processors where tasks are scheduled.

Dg_r is a direct graph as defined in Definition 4.1, whose vertex set $Dg_r(V)$ represent resources shared by tasks.

St $St = \langle T_i, r_i, d_i, D_i, s_i, f_i \rangle$ store tasks properties which configures a task τ_{ij} for a specific solution Sl, such $0 \leq i < |Dg(V)|$ and j is the n^{th} execution of τ_i.

s_i the start time is the time at which a task τ_i starts executing.

f_i the finish time is the time at which a task τ_i finishes its execution.

$Sl(St)$ is a place holder for a scheduling solution, which contains the set of values to be assigned to tasks in order to fulfill the set of constraints Cs.

Although the scheduling constraints guide solutions for scheduling problems, they can also be applied when there is no requirement for scheduling optimization. The scheduling constraints are also required when engineers intent to solve other DSE Problems, whose solution are tasks in a system that must share resources. For example, when solving a Construction problem, a solution must contain enough computational resource (adequate number and type of processors) to execute the system functions and fulfill their timing requirements. Such interleaving between DSE problems is one of the challenges that the methodology proposed in this chapter tries to address. First, multiple problem representation is provided, so that engineers can make use of separation of concerns to define the constraints according to each scenario. Moreover, model elements shared between multiple DSE Problems allow for exchanging information between these problems, so that evaluators and solvers can deal with such interleaving information. The following constraints can be defined by an engineer in order to solve a DSE Scheduling Problem:

- *Precedence*: Determines the precedence between a pair of tasks:

$$\textit{if}\, \tau_i \prec \tau_j \quad \Rightarrow \exists\, St_i, St_j \in Sl \mid f_i < s_j \tag{5.25}$$

- *Assigned Start Time*: Specifies when a task must start executing;

$$St_i \langle T_i, r_i, d_i, D_i, s_i, f_i \rangle \in Sl \mid s_i = s_{assigned} \tag{5.26}$$

- *Assigned Finish Time*: Specifies when a task must finish its execution:

$$St_i \langle T_i, r_i, d_i, D_i, s_i, f_i \rangle \in Sl \mid f_i = f_{assigned} \tag{5.27}$$

- *Deadline*: Assures that a value lower than the deadline is assigned to a finish time:

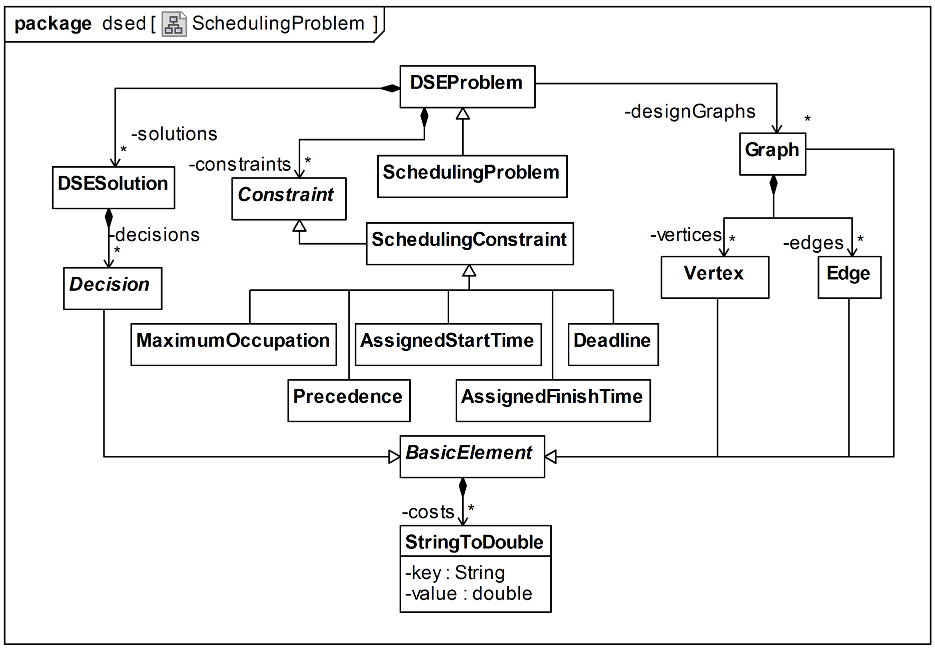
$$\forall\, St_i \in Sl \mid f_i \leq D_i \tag{5.28}$$

- *Maximum Occupation*: Defines an upper limit Occ_{max} for occupation of a resource, so that a schedulability test of periodic task sets can be satisfied:

$$\sum_{i=0}^{|Dg_t(V)|} \left(\frac{C_i}{T_i}\right) \leq Occ_{max} \tag{5.29}$$

The DSE Scheduling Problem presented in Definition 5.5 represents in graphs the set of elements to be scheduled, such as tasks, resources and messages. Multiple graphs can be used, in order to represent resources where the elements must be scheduled, such as processors and buses. The costs annotated in vertices, edges and graph, or in solution and decisions, provide a flexible representation for a wide variation of information required for the scheduling definition, such as deadline, delay, jitter, and others. The proposed DSED metamodel elements capture such concepts and are illustrated in Figure 5.10.

Figure 5.10: Design Space Exploration Domain Metamodel - Scheduling Problem.



The `SchedulingProblem` (P_{sched}) element represents the DSE Scheduling Problem. It contains a set of constraints (Cs), which the problem is subject of. Constraints 5.25 - 5.29 are represented by elements `Precedence`, `AssignedStartTime`, `Assigned-FinishTime`, `Deadline`, and `MaximumOccupation`, respectively. `Scheduling-Problem` has also a subset ($D\prime$) of design graphs, which contain `Graphs` that represent task graph (Dg_t) and processor graph (Dg_p). The vertices represented by the element `Vertex` (V) in these graphs represent the tasks or processors, and `Edges` (E) define dependencies between them. Some elements of this metamodel, such as `Vertex` and `Decision` (St), are specializations of `BasicElement`, so that they have a list of `Value`. Therefore, they can store values for different properties related to the scheduling problem, such as the tuple of task properties τ, which contains deadline (D), release time (r), period (T), and other properties.

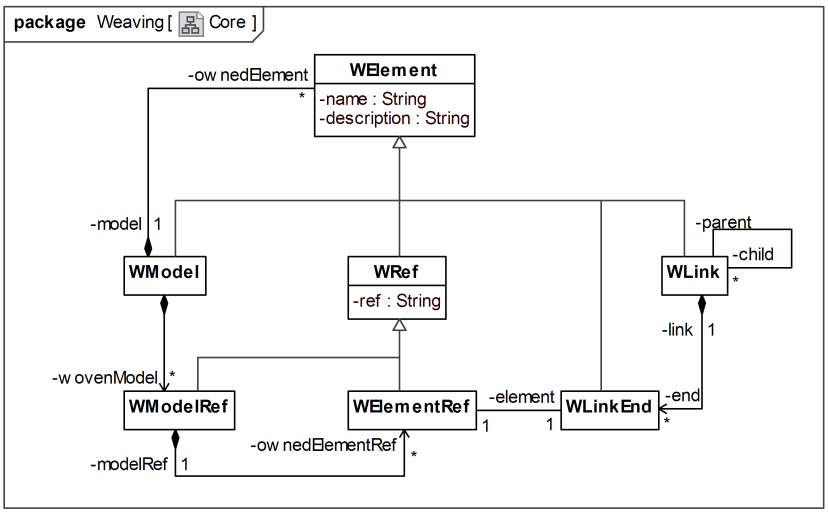
5.5 Design and DSED Model Weaving

In order to improve DSE model specification, the ideas presented in (OLIVEIRA et al., 2010) were extended to DSE-related elements. Model weaving, is used to combine design and DSED model elements without compromising the separation of concerns. In this fashion the variability of design is seen as an orthogonal concern of system design, and as such it must be woven to system design elements representing the final model used as input for DSE.

The separation of concerns is a concept present in different development process models. For example, Y-Chart and PBD propose application and architecture, whereas ROPES and RUP-SE propose structural, communication, deployment, and others concerns. The concrete approach used by engineers to achieve separation of concerns varies in the literature, such as Object-oriented Programming, Aspect-oriented Programming, MDE and other approaches. Although the Object-oriented Programming is considered a huge contribution to software development, it is still requiring complementary approaches to improve the software development (BEZIVIN, 2005). Moreover, the embedded systems domain has additional challenges that the Object-oriented approach only is not enough to overcome. Therefore, many proposals try to improve the development by applying different approaches, such as Aspect-oriented Design (WEHRMEISTER et al., 2008; LINEHAN; CLARKE, 2012) and MDE (TERRIER; GéRARD, 2006; PIEL et al., 2008).

“Metamodeling is a convenient way for isolating concerns of a systems” (DEL FABRO et al., 2005), such that it creates multiple modeling dimensions (KENT, 2002). Although model transformations are the most common way to define relationships between these dimensions (BEZIVIN, 2005), model weaving is another important operation in MDE with similar purpose. Model weaving establishes links between distinct (meta)models, and produces a weaving model that relates with these models, remaining linked to them to be used for different purposes, such as traceability, tool interoperability, model composition, and model alignment (DEL FABRO et al., 2005). Figure 5.11 illustrates the weaving metamodel used to represent the link from multiple design models to DSED models.

Figure 5.11: Weaving Metamodel based on the AWM metamodel.



(DEL FABRO et al., 2005)

The basis of the Weaving Metamodel is the `WElement`, which represents all metamodel elements. It has name and description attributes, which are inherited by all other elements. The element `WModel` represents the weaving model and serves as a container for all other elements. In the proposed methodology the model that weaves the DSED model to multiple design models is a `WModel`. It is composed of `WElement` and `WModelRef`, which is a reference to the models being woven, such as the DSED, UML, and Simulink models. Such references allow keeping track of woven models. `WModelRef` also contains `WElementRef`, which represents elements owned by a model, e.g. classes from a UML model. The link between elements in different models is represented by `WLink`. A bidirectional association to itself, namely `parent` and `child`, allows the hierarchical specification of `WLinks`. It also contains `WLinkEnds`, which is a link extremity. Each `WLinkEnd` is associated to a `WElementRef`, which represents a referenced element of a woven model. Such indirect association between `WLink` and `WElementRef` enables the linkan between arbitrary number of elements. `WModelRef` and `WElementRef` extend the abstract element `WRef`, which contains the field of type String named `ref`. This field contains the identifier of the woven elements and models. Such identifier is the physical link between models and is technology-dependent, so that an identification mechanism is required, e.g. XPointer [4] or XMI-IDs [5]. Such a mechanism is provided by model weaving tools, such as AWM (DEL FABRO et al., 2005), which is the one adopted in the proposed methodology.

In this thesis model weaving is applied to create a link between design and DSED models with the following purpose: i) allow the adoption of multiple design models (or languages) during the DSE process; ii) allow the back annotation of design decisions taken during the DSE process into design models; iii) represent the variability of design models as an orthogonal design concern; and iv) allow direct exploitation of design information during the DSE, when creating heuristics for specific domain or application.

5.6 DSE Rules Definition

In the proposed approach, DSE rules are model transformation rules. Therefore, the name “DSE rules” is used instead of the usual “constraints”, because transformation rules provide constraint expressions to define source/target matching patterns and additionally an action block, so that by using DSE rules one can not only assert properties in the model but also make changes in the target model.

These DSE rules receive an unconstrained design space as input and generate a constrained design space as output, when executed by a transformation engine. Such rules are constraints to guide and prune the available design space, to reduce the exploration time and ensure the feasibility of a candidate solution. In order to make the identification of rules easy, we classify them into three categories:

- *Structural rules*: These rules are applied to avoid illegal designs, which could appear after the combination of different design decisions. Typical rules avoid double assignments of an element, e.g. different processors assigned to the same slot in a given communication structure or the same task assigned to different processors. Other rules may ensure that all tasks must be mapped, or at least one processor must

[4]XPointer is a system for addressing components of XML based media.

[5]XML Metadata Interchange (XMI) is the OMG standard for exchanging model information between modeling tools and repositories. XMI-ID is the unique identifier which should remain the same during all the modeling stages

be allocated. These rules can specify integration issues, such as two components that could not be integrated into the same system because of incompatibility issues.

- *Non-Functional rules*: Even if a design is feasible, it can be invalidated when checked against NFRs, which must be satisfied by the system. In this way, these rules avoid the selection of solutions that violate requirements, such as task deadlines, maximum delays, and maximum energy consumption. One challenge to the DSE process is to deal with system metrics, which cannot be partially evaluated. As such, the DSE process may postpone the design space pruning to a second step, after system evaluation, when it could filter the candidates from the design space. This procedure avoids selecting the candidate again by removing design alternatives that may cause requirements violation.
- *Defined Design Decisions rules*: Designs usually start with pre-defined design decisions and previously developed components, and the selected platform may impose restrictions, which an engineer must respect. Moreover, engineer's experience may influence how the automated DSE process proceeds. Therefore, these rules are specified in cases where one needs to interfere on the DSE process through specific design decisions. Typical rules define specific task mapping, processor allocation, specific task or processor execution frequency, and others.

Considering this classification, an engineer of the proposed DSE methodology is expected to define some rules for each category, which applies to his/her set of DSE problems. The DSED metamodel provides elements to represent a set of commonly used constraints, which can be defined by an engineer, when creating a DSED model. Such elements are abstract representations of the constraints, whose implementation is not bound to any language and solver. However, they possess a strong semantic, which was defined in Section 5.4. The DSED metamodel can be easily extended to include additional constraints, but the actual implementation depends on the solver adopted in the tooling environment, such as a solver based on FORMULA requires constraints written in FORMULA, and the solver presented in Chapter 6 requires the DSE rules implemented in VIATRA II, because it is the transformation engine integrated in the solver.

The direct representation of specific constraints in elements of the DSED is an important mechanism to overcome the inflexibility of current DSE methods. Model transformations access the DSED model to generate solutions according to constraints specified in the DSED model. By identifying a constraint specified in the DSED model, a transformation must execute according to the semantic defined in Section 5.4. The advantage of this approach is that constraints can be quicker defined, by dealing only with domain concepts and any specific constraint language syntax. A library of DSE rules boost the productivity during the DSE process, because well-defined and test rules can be automatically reused inside solvers, by simple testing if there exists a instance of specific constraint in the DSED model.

DSE rules can be implemented in any model transformation language that supports the transformation of ECORE-based models. Such restriction exists only because the metamodels used to support this methodology were defined in ECORE. Transformation from ECORE to other metamodeling language, support to multiple languages, or metamodel agnostic frameworks, e.g. VIATRA II[6] can overcome such restriction. Another way to implement such constraints is the implementation of them in the input model required by

[6]http://www.eclipse.org/viatra2/

the adopted solver. Flexible frameworks provide their own language, such as Mathlab, FORMULA, and SPLOT. Other environments provide building blocks of optimization algorithms implemented in Java or C++, so that engineers can construct their own solvers, which also must include an implementation of constraints that must test which of them are defined in the DSED model. Such an implementation is possible, because, Java and C++ APIs can be generated from the DSED metamodel, by using EMF and EMF4CPP[7] frameworks, respectively. Chapter 7 provides more information about the implementation of constraints in order to support the proposed methodology.

5.7 Design Space Exploration Process

The DSE process is responsible for generating candidate designs based on some exploration/optimization strategy. The content of the DSED model defines the DSE problems to be solved, including the metrics required and evaluation and solver tools to be used. All this information is easily accessed by using the API generated from the DSED metamodel or by model transformations. Even some constraints can be derived from the DSED model, such as pre-defined decisions, variable domains and the configuration semantic for Construction problems. By accessing the DSED model and using adequate transformations the information can be applied to different exploration mechanisms, in order to generate solutions for different combinations of DSE problems. The results are stored again in the DSED models after a population of solutions is generated.

The most general way to implement an exploration mechanism (a solver for DSE problems) for the DSE process is to adopt some optimization tool and transform the DSED model to the input model conforming to the requirements of the adopted tool and write its output into the DSED model. In this method there is no interaction between the optimization tool and DSED during the DSE process, which may lead to limitations, such as problems to find support for constraints specified in the DSED, combination of multiple DSE problems, and exploitation of problem specific features, so that it may also require pre- and post-processing to encode the problem for the solver or remove design candidates. Although this pragmatic approach does not exploit all benefits of the proposed methodology, it allows access to global optimizers in a simple way, so that less expertise is required from users that are integrating the DSED and the optimization tool. They provide both, general design candidate generation functions, given an appropriate encoding of the problem, and search/optimization functions. Besides, optimization tools provide strong support to different solvers, mathematical analysis, and advanced visualization techniques of the results. Examples of such tools are modeFrontier[8] and Guimoo (LIEFOOGHE et al., 2007). Therefore, the DSED model, DSED API and the methodology flow themselves are contributions to DSE process, because they provide a convergence mechanism, which allows standard representation of DSE concepts, easy integration of multiple solvers in the DSE process, and integration of the last in the development process.

Alternatively, the DSED model and API can be interactively used during the DSE process, by using engineers' own implementations of solvers and optimization frameworks. Optimization frameworks help engineers by providing reused components of common optimization algorithms and an API, so that engineers can implement their own heuristics according to their requirements. Opt4J (LUKASIEWYCZ et al., 2011) and Watchmaker [9]

[7]https://code.google.com/p/emf4cpp/

[8]http://www.modefrontier.com

[9]http://watchmaker.uncommons.org/

are examples of this kind of frameworks. By using this method, engineers are requested to implement a design candidate generation function and an optimization mechanism. This approach requires strong knowledge of solvers, optimization algorithms, and the problem to be solved. However, one can explore the benefits of the methodology, as the user has full control of the exploration mechanism and of the interaction with the DSED and Design-DSED Model.

A third method to implement/integrate solvers is to use adequate model transformation engines and transformation rules, as proposed by the author of this theses and colleagues in (OLIVEIRA et al., 2009), and other works such as (JACKSON; SCHULTE, 2008; SAXENA; KARSAI, 2010; SCHATZ; HOLZL; LUNDKVIST, 2010; HEGEDUS et al., 2011). By using this method a metamodel of the DSE problem must be defined. Afterwards, constraints over (meta)models or constraints implemented as transformation rules are used to generate design candidates according to requirements. Going beyond model transformation, some approaches allow for automatic generation of models from the specification of (meta)model, constraints/transformation rules, and partial models. Such an MDE approach provides a flexible mechanism, wherewith a engineer can define under which conditions the DSE problems must be solved, without requiring changes in the DSE tool. Constraints and relations between DSE problems can easily be defined, and additional actions could be specified if some conditions are met. Moreover, the model transformations are artifacts that can be evolved, as problems and solutions change. In this way the DSE rules, Design-DSED, and DSED models are easily integrated into the development process, because there are tools to easily handle these models and transformations - which are models, too. Although the support for automatic generation of models is still incipient, some tools start supporting it, such as the work in (PETTER; BEHRING; MüHLHäUSER, 2009), VIATRA II (HEGEDUS et al., 2011), and FORMULA[10]. By adopting this method, engineers are not requested to implement a design candidate generation function, but they are requested to implement optimization functions. The generation of optimized models are rather limited, because there is no optimization mechanism to guide the search or it is limited to values existing during the generation, as no interaction with evaluators are provided. Furthermore, as discussed in section 4.5, except for the work in (SAXENA; KARSAI, 2010) the MDE methods for DSE are better classified as MDE framework as they not provide any specific DSE concept and abstraction for DSE, and all of them request engineers to completely customize the tooling by defining their own (meta)models and transformations. Therefore, the methodology proposed in this thesis complements the general works proposed in (JACKSON; SCHULTE, 2008; SCHATZ; HOLZL; LUNDKVIST, 2010; HEGEDUS et al., 2011), in the way that such methods can be specialized for the proposed methodology or used to partially implement it. In the context of this thesis, such methods are global solvers, which can be integrated into the proposed methodology to solve any of the DSE problems specified in the DSED metamodel.

The method adopted in this thesis to support the DSE process and implement a solver combines two alternatives. It uses an optimization framework to support the optimization mechanism, which is an implementation of the Crowding Population-based Ant-Colony Optimization Algorithm for Multi-Objective (CPACO-MO) (ANGUS, 2007). This mechanism is responsible for searching in the design space and construct the Pareto-set of design alternatives, and is described in Section 6.4.1. The design candidate generation function adopts model transformation rules to represent the DSE rules and use transfor-

[10]http://research.microsoft.com/en-us/projects/formula/

mation engines to refine a DSED model, which contains DSE problem instances to be solved, into a DSED model also containing solutions for the specified problems. The design candidate generation function is described in Section 6.4.2.

The method implemented to support the DSE process exploits the DSED metamodel, which provides specific concepts and abstractions for DSE, and a library of provided common DSE rules, which reduces the effort to define the transformation required to generate adequate design candidates. The method implemented balances global and specific optimization, by providing an implementation of a global optimizer and an integration with the MDE framework, so that engineers can use model transformations in side a pre-implemented step-wise search to guide the DSE process in a very flexible way. Such implementation also allows the interaction during the DSE process of solver with values generated by evaluators in the DSED model, which are used to guide DSE process and prune the design space. Additional information on the implementation of the adopted method is presented in Chapters 6 and 7.

5.8 Evaluation Process

During the DSE process an evaluation can take place at different phases. In order to generate one design candidate, a partial evaluation at each decision can be performed. Such a kind of evaluation is useful when step-wise search is performed, so that the next decision is supported by the evaluation of available alternatives, if there is enough information for that evaluation, until all decisions are made to compose a solution.

However, there is a point in time where a candidate design cannot be partially evaluated. This is the case, for example, when the execution time of a task in a processor is not known at the time of a decision. Therefore, the decision is made without considering any previous evaluation of alternatives, postponing the evaluation until a complete solution is known, containing all required decisions. After the complete solution is known, it could be fully evaluated and the results noted to build up a population of design candidates.

In order to support both types of evaluation, the DSED metamodel provides elements to represent evaluation metrics for partial evaluation in each `Vertex` and `Edge` of a `Graph`. Fully evaluation or previous knowledge of metrics can be stored in the element `Cost` associated to a `Solution` or a `Decision`. The API generated from the metamodel provides easy access to this information, so that evaluation tools can take advantage of them.

The interface `Evaluator` is provided, so that a proxy class, which must intermediate the solver and evaluation tools, can be implemented to meet the user requirements for evaluation. By using this interface, the evaluation tool can be automatically configured and executed during the DSE process.

Assessing each candidate solution must not require detailed synthesis and cycle-accurate simulation, because evaluation time will be prohibitive if a large set of design candidates must be evaluated. Therefore, the proposed methodology adopts and extends the SPEU tool (OLIVEIRA et al., 2006). SPEU implements a hybrid estimation method, which provides analytical estimates on physical system properties. These properties are directly obtained from UML models, C++ source code and/or compiled binary code, which are transformed into CDFGs.

Before starting an evaluation, the SPEU proxy accesses DSED and UML models in order to generate its internal representation, containing the CDFG extracted from UML models and the architectural decisions produced during the DSE process and stored in the

DSED model. After a fast evaluation, the SPEU proxy stores the estimated properties in the DSED model again, so that the DSE process can continue its execution.

The approach implemented by SPEU allows quick and static evaluation of candidate designs using information of different sources and at different abstraction levels, without depending on costly synthesis and simulation evaluation cycles. Therefore, SPEU is adequate to the purpose of the presented methodology. Further details about the SPEU tool are provided in Section 7.3

5.9 Discussion

Differently from the proposals found in the literature of MDE applied to embedded systems, which are only concerned about the integration of design tools to DSE by applying model-to-model transformations from the output of one tool to other ones (MURILLO; MURA; PREVOSTINI, 2010; KANGAS et al., 2006), the methodology presented in this chapter exploits the MDE approach to improve the flexibility, reuse, and productivity for DSE of embedded systems. Initially, a methodology flow was presented, which integrates the DSE activities in the development process. Afterward the DSED metamodel was defined, in order to represent DSE domain concepts. Such a metamodel allows the explicit specification of variability in embedded system design models. A method for weaving design models with the DSED model is used to create a link between these models, which is used to exploit domain-specific knowledge in the DSE process, improve the rationale of design decisions, and back annotate the DSE results. Moreover, three methods that can be used to implement the DSE process are identified, namely the bridging between the DSED model with optimization tools, the use of optimization frameworks by implementing domain specific heuristics, and the use of transformation engines that execute the DSE rules defined to refine the DSED model. Such methods trade-off the flexibility and control over the process. The method adopted for this thesis in order to support the DSE process combines two alternatives, an optimization framework to implement an optimization mechanism, and the use of model transformation rules and a transformation engine to generate design candidates. The DSE process method also implements a step-wise algorithm, which integrates an MDE framework to execute the DSE rules that guide the DSE process and prune the design space. Finally, the methodology foresees the integration of multiple solver and evaluation tools and provides an evaluation tool adequate for quick DSE at high abstraction level.

Similarly to the proposed methodology, there are also other approaches that use a metamodel to represent the design space or some concepts of the DSE domain. In the work presented in (NEEMA et al., 2003) a metamodel is defined to represent the design space for the construction problem. Besides addressing only one DSE problem, it requires an engineer to model the design space by defining templates that can be filled with components from a library. The approach proposed in (SAXENA; KARSAI, 2011) requires the extension of the metamodels that represent the design with the variability concepts of the DSE problems to be addressed and the late generation of a modeling tool from the extended metamodel, which prohibits the use of the conventional modeling tools by engineers. In the approach presented in (SCHATZ; HOLZL; LUNDKVIST, 2010) the DSE model, design model and the respective metamodels are mingled. The result is a partial model, which contains an incomplete model of the systems in development and the rules that define how such an incomplete model can be filled with new elements. Such a model is used as input for a transformation engine able to generate the missing elements, according to the rules defined. However, if it is considered a metamodel such as UML, or Simulink, it can creates a huge model to be solved, because no abstraction is provided to reduce the search space. Although the approach provides high flexibility, because any metamodel directly represents a design space, this approach does not provide abstraction to improve the specification of variability and constraints, hence engineers must use simple metamodels or create very detailed models to reduce variability and the size of the model to be created. Moreover, there are no computational support for integrating design and DSE models.

In the methodology described in this chapter DSE concepts were gathered, and used to define a metamodel to represent the DSE domain. As such a DSE domain (DSED) metamodel was defined to represent the concepts that are commonly found in any DSE problem. Moreover, it also identifies and provides elements to represent four DSE classes of problems, namely Configuration, Construction, Mapping, and Scheduling, which are able to represent a variety of DSE problem instances. As a result, the DSED metamodel allows for flexible representation of different DSE scenarios, by combining different problems, evaluators and solvers, while managing the complexity by providing abstraction from design and specialized problem definition.

The inflexible bond of the DSE method to a design language is present in many proposals, such as in (KANGAS et al., 2006) and (NEEMA et al., 2003). One reason for this dependency is the lack of an explicit and orthogonal DSE domain model, such as the one proposed in this chapter. As there is no DSE model, the required information must be extracted from the design model or specified together, resulting in a dependency between design and DSE tools. Another reason is the requirement imposed by evaluation tools that may have easy integration with some design tools or whose data model is shared with the DSE method, as presented in (PIMENTEL, 2008) and (LIEVERSE et al., 2001).

The proposed approach shown in this chapter allows the orthogonal specification of the variability in the embedded system design model as another system design view, and use the DSED for this purpose. An aspect weaving method is used to implement the link between design elements and the DSED model, which contains information about the variability of design elements. Such orthogonal specification allows the application of the proposed methodology in different design environments, independently from the design language used, such as UML, SysML, and Simulink. Moreover, it also allows the integration of information extracted from theses languages in a combined DSED scenario.

Similarly to the DSE rules method proposed in this thesis, there are other proposals that apply model-to-model transformations to specify DSE rules or for similar purposes. The Prolog model specification proposed in (SCHATZ; HOLZL; LUNDKVIST, 2010) allows the specification of incomplete transformation rules, where the lack of precision is filled by proposing different models that fulfill the transformation rule specification. This approach can be applied to a variety of problems. However, it has a bad separation of concerns, thus it is difficult to integrate the proposal into a development process. No solution for the application of different solvers and evaluation is proposed, what reduces its applicability. A constraint solver is integrated into the background of a transformation engine in (HEGEDUS et al., 2011), which similarly to (SCHATZ; HOLZL; LUNDKVIST, 2010) also presents a general approach to optimize models generated by transformation engines. It improves the application of transformation rules by extracting dependency information, providing hints to remove dead end states, and prioritizing possibles operations. In (SAXENA; KARSAI, 2010) the Constraint Specification Language (CSL) is proposed on basis of a subset of the OMG's OCL. This language is used to define the DSE rules, but no model-to-model transformation strategy is implemented based on it. Instead, the model defined by CSL is translated into an intermediate language used as front-end to different constraint solvers. The CSL provides some high-level constructs to reduce the verbosity of OCL and to overcome the limitation of OCL to address multi-context constraints. However, such limitations are solved by highly adopted model-to-model transformation languages, such as OMG's QVT, ATL and VIATRA II.

This chapter presented a methodology that adopts a conventional model-to-model transformation language, which presents more benefits, such as easy adoption, strong supporting tools, multi-context constraints, etc, which can also be translated to intermediate constraint solver languages, instead of a proprietary language, such as in (SAXENA; KARSAI, 2010). Furthermore, smart transformations, which exploit the semantics of elements, are used to extract variability from design models and include domain expertise in model transformations, in order to achieve more improvements on the DSE process. These transformations are used to implement not only integration transformations, which generate DSED models from design models and annotate the results back, but they are also DSE rules. The DSE rules specified using a model transformation language provide full flexibility in the DSE process. By using this approach any rule can be implemented to satisfy the engineering needs. Optimized transformation engines such the one used in (HEGEDUS et al., 2011) and FORMULA, can be used to complement the methodology proposed here. Moreover, DSE rules are independent from the design language used, because they refer to DSED elements in order to prune the design space and guide the DSE process, by exploiting the domain specific knowledge added to the DSED models.

Therefore, the contribution described in this chapter improves the automation reuse and abstraction. It also manages the complexity by allowing the engineer to breakdown the DSE problem into smaller pieces and to combine the results in a next step to further exploration. Such improvements can impact the productivity of engineers, that can concentrate on application design and trade-off analysis of results, instead of wasting precious time on integration of complex tool chains and translation of requirements into proprietary constraint languages at every different DSE scenario.

6 IMPROVING MAPPING IN PBD METHODOLOGIES

This chapter describes a method to improve the mapping between layers in a PBD methodology. It starts revisiting the challenges faced by DSE methodologies in the current PBD approach. Following, this chapter defines the Categorical Graph Product (CGP) based on the work in (WEICHSEL, 1962) and briefly discusses the properties of CGP that are relevant for the design space abstraction proposed. Conforming to definitions in Chapter 5, the DSE Mapping Problem is redefined to accommodate the design space abstraction, considering the representation as CGP of design graphs. The method used to generate candidate designs is also presented. Finishing this chapter the proposed method is compared to other methods found in the literature.

6.1 Motivation

Although the PBD approach is very valuable for the design of embedded systems, developing applications for the existing complex platforms is a hard task. Developing a new platform from the scratch is a big bet for companies too (GOERING, 2002). Furthermore, the mapping between platform layers requires advanced methods. On the one hand the evaluation of many design alternatives at low-level is prohibitive due to the long evaluation cycles and short time-to-market, on the other hand getting benefits from the optimization potential at higher abstraction layers is difficult (SANGIOVANNI-VINCENTELLI, 2007). The following issues arises on the mapping between platform layers:

- Mapping between layers requires multiple design decisions, involving different design activities. The order in which these decisions are taken matters, and different orders result in different system metrics.

- Multiple design decisions, functions and components lead to a large number of design combinations. However, only a small number of feasible alternative designs can be found due to NFRs.

- The state-of-the-art methods provide solutions for specific design activities. These solutions do not provide extension mechanisms and have limited facilities to define and reuse constraints. Due to such low flexibility, it is difficult to apply these methods to different DSE scenarios.

- Increasing algorithm performance can be done by exploiting the problem structure, thus leading to a problem-specific algorithm. Such a specialization also worsens the flexibility of the DSE methodologies.

More than efficient optimization algorithms, the emerging complex design space during the mapping between platform layers requires efficient and flexible ways to specify DSE rules in order to guide the exploration process. Moreover, the lack of flexibility and the interdependence between design decisions require adequate representations of the design space. Finally, there is also a need for a method that exploits the trade-off between specialized heuristics and global optimizations, in order to reach an easier and flexible DSE tool.

The method proposed in this chapter improves the methodology presented in the Chapter 5 considering only the DSE Mapping problem. The mapping problem is the central issue in System-Level design following the Y-Chart and more specifically in PBD approaches. Different design activities can be represented as mappings between graphs. Therefore, the method presented in this chapter represents the design space as a categorical product of graphs (CGP), which maps automatically multiple graphs and expose element dependencies. The representation of the design space by using CGP also raises the abstraction and shows multiple DSE problems as a single problem instance - DSE Mapping Problem. This method exploits local constraints, to guide the search in the design space and relies on the fact that optimizing a path in the CGP is the same as optimizing a path in each graph individually. Moreover, the representation of the design space as a combined graph, which a design activity is mapped to the general problem of finding a sub-graph, detaches the optimization algorithm from the problem to be solved.

6.2 Categorical Graph Product

Initially, lets us define the CGP (WEICHSEL, 1962) and identify the properties presented by such a product that make it adequate to represent the design space as described in the next section.

Definition 6.1 (Categorical Graph Product)**:**
Let $G_1 = \langle V_1, E_1, \delta_{s1}, \delta_{t1} \rangle$ *and* $G_2 = \langle V_2, E_2, \delta_{s2}, \delta_{t2} \rangle$ *be two graphs as presented in Definition 4.1.* $G_1 \otimes G_2$ *is a categorical graph product, which is defined as follows:*

$G_1 \otimes G_2 = \langle V_1 \times V_2, E_1 \times E_2, \delta_{s1} \times \delta_{s2}, \delta_{t1} \times \delta_{t2} \rangle$*, which represents the graph product between* G_1 *and* G_2*, where* $\{\delta_{s1} \times \delta_{s2}\}$ *and* $\{\delta_{t1} \times \delta_{t2}\}$ *are unambiguously induced by the dot product between vertices and edges, considering that any two vertices* (u_1, u_2) *and* $(v_1, v_2) \in G_1 \otimes G_2$ *are adjacent, if and only if* u_1 *and* $v_1 \in G_1$ *are adjacent and,* u_2 *and* $v_2 \in G_2$ *are adjacent.*

Two projection functions $\pi_1 = \langle \pi V_1, \pi E_1 \rangle : G_1 \otimes G_2 \to G_1$ *and* $\pi_2 = \langle \pi V_2, \pi E_2 \rangle : G_1 \otimes G_2 \to G_2$ *are defined and return the graphs* G_1 *and* G_2*, respectively.*

Because graphs are general representations of data, they are highly used in DSE and other domains. Graph products are also implicitly or explicitly adopted by DSE methodologies. The implicit use of a graph product is observed as a side effect of the combinatorial nature of DSE problems, because system elements are represented as lists or graphs and the Cartesian graph product is implicitly considered, when combining alternatives. For example, some methods adopt heuristics based on genetic algorithm operators, such as mutation and crossover, which randomly combine elements to generate alternative designs (DICK; JHA, 1998) (BLICKLE; TEICH; THIELE, 1998), and other methods apply OBDD, which uses a tree-based representation of the design space to enumerate all alternative designs (NEEMA et al., 2003).

Both, implicit and explicit adoptions of graph product lead to an explosion of states. In order to deal with such challenge, physical or logical implementations of the graph product and different alternative generation strategies are proposed, and they were already identified in Section 3.8.

In the methodology proposed in this chapter, if the CGP is physically implemented, then all vertices of the resulting graph product are represented, so that all vertices are traversed at least once. In such a case the explosion of states can not be precluded, even if the exploration mechanism does not traverse the entire graph in order to generate one alternative design, as done by exhaustive methods. A logical implementation of the CGP provides to the exploration mechanism projection functions π in order to retrieve in the CGP a vertex and its neighbors, and use the source and target functions δ from the original graphs to iterate on the product. Similarly to methods that implicitly adopt graph product and use heuristics to generate alternative designs, by using a logical implementation the exploration mechanism does not enumerate all alternatives, and the graph produced will not be completely traversed, thus only part of its vertices will be produced. The trade-off between logical and physical implementations, as well as the details of the exploration mechanism are discussed in Section 6.4.

The explicit adoption of CGP, presented in this chapter, aims to automatically represent the mapping between graphs and reduces the design space by using edges representing alternatives, which can be pruned to reflect design constraints on possible combinations of alternatives (OLIVEIRA et al., 2009). The CGP allows different views of the same information while preserving the original semantics, such that the resulting graph can be used to perform different design activities based on the same design information - different algorithms on the same data. Moreover, a CGP merges semantically different information, by combining different graphs in the form of a product. As such, DSE can be performed using different design information, but relying on the same structure and algorithms too. Finally, by using the CGP an initial mapping between graphs can be automatically computed, and the constraints for the induction of the vertices and edges expose the dependencies between vertices in each graph together in the CGP. As result, an iteration on the CGP corresponds to iterations through multiple graphs simultaneously.

Therefore, CGP is appropriate for representing simultaneous and interdependent design options, which appear during DSE, so that such a representation can be automatically derived from a system specification and, at the same time, it is flexible to be employed on different DSE scenarios.

6.3 Design Space Representation

Similarly to most DSE approaches we define the design space as a mapping of graphs. However, differently from the usual approach, such as the work presented in (BLICKLE; TEICH; THIELE, 1998), which adopts a manual mapping between semantically defined graphs, our approach uses the CGP in order to automatically generate the mapping between graphs. From the design space generation point of view these graphs are free of any specific design semantics, which is adequate for an abstraction. In Definition 6.2 the design space representation based on the CGP is formalized.

Definition 6.2 (Design Space Graph)**:**
Let D be the set of design graphs, where $Dg_i = \langle V_i, E_i, \delta_{si}, \delta_{ti} \rangle \in D, i = \{1..n\}$ and $n = |S|$.

The design space is the graph $Ds = \langle V_{Ds}, E_{Ds}, \delta_{sDs}, \delta_{tDs}\rangle$, resulting from the categorical graph product of the sequence of terms, which are all graphs in D. In this fashion:

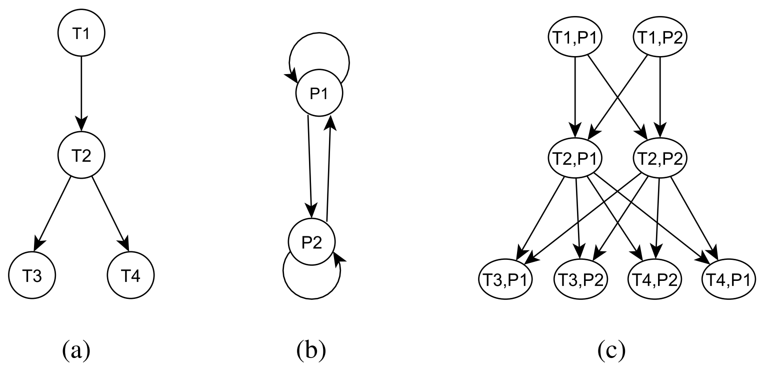
$$\begin{aligned} Ds = \langle V_{Ds}, E_{Ds}, \delta_{sDs}, \delta_{tDs}\rangle = Dg_i \otimes Dg_{i+1} \cdots \otimes \cdots Dg_n = \\ \langle V_i \times V_{i+1} \cdots \times \cdots V_n, E_i \times E_{i+1} \cdots \times \cdots E_n, \\ \delta_{is} \times \delta_{i+1s} \cdots \times \cdots \delta_{ns}, \delta_{it} \times \delta_{i+1t} \cdots \times \cdots \delta_{nt}\rangle \end{aligned} \tag{6.1}$$

This represents the graph product between Dg_i, Dg_{i+1}, $\cdots$, Dg_n, where $\{\delta_{is} \times \delta_{i+1s} \cdots \times \cdots \delta_{ns}\}$ and $\{\delta_{it} \times \delta_{i+1t} \cdots \times \cdots \delta_{nt}\}$ are unambiguously induced by the dot product between vertices and edges, considering that any two vertices $(u_i, u_{i+1}, \cdots, u_n)$ and $(v_i, v_{i+1}, \cdots, v_n)$ are adjacent in Ds, if and only if u_i is adjacent with v_i in Dg_i, u_{i+1} is adjacent with v_{i+1} in Dg_{i+1} and u_n is adjacent with v_n in $Dg_n, i = \{1..n-1\}$, where n is the number of graphs in D.

Definition 6.2 considers that any graph can be used to produce the CGP, in order to represent the design space. Because Definition 6.2 is aimed at the abstraction of the underlined elements to be mapped, there is no reasoning about graph semantics. Such graphs are extracted form design models and represented in the DSED model presented in Chapter 5. In general cases these graphs are, for example, *task graphs*, *architectural graphs*, *communication structure graphs* and others that commonly are mapped during development. Such graphs can usually be combined in any way, except for design constraints. The graph semantic is taken into account during the evaluation of DSE rules, which are defined by the engineer and are responsible for guiding the exploration mechanism and for removing edges that lead to inappropriate combination.

In order to illustrate the proposed design space abstraction, let's consider the design graphs Dg_t and Dg_p. Graph Dg_t is illustrated in the Figure 6.2(a) and represents a task graph where the vertices are tasks and edges specify the data dependencies between them. Graph Dg_p shown in the Figure 6.2(b) represents the processing units and the allowed communications between them. Graph $Dg_t \otimes Dg_p$ in the Figure 6.2(c) is the CGP between graphs Dg_t and Dg_p, representing a design space for the task mapping design activity.

Figure 6.1: Design Space Representation by using Categorical Graph Product between two graphs: (a) Task Graph Dg_t; (b) Processor Graph Dg_p; (c) Design Space $Dg_t \otimes Dg_p$.



One vertex in the design space is a design decision, which represents the mapping between vertices from the graphs that are used to compute the design space and is defined as follows:

Definition 6.3 (Design Decision)**:**
Let Ds be a graph representing the design space as in Definition 6.2.

The vertex $v_p \in V(Ds)$ is a tuple $v_p = \langle v_{ij}, v_{i+1k}, \cdots, v_{nl} \rangle \mid i = \{1..n\}, n = |D|, j, k, l = \{1..m\}, m = |V(Dg_i)|$, representing a design decision that maps the vertices v_{ij}, $v_{i+1k} \cdots$, v_{nl} to each other.

Considering Definition 4.1, the design space is the graph $Ds = \langle V, E, \delta_0, \delta_1 \rangle$, resulting from the CGP of all graphs in D. From now on Definition 4.1 is used to refer to the design space without distinguishing the vertices from the design graphs that compound the design decision. Such a reference emphasizes the graph representation and operation over graphs. Yet Definition 6.2 is used to emphasize the composition of graphs in the product and the components of the design decision tuple.

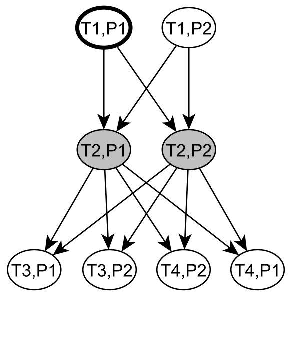
In order to construct a solution to the DSE Mapping Problem, the exploration algorithm must iterate on the vertices of the design space. Such an iteration selects vertices in the design space by following the alternatives provided by the edges in the design space. Alternative decisions represented by edges are defined as follows:

Definition 6.4 (Alternative Decision)**:**
Let Ds be the design space graph as in Definition 6.2, whose design decisions are defined in Definition 6.3.

The edge $e_{uv} \in E(Ds)$ identifies one alternative decision, so that at the specific vertex $u \in Ds$, the adjacent vertex $v \in Ds$ is one design decision available from u. The set of edges $E_u \subseteq Ds(E)$, generated by the source function δ_0 and target function δ_1, represents all alternative decisions from the vertex u.

Figure 6.2 illustrates Definitions 6.3 and 6.4. The resulting design space $Dg_t \otimes Dg_p$ from the previous example in the Figure 6.1 is replicated again in Figure 6.2. The vertices in $Dg_t \otimes Dg_p$ represent design decisions, and the edges identify available alternative design decisions from a specific vertex. Let's assume that in a previous iteration step the vertex $\langle T_1, P_1 \rangle$, highlighted with the bold line, was selected from the design space. This vertex specifies that task T_1 should be mapped onto processor P_1. The outgoing and incoming edges of vertex $\langle T_1, P_1 \rangle$ identify the alternative decisions, such that the vertices highlighted with shadow ellipses $\langle T_2, P_1 \rangle, \langle T_2, P_2 \rangle$ are the ones available for selection staying on the vertex $\langle T_1, P_1 \rangle$. After the selection, the iteration in the graph $Dg_t \otimes Dg_p$ continues until a candidate design is generated.

Figure 6.2: Example of design decisions and alternative decisions representation.



At the end of the iteration the exploration mechanism must return a candidate design, which represents the design decisions selected from the design space. The candidate design is a sub-graph of the design space graph presented in Definition 6.5. As such, the

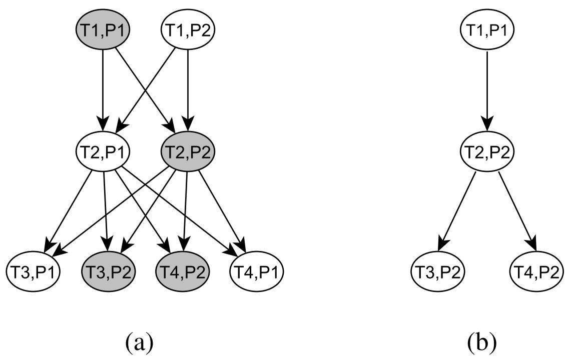
DSE problem consists in searching for sub-graphs, which represent design candidates. Such a problem is formulated in the same way, independently of the design activity performed.

Definition 6.5 (Candidate Design)**:**
Let Ds be the design space graph as in Definition 6.2.

A Candidate design is the sub-graph $C \mid C\,(V) \subseteq Ds\,(V)$ and $C\,(E) \subseteq Ds\,(E)$, where the vertex set $C\,(V)$ represents the design decisions selected manually or automatically from the design space.

By using the CGP representation for the design space, the DSE problem consists in searching for sub-graphs, which represent candidate designs, independently of the design activities to be performed. Figure 6.3 illustrates a sub-graph selected from the design space presented in the previous example. The selected vertices are identified with shadowed ellipses in Figure 6.4(a), and Figure 6.4(b) illustrates the resulting sub-graph, which is composed by vertices $\langle T1, P1\rangle$, $\langle T2, P2\rangle$, $\langle T3, P2\rangle$, and $\langle T4, P2\rangle$. The procedure to select vertices to produce a sub-graph requires an exploration mechanism, in order to optimize the generation of candidate designs. The exploration mechanism and the generation of candidate designs are discussed in Section 6.4.

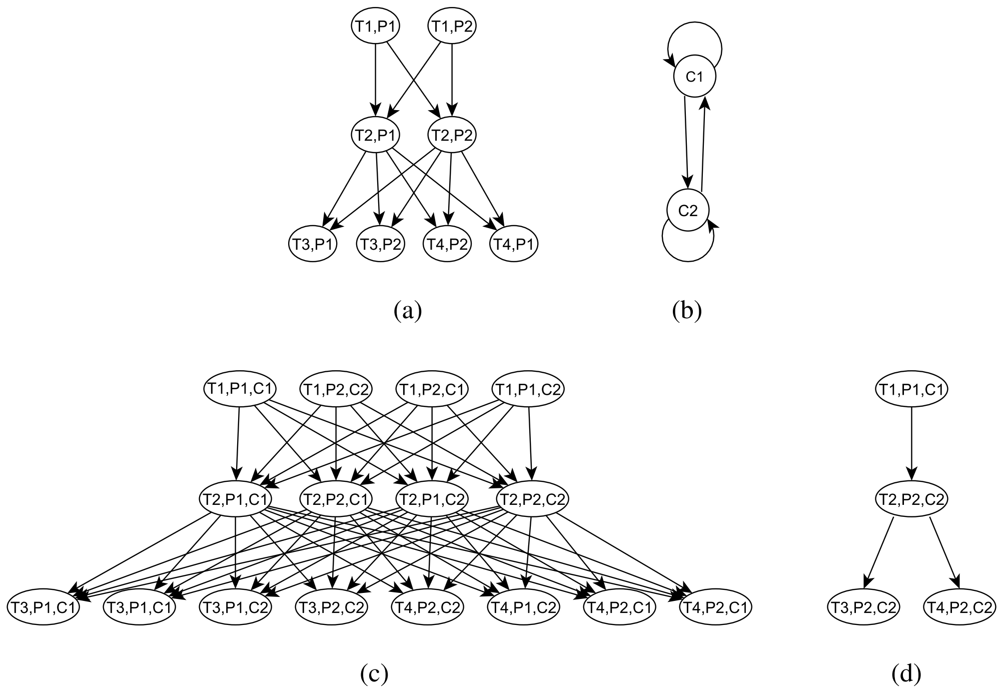
Figure 6.3: Example of a sub-graph representing a candidate design: (a) Design Space $Dg_t \otimes Dg_p$; (b) Candidate solution - sub-graph $Dg_t \otimes Dg_p$.



The application of the CGP between multiple graphs allows to produce a single design space graph for multiple design activities. As an example, consider again the same task graph Dg_t and processor graph Dg_p from the example illustrated in the Figure 6.1 before. The design space resulted from the CGP $Dg_t \otimes Dg_p$ is reproduced again in Figure 6.5(a). Now, consider a communication graph Dg_c, as illustrated in Figure 6.5(b), representing a communication structure, such as a hierarchical bus with two segments used to integrate the selected processors. The CGP between the design space graph $Dg_t \otimes Dg_p$ and graph Dg_c results in the graph $Dg_t \otimes Dg_p \otimes Dg_c$ that represents the mapping between tasks of graph Dg_t and selected processors of graph Dg_p and simultaneously the possible allocation of the processors selected from graph Dg_p to the communication structure represented by graph Dg_c. Such a procedure can continue for multiple products, and the available decisions depend on which graphs are produced from the design and on the DSE rules that guide the exploration and prune the design space.

Considering Definition 6.1 to 6.5 a new DSE Mapping Problem is formulated, in order to accommodate the CGP representation of the design space, shown in Definition 6.6 as follows:

Figure 6.4: Design Space Representation by using Categorical Graph Product between multiple graphs: (c) Design Space $Dg_t \otimes Dg_p$; (b) Communication Graph Dg_c; (c) Design Space $Dg_t \otimes Dg_p \otimes Dg_c$; (d) Candidate solution, sub-graph $Dg_t \otimes Dg_p \otimes Dg_c$.



Definition 6.6 (DSE CGP Mapping Problem)**:**
$P_{CGP} = \langle P \langle D', Sl(Mp), Rm, Cs \rangle, Ds \rangle$ is a DSE CGP Mapping Problem, which extends the Definition 5.4 as follows:

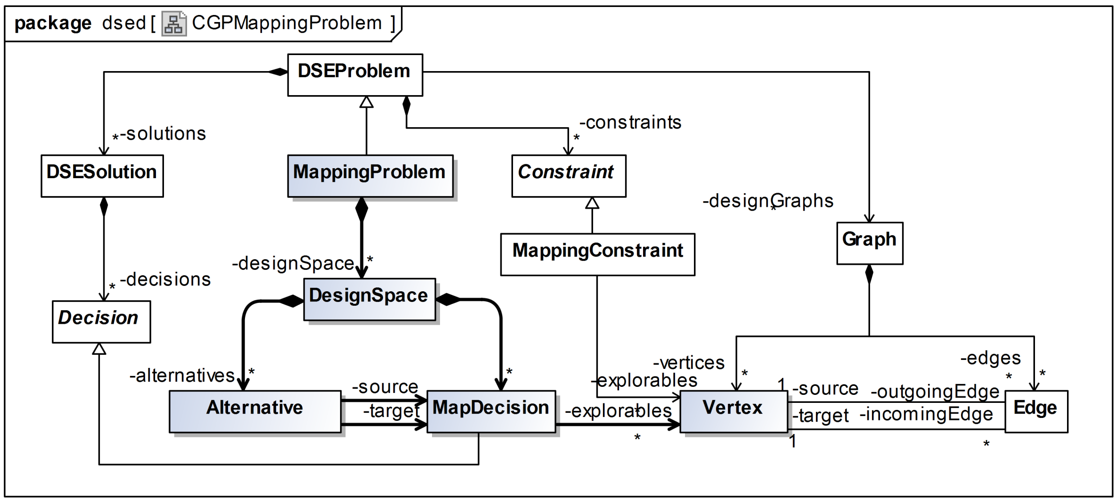
P	is the general DSE problem as specified in Definition 5.1.
Mp	is a map decision as defined in Definition 6.3.
$Sl(Mp)$	is the set of map decisions that solve the problem P_{CGP} and is subject to the same constraint set Cs of the DSE Mapping Problem from Definition5.4.
Ds	is the design space graph as defined in Definition 6.2, resulting from the CGP of graphs in D';

The model presented in Definition 6.6 accommodates the CGP in the DSE Mapping problem from Definition 5.4. It allows the mapping between multiple graphs and represents the mapping as a CGP. Such a representation provides a mingled view of the graphs, exposing the dependence between vertices of all graphs together, so that it can be explored by heuristic optimization methods. To improve the application of the method to different DSE scenarios, it does not assume any graph semantics, hence providing higher abstraction level and flexibility. Moreover, the CGP adopted in this model allows for automatic generation of an initial mapping between graphs, which reduces the design space and allows further refinement by applying local constraints during the DSE process.

It is not made any assumption on the implementation of the CGP that represents the design space, so that different solvers can choose between to exploit the benefits of static and physical implementation of the CGP, which generates the complete design space before starts searching, or the benefits of dynamic and logical implementation, which generates part of the design space as the search iterates on the design space. Finally the proposed model balances the specification of a generic problem (i.e. finding an optimized sub-graph) to be solved by global optimizers with a structure that allows the implementation of the best heuristic to solve a specific problem, by exploiting the values that can be annotated in the model during the search.

Based on Definition 6.6 the DSED metamodel was updated to accommodate the CGP representation of the design space for the DSE Mapping problem. Figure 6.5 illustrates the new DSED elements.

Figure 6.5: Extended DSED Metamodel containing the CGP representation of the design space for the Mapping Problem.



In Figure 6.5 elements added to the DSE Mapping Problem introduced in Figure 5.9 are identified as shadowed classes and bold lines. The class `DesignSpace` represents the design space following Definition 6.2. An `AlternativeDecision` represents an edge in the design space graph, as defined in Definition 6.4. `AlternativeDecision` contains references to `DesignDecisions`, representing the source and target vertices, by following Definition 6.3. Notice that the `Mapping` may contain many `Design-Spaces`. Such a multiplicity allows for the physical implementation of the complete design space or for multiple partial representations, which are especially useful when the design space is logically implemented. The discussion on physical and logical implementations of the design space graph is presented in the Section 6.4.2 and 6.5.

6.4 Solving the DSE CGP Mapping Problem

Since we represent the design space as a graph product and the optimization problem consists in finding a (quasi)-optimal sub-graph, the exploration algorithm is not aware of specific DSE information and of the semantics for vertices and edges in the design space and its sub-graphs. This means that the exploration algorithm is detached from the design space and from the specific design exploration problem and thus does not require a specific design optimization approach. This way, we could use any other multi-objective heuristic. However, under certain conditions heuristics based on Ant Colony algorithms have been shown to outperform others (DORIGO; MANIEZZO; COLORNI, 1996).

In order to solve the proposed DSE CGP Mapping Problem, an algorithm based on the Ant Colony Optimization (ACO) algorithm was adopted. ACO was first proposed in (DORIGO; MANIEZZO; COLORNI, 1996). It is based on the foraging behavior of ants, which let on the way a substance called pheromone, as they search for food. The better the quality of food source, the higher is the amount of pheromone deposited. A higher concentration of pheromone attracts more ants through the path, emerging a shorter path between the food source and the nest. Since the ACO proposal several works extended the original algorithm by augmenting it with new features and by applying it to different problems. The algorithm adopted in this work is the Crowding Population-based Ant Colony Optimization for Multi-Objective (CPACO-MO) (ANGUS, 2007), which combines features from ACO and Evolutionary Algorithms, such as Non-dominated Sorting Genetic Algorithm-II (NSGA-II) (DEB et al., 2000). CPACO-MO presents some features that show to be interesting for the proposed DSE Mapping Problem using the CGP representation, such as:

- An ant can be assigned a starting position, which is a vertex in the design space;
- An ant searches for a minimum/maximum cost to optimize a solution;
- Once a candidate solution is created, and after completing the update of pheromone values in the path selected in the iteration, an ant dies, freeing all allocated resources;
- Ants build up solutions by using a stepwise approach, which selects one vertex (solution component) each time according to a combination of pheromone and heuristic values associated with every vertex in the in the design space. The choice of which vertex is chosen is usually a probabilistic one (transition probability);
- The transition probability used to iterate on the design space in order to construct solutions provides a mechanism to balance between exploration of new solutions and exploration of similar solutions already found;
- The step-wise approach exploits the CGP representation of the design space, allowing the application of DSE rules at each step, guiding the iteration by removing edges that are not allowed to be followed. Such a process has advantages when compared with the random permutation present in algorithms based on Genetic Algorithms, which can lead to infeasible solutions and requires specialized repair mechanisms;
- Use of a population of multiple individual agents (ants) to construct candidate solutions sequentially or in parallel;
- Flexible representation and calculation of multiple objectives.

6.4.1 Exploration Mechanism

Algorithm 6.1 outlines the procedure performed by CPACO-MO. The procedure starts by initializing a population S of pre-defined size S_{size} with randomly generated candidate solutions. Each candidate solution s is evaluated according to the function $\gamma : Sn \rightarrow C$ presented in Definition 5.1. These solutions are inserted into the population and to each solution is assigned an integer rank according to the *Fast Non-dominated Sort* procedure (DEB et al., 2000). Then all solutions in the population are used to update the pheromone map by using a $+\Delta\phi$ according to Equation 6.2.

$$\Delta\phi_{ij}^{s} = \frac{1}{s_{rank}} \tag{6.2}$$

Where:

$\Delta\phi_{ij}^{s}$	is the value to adjust the pheromone ϕ_{ij} in the edge $\langle i, j\rangle$ by the ant s.
s_{rank}	is an integer rank assigned by the *Fast Non-dominated Sort* procedure.

Algorithm 6.1 Crowding Population-Based Ant Colony Optimization: CPACO.

```
1: Uniformly initialize the pheromone map values to $\phi_{init}$
2: for $j = 1$ to $h_{size}$ do
3:     $s \leftarrow$ GENERATE RANDOM SOLUTION
4:     Evaluate Solution $(s)$
5:     Insert Solution s in the history $h$
6:     FAST NON-DOMINATED SORT $(h)$
7:     Update the pheromone map $(+\Delta\phi)$ according to (6.2)
8: end for
9: while stopping criterion not met do
10:    for $j = 1$ to $m$ do
11:        SCALING VALUE ASSIGNMENT
12:        $s \leftarrow$ GENERATE SOLUTION
13:        Evaluate Solution $(s)$
14:    end for
15:    CROWDING REPLACEMENT
16:    FAST NON-DOMINATED SORT $(h)$
17:    Update the pheromone map $(+\Delta\phi)$ according to (6.2)
18: end while
```

Line 9 is the main loop of the algorithm, which will be executed until a stop condition is met, such as a maximum computation time or a minimum difference in the objectives between iterations. In the internal loop at Line 10 the algorithm generates a set of candidate solutions, from which the best candidates are selected to compose the population.

Before starting generating solutions, the *Heuristic Scaling Value Assignment* procedure presented in Algorithm 6.2 assigns to each ant a weighting factor λ, by which ants exploit the heuristic matrix by different amounts. This procedure allows the construction of solutions biased by different objectives, and thus drives the search to the relevant areas of the design space.

The *Generate Solution* procedure iterates on the edges of the design space, in order to step-wise create a candidate solution by selecting vertices from the design space using

Algorithm 6.2 Heuristic Scaling Value Assignment.

```
1: for $i = 1$ to $h_{size}$ do
2:     $R_i = random\,[0,1]$
3: end for
4: Sort $R$ in ascending order
5: $\lambda_1 \leftarrow R_1$
6: for $i = 2$ to $h_{size} - 1$ do
7:     $\lambda_i \leftarrow R_i - R_{i-1}$
8: end for
9: $\lambda_h \leftarrow 1 - R_h$
```

a transition probability according to Equation 6.3. This procedure is presented in Algorithm 6.4 and is discussed in Section 6.4.2. After generation the candidate solution is evaluated.

$$p_{ij} = \frac{\prod_{n=1}^{|Sdg|} \left[\phi_{ij}\right]^{\alpha} \cdot \prod_{d=1}^{h} \left[\eta_{ij}^{d}\right]^{\lambda_d \beta}}{\sum_{l \in N_i^k} \left[\phi_{init}\right]^{\alpha} \cdot \prod_{d=1}^{h} \left[\eta_{il}^{d}\right]^{\lambda_d \beta}} \tag{6.3}$$

Where:

p_{ij}	is the probability of ant k selecting the edge connecting vertex i and j.
ϕ_{ij}	is the pheromone value for edge connecting vertex i and j.
α	is the magnitude of pheromone influence on the probabilistic decision.
h	is the number of objectives.
η_{ij}^{d}	is the heuristic value for the edge connecting vertices i and j.
β	is the magnitude of heuristic influence on the probabilistic decision.
λ	is the heuristic exponent weighing factor.
N_i^k	is the set of design alternatives that ant k has not yet visited.

When all candidates solutions are generated, the *Crowding Replacement* procedure is used to control which generated solutions will integrate the population, according to Algorithm 6.3.

This procedure compares all solutions generated during the iteration against a randomly selected subset $S\prime$ of S to find the most similar solution in the population. Then the most similar solution is replaced if the newly generated solution is strongly-dominating the other. The similarity between two solutions is measured in the decision space, by using the Hamming Distance according to Equation 6.4. Such a metric identifies the number of shared solution components, where a ratio equal one means that the solutions are completely different, whereas zero means that the solutions are the same. At the end of each iteration, an integer rank is assigned to all solutions in the population, by using the *Fast Non-dominated Sort* procedure and these solutions are used to update the pheromone map by using a $+\Delta\phi$ according to Equation 6.2.

Algorithm 6.3 Crowding Replacement Procedure.

```
1:  for $j = 1$ to $m$ do
2:      $S\prime \leftarrow$ randomly $c$ chosen solutions from $S$
3:      for $k = 1$ to $c$ do
4:          $d \leftarrow distance\left(s_j^{new}, s_k\right)$ calculated according to (6.4)
5:          if $d < leastDistance$ then
6:              $leastDistance \leftarrow d$
7:              $s_{closest} \leftarrow s_k$
8:          end if
9:      end for
10:     if $s_j^{new} \succ\succ s_{closest}$ then
11:         Remove $s_{closest}$ from the population
12:         Add $s_j^{new}$ to the population
13:     end if
14: end for
```

$$Distance = \frac{V_{shared}}{|C\left(V\right)|} \tag{6.4}$$

Where:

V_{shared}	is the number of shared vertices between s_j^{new} and s_k.
$\|Cd\left(V\right)\|$	is the number of vertices in a candidate design s, such that $s = Cd \subseteq Ds$.

When the stop conditions are met, the final population is a set of non-dominated design points (Pareto-optimal). An engineer can select from this population one or more solutions for further investigation, by considering design trade-offs and system requirements.

CPACO-MO combines features from two classes of optimization algorithms, namely ACO and Evolutionary Algorithm, and its complexity is comparable to other algorithms of these classes, in special when compared to the PACO and NASGA-II algorithms. In order to define the influence between different objectives, CPACO-MO sort weights generated for each objective assigned for each solution to be generated in the *Heuristic Scaling Value Assignment* procedure, which has a complexity of $O\left(hN\right)$, where h is the number of objectives and N is the number of solutions generated per iteration. In the *Crowding Replacement* procedure CPACO-MO requires the selection of a subset c of the population S, in order to identify the closest neighbors from each generated solution and performs one dominance test, resulting in the complexity $O\left(hN^2/c\right)$, where c is the crowding factor that determines how many solutions must be compared. Moreover, at every iteration CPACO-MO performs a dominance ranking based on the *Fast Non-dominated Sort* procedure, whose worst case complexity is $O\left(hN^2\right)$. At the end of the iteration CPACO-MO updates the pheromone matrix. In the worst case this matrix has size $O\left(|E\left(D\right)|\right)$, where $V\left(D\right)$ is the number of edges in the design space graph. CPACO-MO's complexity is governed by the non-dominated sort procedure, resulting in the overall complexity of $O\left(hN^2\right)$, which is equivalent to the state-of-the-art NASGA-II algorithm. The Generating solution procedure is presented in the next section, followed by its complexity analysis.

6.4.2 Generation of Candidate Designs

The algorithm proposed to generate candidate designs is based on the breadth-first search algorithm (CORMEN et al., 2009). It iterates on the design space graph and is guided by the DSE rules, which prune locally the design alternatives at a selected decision. A selection function, which is implemented according to an optimization criteria/algorithm, selects the next decision from a list of alternatives. In this way, the algorithm iterates the graph until a final condition defined as a DSE rule holds. Similarly to the breadth-first search algorithm, it discovers all alternatives at distance k from a specific design decision before discovering other alternatives at distance $k+1$. The design candidate generation is presented in Algorithm 6.4.

Algorithm 6.4 Design Candidate Generation.

```
Require: $Ds = \langle V_{Ds}, E_{Ds}, \delta_{Ds0}, \delta_{Ds1} \rangle$          ▷ Design space graph
 1: $s \leftarrow$ TAKE RANDOM DECISION $(V(Ds))$
 2: $V(S) \leftarrow V(S) \cup \{s\}$
 3: $Q \leftarrow \emptyset$
 4: ENQUEUE $(Q, s)$
 5: repeat
 6:     $u \leftarrow$ DEQUEUE $(Q)$
 7:     $A \leftarrow$ CONSTRAINTS $(Ds.Adj[u], u, S)$
 8:     while $A \neq \emptyset$ do
 9:         $v \leftarrow$ TAKE DECISION $(A, u, S)$
10:         $V(S) \leftarrow V(S) \cup \{v\}$
11:         ENQUEUE $(Q, v)$
12:         $A \leftarrow$ CONSTRAINTS $(A, u, S)$
13:     end while
14:     if $(Q \neq \emptyset)$ then
15:         $finished \leftarrow$ CHECK FINAL CONDITION $(S)$
16:         if $(not finished)$ then
17:             if $v$ is a root vertex then
18:                 $A \leftarrow$ CONSTRAINTS $(Rv, u, S)$
19:             else
20:                 $A \leftarrow$ CONSTRAINTS $(Lv, u, S)$
21:             end if
22:             if $A \neq \emptyset$ then
23:                 $v \leftarrow$ TAKE DECISION $(A, u, S)$
24:                 ENQUEUE $(Q, v)$
25:             elsereturn $S = \emptyset$
26:             end if
27:         end if
28:     end if
29: until $finished$
        return $S = \langle V_S, E_S, \delta_{S0}, \delta_{S1} \rangle \,|V_S, E_S \subset Ds$          ▷ Solution sub-graph
```

Algorithm 6.4 assumes there is a graph $Ds = \langle V_{Ds}, E_{Ds}, \delta_{Ds0}, \delta_{Ds1} \rangle$, which represents the design space according to Definition 6.2. The functions δ_{Ds0} and δ_{Ds1} are used to compute the adjacency list of vertex $v \in V_{Ds}$, which is represented as $D.Adj[v]$. A first-in first-out queue Q is used to manage the visited vertices. It also assumes that there is a function CONSTRAINTS, which requires the adjacency list of vertex x and returns a constrained list of adjacent vertices, representing the actual alternative decisions A available from the vertex v. The function TAKEDECISION implements a selection procedure based on some heuristic. It requires a list of alternative decisions A and the current vertex. Then it returns the selected decision. Because of the application of DSE rules, the design space can be composed of multiple components. This means that after the selection of a vertex, there may be no alternative decision available - i.e. no adjacent vertices. Such a situation reveals an infeasible region, where the exploration mechanism cannot find a feasible solution, hence it returns an empty candidate. In order to avoid getting trapped in such an infeasible region, the algorithm uses two auxiliary lists, one containing the root vertices and the other one containing the sink vertices. Both lists can be physically or logically implemented, according to the CGP implementation.

In the Line 1 of Algorithm 6.4, the function TAKERANDOMDECISION initializes the procedure by randomly selecting a vertex from $V(Ds)$ and assigning it to s. The construction of the sub-graph S that represents the solution starts in Line 2, by adding the start vertex s to S. Lines 3-4 initialize the queue Q to contain only the start vertex s. The main loop in Lines 5-14 iterates as long as the final condition does not hold. Line 6 determines the current vertex u and removes it from Q. Line 7 applies the DSE rules by calling the function CONSTRAINTS, which operates on the adjacency list of the current vertex u and assigns the result to A. The inner loop in Lines 8-13 repeats the procedure to construct a solution until no more alternatives exist in A. In Line 9 the function TAKEDECISION selects a vertex from the list A and assigns it to the next vertex v, according to the transition probability in Equation 6.3. The selected vertex is added to S in Line 10, and in Line 11 it is put in the tail of the queue Q, so that it can be explored later. The function CONSTRAINTS verifies if there are still alternatives to be visited from the current vertex u, in Line 12. If there are no more alternatives from the current vertex, the inner loop is finished and it is verified if the solution reaches a final condition in Line 14. Finally the algorithm returns the design candidate in form of a solution sub-graph S in Line 15.

The *Design Candidate Generation* procedure is executed N times, where N is the number of solutions to be generated at each iteration of the DSE process. The initialization at each iteration consists in the selection of a random decision from the design space graph Ds and the initialization of the queue Q. The enqueuing, dequeuing, and assignments are assumed to take $O(1)$ time. If the CGP is physically implemented, randomly picking up a vertex from Ds takes $O(1)$ time. However, if the logical implementation is used, this operation must select one vertex from each design graph in D, so that it takes $O(|D|)$. The construction of one solution sub-graph S takes place in the outer loop, which iterates until the final condition is reached, which vary according to the DSE scenario. Therefore, lets assume that Equation 5.20 and Equation 5.18 represent two final conditions, in order to provide a bound to this iteration. Equation 5.20 defines that all vertices of graph $Dg_1 \in D$ must be mapped, and Equation 5.18 defines that a vertex in Dg_1 can only be mapped once. These two constraints are reasonable and occurs often in mappings between platform layers, such as mapping all vertices from a task graph into an architecture graph. Accordingly, the outer loop will be executed $O(|V(Dg_1)| - 1)$ times in the worst case. In the inner loop, alternatives available at a previously selected vertex are explored. First the procedure CONSTRAINTS is executed in order to prune the current design space. The complexity of such procedure depends on the size of the model, number of constraints, and mainly on the implementation of the transformation engine adopted(HORVáTH et al., 2010; BERGMANN et al., 2008). Because of such a variation, in this analysis it is assumed to be $O(1)$. The procedureTAKEDECISION implements a step-wise heuristic to select the next design decision based on the CPACO-MO's transition probability, whose worst case is $O(h|V(Ds)|)$, where h is the number of objectives and $V(Ds)$ the set of vertices in the design space, because it iterates on the adjacent lists of the selected vertex, which in the worst case is connected to all others vertices. The inner loop is executed in the worst case $O(|V(Ds)|)$ times, which results in the algorithm's overall complexity equal to $O\left(h|V(Dg_1)||V(Ds)|^2\right)$.

6.5 Discussion

The use of CGP and model-to-model transformations to guide the step-wise generation of solutions in the optimization loop leads to an increase of computational complexity, when compared to random operators used by evolutionary algorithms, such as crossover and mutation. However, in addition to such operators, evolutionary algorithms are garnished with repair and other functions that operate on the generated solution, in order to maintain solution feasibility, which also results in an increase of complexity. The following strategies are used to implement such functions:

- penalize bad solutions in the fitness function (evaluation), so that the exploration mechanism avoids the creation of similar solutions in next iterations by itself;
- use repair functions to fix the candidate solution after the generation;
- use complex generation functions to assure the feasibility of the solution.

Clearly there is a trade-off in the implementation of such strategies. On the one hand, computation effort is used to assure generation of feasible solutions and to save time later by avoiding to repair or discard candidates due to their infeasibility. On the other hand, quick generation accelerates the exploration, by assuming the risk of wasting time when repairing, discarding or evaluating a false candidate.

In (BLICKLE; TEICH; THIELE, 1998) a Cartesian graph product with an additional manual definition of mapping edges is used to define a specification graph (design space). Such a manual mapping definition can be compared to the "structural" and "pre-defined decisions" DSE rules presented in Chapter 5. The approach proposed in (BLICKLE; TEICH; THIELE, 1998) allows for a flexible definition of DSE scenarios. However, after random operators specific "allocation", "bind", "check for valid binding", and "update allocation" functions are used to repair the solutions generated by the exploration mechanism, adding in this way an overall quadratic complexity to the optimization loop. Besides such functions to apply "structural" and "pre-defined decisions" DSE rules on solutions, no information on the application of non-functional constraints is provided, except that a penalty strategy is adopted in the fitness function, which means that illegal solutions are still being generated and evaluated. Therefore, this causes waste of computational effort, which in some cases can be very high due to long evaluation cycles.

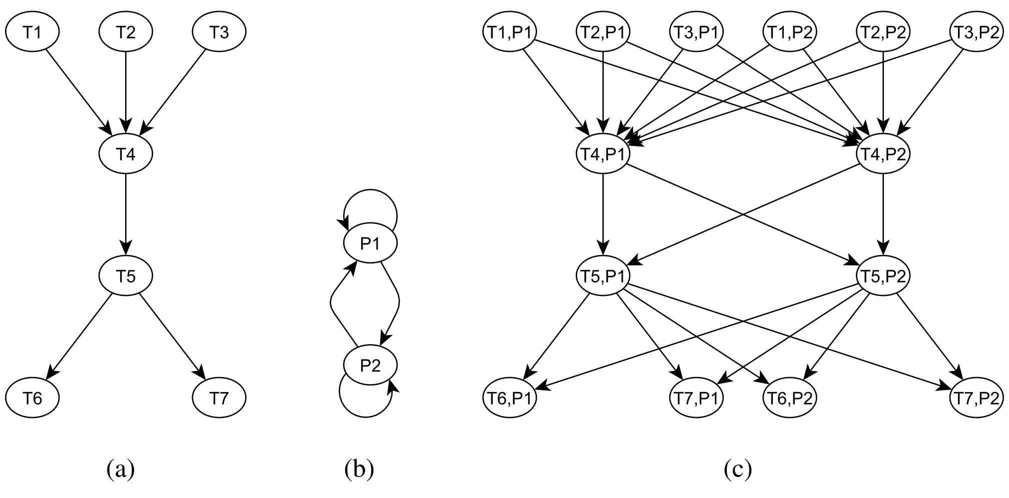
In (DICK; JHA, 1998) the application of random operators is limited, and the concept of cluster of solutions is introduced to assure the structural feasibility. Although some checks are performed to verify if a solution meets the NFRs, bad solutions are allowed to be in the solution set and are penalized in the fitness function. The goal is to save computational effort by reducing the effort on the verification of feasibility and to allow these bad solutions to influence the next generations through genetic random operators. However, customized operators to generate new solutions adopt Pareto-ranking algorithm, and hence adds quadratic complexity into the optimization loop, in order to sort the solutions prior to the application of genetic operators. This approach also uses a specific coding of problem and solution into the exploration mechanisms, thus only previously implemented design activities, namely allocation, mapping and scheduling, can be performed without any flexibility. Moreover no mechanism to define additional constraints is provided, so that no mechanism to assure feasibility is provided. Therefore an evaluation step may fail, when implemented by other tools.

In a similar way, (KANGAS et al., 2006), (ERBAS; ERBAS; PIMENTEL, 2003), and other works present similar implementations, hence it is believed that the algorithm presented in this chapter still has a competitive performance. Furthermore, the resulting worst case complexity occurs only in one specific case, when all graphs involved in the product are fully connected. Such a scenario is unusual and may indicate bad design architectures, such as a set of tasks completely connected to each other, mapped to a communication structure that allows direct point-to-point communication between all processors. In the proposed method each selected design decision, as well as previously evaluated solutions helps to reduce the set of alternatives, hence the vertices to be visited, because the CONSTRAINT procedure removes redundant decisions and decisions that lead to infeasible solutions. In this way Algorithm 6.4 proposes a compromise between flexibility and performance to improve the DSE process.

The following example illustrates the generation of one solution by applying Algorithm 6.4. It highlights the differences between physical and logical implementations of the CGP and the application of constraints to prune the design space.

Let graph Dg_t be a task graph, depicted in Figure 6.7(a), and graph Dg_p be a processor graph representing two processors connected point-to-point, as illustrated in Figure 6.7(b). Figure 6.7(c) shows the CGP $Dg_t \otimes Dg_p$, which represents the full design space for mapping the task graph Dg_t into processor graph Dg_p, without considering any DSE rules. Notice that although graph Dg_p is fully connected, the edges in graph $Dg_t \otimes Dg_p$ are limited by the connectivity in Dg_t, because vertices in $Dg_t \otimes Dg_p$ are adjacent only if the referenced vertices in Dg_t and Dg_p are both adjacent. Therefore, the number of alternatives to be evaluated at each step is reduced and depends on the connectivity of each graph in CGP.

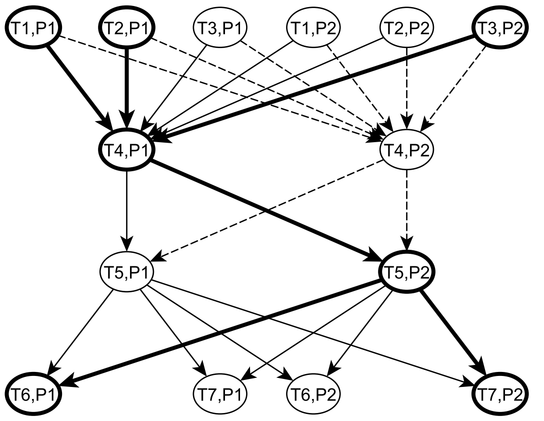
Figure 6.6: Example 1 of hard feasibility in DSE: (a) Task Graph Dg_t; (b) Processor Graph Dg_p; (c) Design Space $Dg_t \otimes Dg_p$;



The generation of solutions relies on the graph's edge source function π_1 and target function π_2, in order to iterate on the design space graph generated by an implementation of the CGP. In a physical implementation the complete design space graph is computed prior to the exploration process once, which results in a storage complexity equal to $O\left(\Pi_n^i \left|\left(Dg_i\left(V\right)\right|\right)$, where n is the number of design graphs in the CGP. The design space is constantly pruned at each iteration, whenever infeasible or illegal edges and vertices are found. In this way, the computation effort required to generate the complete design space graph is mitigated, because more and more alternatives are permanently removed, while new solutions are evaluated. The benefit is the reduction of the computation time by avoiding dynamically verification of paths for feasibility at each iteration and reducing the execution of the DSE rules body, by preventing rule matches. Moreover, sub-sets of solutions (vertices and edges) that were evaluated previously can be definitively removed if they do not meet requirements, avoiding in this way unnecessary evaluation. However, the drawback is that vertices of the design space are visited once, even if they will never be required by the exploration mechanism. Such limitation is also presented in approaches that uses BDD such as (NEEMA et al., 2003) and (SCHLICHTER et al., 2006).

Figure 6.7 illustrates a solution generated by using a physical implementation of the CGP and Algorithm 6.4 for the set of graphs presented in Figure 6.6. For this sample, all vertices of the design space were visited at least once, in order to produce the CGP. The edges and vertices highlighted with a bold line indicate the selected decisions. Even although all vertices are computed, additional DSE rules can be defined, so that more edges are definitively removed at each iteration, reducing the number of edges to be evaluated in the TAKEDECISION procedure. For example, lets consider that the vertex $\langle T_4, P_2\rangle$ is part of a previously generated solution and that after its evaluation it was found out that processor P_2 cannot execute task T_4 with the required performance. Then the performance metric is stored in the edge attributes and used by the DSE rules to remove this alternative from the design space, by applying the NFR DSE rules. Such an example is illustrated in Figure 6.7 by the dashed edges representing the removed edges.

Figure 6.7: Solution generation example for physical CGP implementation: Solution graph.



Differently from the physical implementation, the logical one only generates and visits a vertex if it is required at each iteration. This leads to less storage complexity, which is $O\left(\Sigma_n^i \left|Dg_i\left(V\right)\right|\right)$, where n is the number of design graphs in the CGP. However, as the design graphs are already stored in the DSED model, there is no additional storage.

This approach also reduces the computational effort at initialization, because the CGP is not computed. However, at each iteration infeasible and illegal vertices appear in the adjacency list of the current vertex before the application of DSE rules and they must be pruned before making the alternatives available to the TAKEDECISION procedure. This cost is required in order to dynamically adjust the process to user requirements, in this way improving flexibility during the exploration process. The application of DSE rules is the step where user constraints and heuristics can be applied to guide the exploration process. After the application of DSE rules, the list of alternatives is limited, reducing in this way the computational effort to evaluate alternatives. The verification of solution feasibility is reduced as well, because only feasible alternatives are available for selection.

Figure 6.8 illustrates the generation of one solution according to Algorithm 6.4, from the set of graphs presented in Figure 6.6, by using a logical implementation of CGP. In this illustration dashed lines of edges and vertices are used to highlight the logical representation of the product, while bold lines indicate the current vertex used by the solution generation procedure to identify and evaluate alternatives. The vertices filled in gray represent the next selected decision, which will be queued and have its neighbors vertices evaluated in the next step.

Figure 6.9(a) shows the first state, where the vertex $\langle T_1, P_1\rangle$ was randomly selected. In Figure 6.9(b) the selected vertex and the alternatives at this step are shown. The current vertex is used by the generation algorithm to identify and select one alternative design decision from one of its adjacent vertices. The vertex $\langle T_4, P_1\rangle$ is highlighted in gray, in order to indicate the selected vertex.

In the next step, illustrated in Figure 6.9(c), the current vertex is updated to vertex $\langle T_4, P_1\rangle$, because the vertex $\langle T_1, P_1\rangle$ has no more alternatives to select. At this step, the adjacent alternatives of the vertex $\langle T_4, P_1\rangle$ are available for selection, except the vertex $\langle T_1, P_1\rangle$, which is pruned by the DSE rules because it was previously selected. The vertex $\langle T_3, P_{21}\rangle$ is selected and queued, while there are still other alternatives to be selected. As the exploration proceeds, edges and vertices are pruned due to the application of "structural", "non-functional requirement" and "previously-defined design decision" DSE rules. For example, in Figure 6.9(d) two alternatives for mapping the task T_2 are still available, and the alternative $\langle T_2, P_1\rangle$ was selected. The vertices $\langle T_1, P_1\rangle$, $\langle T_4, P_1\rangle$, and $\langle T_3, P_2\rangle$ were pruned from the alternatives, because they were already selected. If there is a "pre-defined design decision" DSE rule requiring the mapping of task T_2 to a specific processor, lets say P_1, the vertices $\langle T_2, P_1\rangle$ and $\langle T_2, P_2\rangle$ would never appear as alternatives in any step, since the first application of DSE rules would have included this map in the solution. The reduction in the number of alternatives after the selection of design decisions can be noticed in Figure 6.9(e).

The design space expands when new areas of the design space are reached, as illustrated in Figure 6.9(f), and it is reduced after visiting some vertices in the adjacency or a number of similar solutions is found. After selection of the last decision, the vertex $\langle T_2, P_2\rangle$ is added to the solution, as illustrated in Figure 6.9(g). The final sub-graph S, which represents a candidate solution can be observed in Figure 6.9(h).

Besides the abstraction and flexibility provided by the adoption of CGP to represent the design space, this method is also an approach to overcome the common difficulty of exploration mechanisms to escape from infeasible areas present in highly constrained design spaces. Figure 6.9 illustrates a trap example from (SCHLICHTER et al., 2006), where the task graph Dg_t and the resource graph Dg_r, shown in Figure 6.10(a) and Figure 6.10(b) respectively, must be mapped. Following the method presented in (BLICKLE;

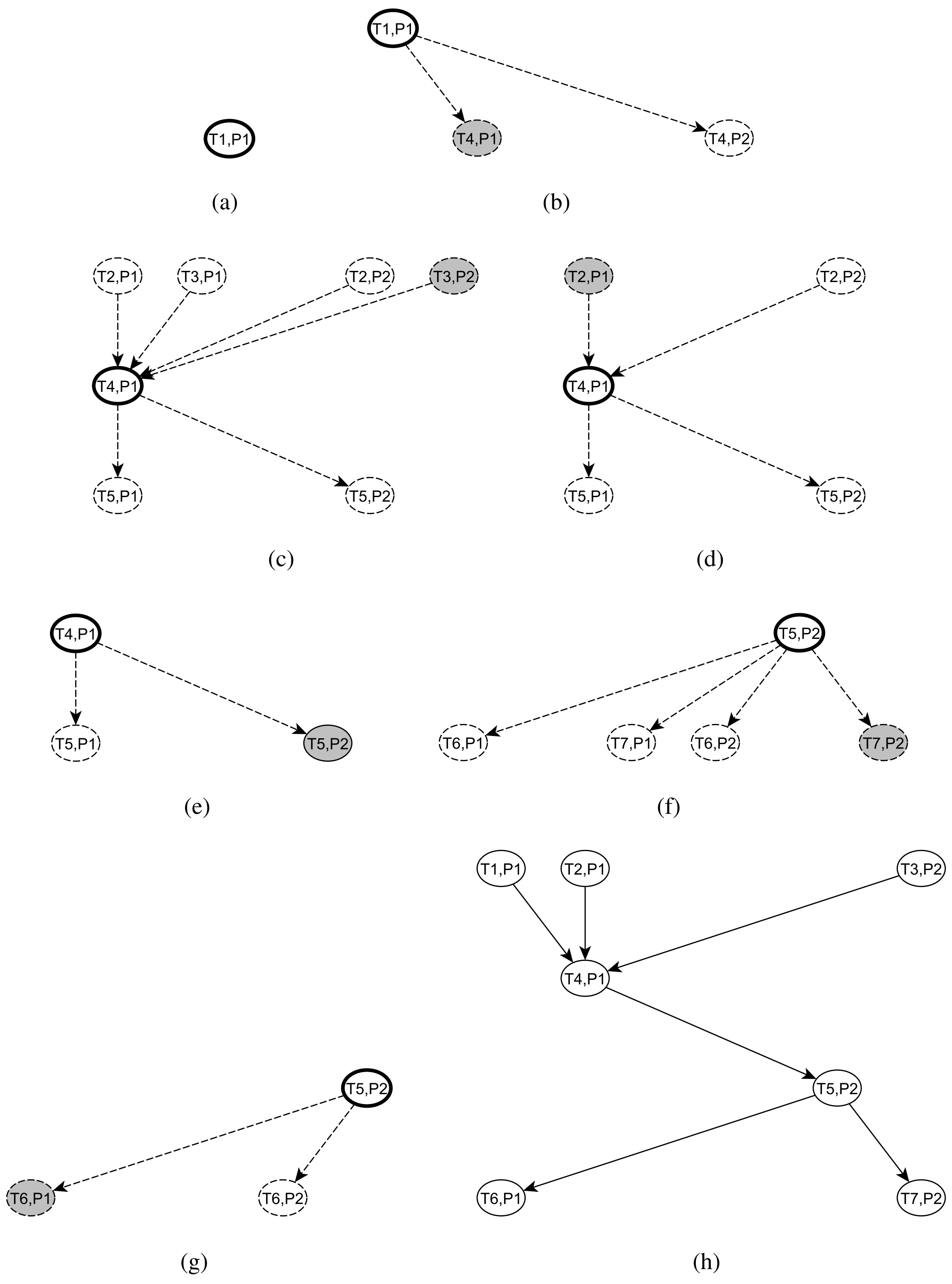
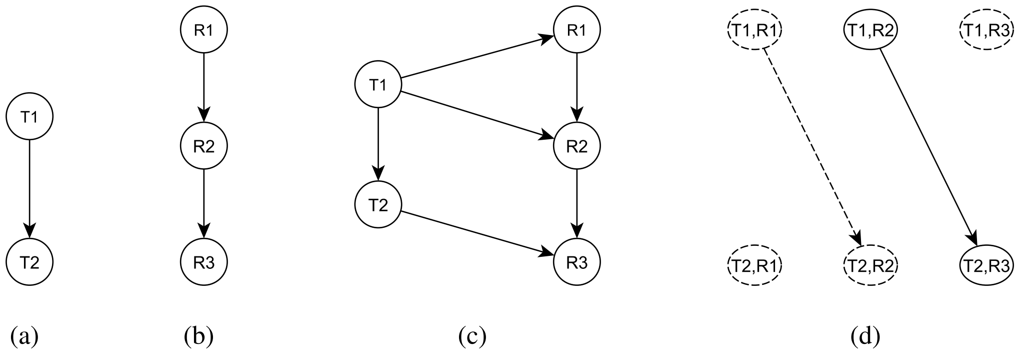


Figure 6.8: Solution Generation Example for logical CGP implementation.

TEICH; THIELE, 1998), a specification graph is built, in order to defines how tasks in graph Dg_t can be mapped into vertices of the resource graph Dg_r. The specification graph is shown in Figure 6.10(c). This example illustrates the unavoidable inter-dependence between design decisions, because the decision of allocation of resources influences the possible mappings, while the mapping decision influences the possible allocation, depending on the order of decisions. Therefore, sequential decoders, such as the one presented in (BLICKLE; TEICH; THIELE, 1998), are not able to find a valid solution in such a highly constrained design space. These algorithms cannot find a solution because T_1 is mapped to R_1 first, as it does not violate any restriction. However, such a map prohibits a feasible mapping of task T_2, because there is no link between R_1 and R_3. In this example the only feasible solution is the sub-graph $S = \{\langle T_1, R_2\rangle, (T_2, R_3)\}$, which can be found by the method presented in this chapter.

Figure 6.9: First example of hard feasibility in design space: (a) Task Graph T; (b) Resources Graph R; (c) Specification Graph S; (d) CGP-based Design Space Ds.



(SCHLICHTER et al., 2006)

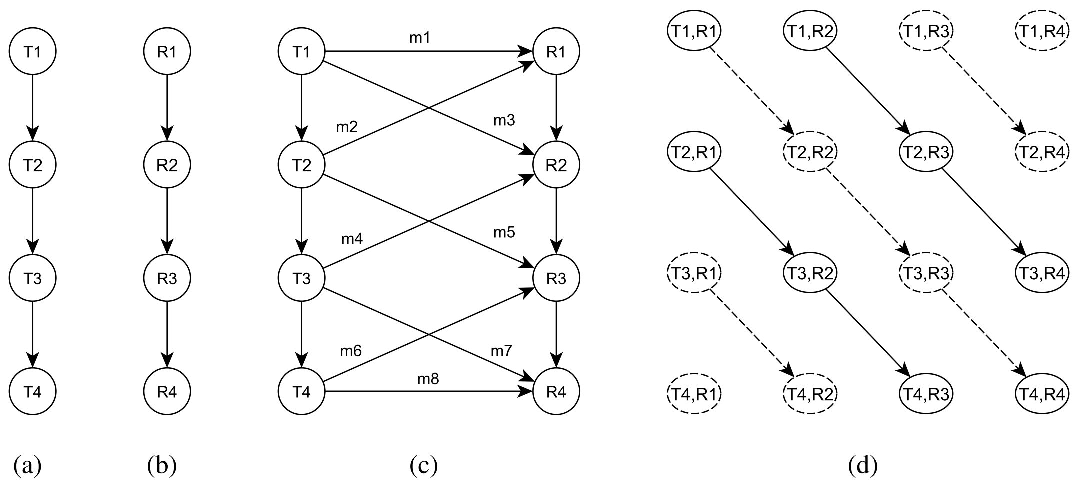
Assuming the same constraints presented in (BLICKLE; TEICH; THIELE, 1998), an DSE rule must implement the constraint defined in Equation 6.5, which enforces communicating tasks to be mapped to the same resource or the existence of an edge in the resource graph connecting two communicating tasks. In this way the generated design space contains only the feasible solution, so that the generation of candidates has no other alternative than selecting the sub-graph S containing the vertex set $S(V) = \langle T_1, R_2\rangle, \langle T_2, R_3\rangle$ as a solution for this mapping problem. Figure 6.10(d) illustrates the resulting design space, by applying the proposed approach. The dashed lines are used to indicate pruned vertices and edges.

It is important to highlight that the definition of additional DSE rules, which are called inside the generate solution procedure presented in Algorithm 6.4, is different from other approaches, because most approaches, such as (KANGAS et al., 2006), (DICK; JHA, 1998), (ERBAS; ERBAS; PIMENTEL, 2003) and others do not offer an interface for that purpose.

$$\exists (v, v\prime) \in Ds \mid \pi(v_r) = \pi(v_r\prime) \ \wedge \ \exists (\pi(v_r), \pi(v_r\prime)) \in R \tag{6.5}$$

Another example is presented in (SCHLICHTER et al., 2006), in order to illustrate how exploration mechanisms can fail during the generation of solutions. This example is shown in Figure 6.10. Another task graph Dg_t and resource graph Dg_r are shown in Figure 6.11(a) and Figure 6.11(b) respectively. The specification graph used to define the mapping problem is shown in Figure 6.11(c). In this example there are only two feasible solutions, which are identified by the two edge sets of the specification graph: $\langle m_1, m_2, m_4, m_6\rangle$ and $\langle m_3, m_5, m_7, m_8\rangle$. Because task T_1 and T_4 do not have common neighbors, exploration mechanisms may be trapped after deciding to map task T_1 into resource R_1 (edge m_1) and task T_4 into resource R_4 (edge m_8). Figure 6.11(d) illustrates the design space graph $Dg_t \otimes Dg_r$ produced, whose dashed lines identify pruned edges and vertices after the application of DSE rules. The design space was produced according to the method proposed in this chapter and the constraints specified in the specification graph are illustrated in Figure 6.10(c).

Figure 6.10: Second example of hard feasibility in design space: (a) Task Graph T; (b) Resources Graph R; (c) Specification Graph S; (d) CGP-based Design Space Design Space Ds.

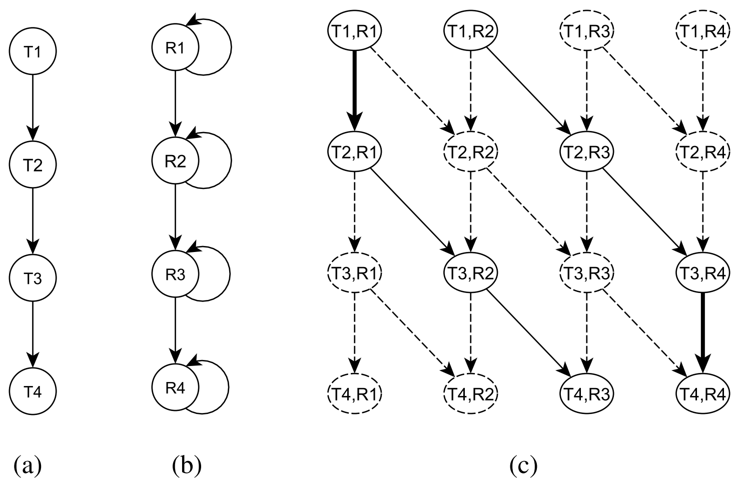


(SCHLICHTER et al., 2006)

In order to avoid getting trapped in design space regions like the one presented in the example above, the lists of root and sink vertices are used, so that the search can migrate to another region of the design space and proceed. If the queue of visited vertices is empty due to the lack of alternatives and the current state of the solution does not satisfy the stop condition, one random vertex is selected from these lists according to the type of the previously selected vertex: it is selected from the root list if there is no incoming edges; it is selected from the sink list in other cases. Such an approach allows for searching in design spaces whose graph is formed by multiple components due to the removal of edges and vertices caused by the application of DSE rules. Consequently, in this example the search is able to find another feasible design space, for example migrating from $\langle T_1, R_1\rangle$ to $\langle T_2, R_1\rangle$, as it is the only root vertex available. Because the vertex $\langle T_4, R_4\rangle$ is a sink vertex, and the edge that reaches it is removed due to the structural constraints, it can be selected only in the first step of the solution generation by random selection of the first vertex. Therefore, the vertices $\langle T_1, R_1\rangle$ and $\langle T_4, R_4\rangle$ cannot be simultaneously selected.

Although the design space like the one shown in the previous example can arise due to some requirements, in other scenarios where communicating tasks can be mapped into the same resources, like in the previous example, the specification of the resources can be alternatively specified by allowing loops which represent the inter-communication inside a resource. In this way, the contiguous feasible region in the design space is longer, requiring less iterations on the root/sink vertex list and better exploitation of dependencies between tasks. Figure 6.11 illustrates this approach, where to the resource graph Dg_r, from Figure 6.11(b), were added loop edges to each resource, as illustrated in Figure 6.12(b). Thereafter, the graph $Dg_t \otimes Dg_r$ contains additional vertical edges, as illustrated in Figure 6.12(c). Like in the previous example, the dashed lines indicate pruned vertices and edges. The bold edges are the two new edges that create the only two feasible paths, which connect all vertices in each solution.

Figure 6.11: Example of alternative specification of design graphs: (a) Task Graph Dg_t; (b) Resources Graph Dg_r; (c) CGP-based Design Space Design Space Ds.



In this chapter it was presented a layer mapping method that represents the design space by using CGP, in order to provide abstraction and automation for DSE. In this method, design graphs, which represent platform layers, are abstracted and combined by using a CGP. Edges of the resulting graph product represent alternative mappings between multiple layers, and the selection of one vertex in this graph represent, simultaneously, multiple design decisions. The DSE Mapping Problem was reviewed to use a CGP representation. Moreover, elements to represent the CGP and the reviewed DSE Mapping Problem were included into the DSED metamodel.

In order to solve the new DSE Mapping Problem a heuristic, which exploits the CGP and the MDE approach, was implemented. This heuristic adopts an exploration mechanism based on CPACO-MO and generates design candidates by iterating in the CGP edges. Model-to-model transformations are used to prune the design alternatives at each iteration, assuring that only feasible decisions are selected to build a solution.

By using the CGP for design space representation, DSE is performed for multiple design activities simultaneously, as each product represents a design activity. Specific properties of this product, such as a restriction on the adjacency, reduce the number of available alternatives, as the navigation on the design space is performed through the edges. Moreover, this representation overcomes the interdependence between design activities, as one vertex in the design space represents multiple design decisions at the same time and the graph is iterated in different directions. This abstraction also exposes the dependencies between elements, such as communication or priority, between elements and it is well suited to combine the communication in multiple hierarchies, such as classes, task, processors, and systems. Moreover, this proposal improves the ability of the exploration mechanism to escape from infeasible regions in the design space. Finally, the application of DSE rules, by means of model-to-model transformations, provides the flexibility to define constraints on the design space and the guidance to the exploration mechanism.

7 TOOL SUPPORT

In this chapter the tools adopted or implemented to support the proposed methodology are presented. The first section starts by identifying the tool flow requirements and assumptions. It also presents the MDE Technological Framework adopted, which is based on the Eclipse Modeling Project. An overview of the tool flow is presented, and the contributions of this work are highlighted. Then the Domain Specific MDE Framework implemented to support the methodology introduced in Chapter 5 and Chapter 6 is described. Such a framework contains common metamodels, models and transformations used by the evaluation and exploration tools, which are also described in the following sections. Moreover, a tool for automatic generation of DSED models from UML ones is described.

7.1 Tooling Overview

Supporting tools are essential to the success of a DSE methodology. Besides an optimization algorithm implemented to automate the search in the design space, other tools are useful to improve the DSE process. Tools for assessment of alternative designs are required to automatically evaluate the solutions proposed by the DSE tool. Tools to create and adequately operate on the models are also important, in order to reduce the development effort. The proposed methodology is supported by the following tools:

- MDE technological framework;
- Domain Specific MDE models and transformations - for embedded systems in general and specifically for DSE;
- an implementation of two solver methods in a tool that orchestrates the DSE process;
- an automatic design evaluation tool;
- a tool for extraction of partial DSED models from UML models.

The adopted MDE technological framework and the implemented tools are prototypes that operate in a limited context, in order to demonstrate the concepts proposed in the methodology. Although the methodology is independent of the design language, only support for UML was implemented. Additional languages could be supported by implementing the adequate transformations to automate the weaving and back annotation processes or the creation of DSED models from other languages. Moreover, the interface tools, such as the back annotation and DSED generation, are tightly coupled to the adopted modeling languages, tools and development environment. Therefore, there is no guarantee that working with different versions of the tools or from other providers can be done without extra effort for integration. The prototypes were developed by using Java 1.6 and the Eclipse IDE Juno 4.2. The tools must support EMF 2.8.1, used as meta-data repository technology and the foundation for interoperability with other EMF-based tools, such as a transformation engine and (meta)model editors.

7.1.1 Adopted MDE Technological Framework and tools

As presented in Chapter 4, an MDE Technological Framework consists of tools to support common tasks for MDE, independently from the application domain. The prototypes were developed by using the framework provided by the Eclipse Modeling Project, which includes tools for metamodeling, model transformation, model edition, etc. However, other tools were also required to support developing the prototypes. Specifically the following tools/technologies were adopted:

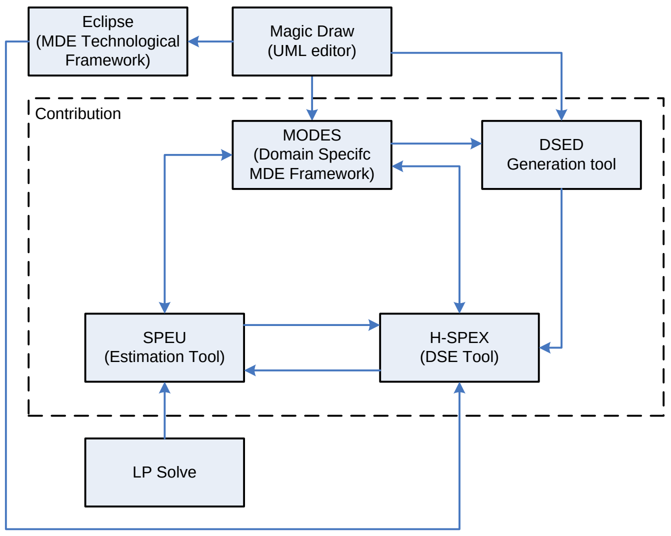
- Eclipse Modeling Framework (EMF) as meta-data repository and code generator;
- ECORE for metamodeling;
- Xtend, ATL and VIATRA II for model transformation;
- Atlas Model Weaving (AMW) for model weaving, e.g. design and DSED models;
- Magic Draw from No Magic for UML modeling;
- ANTLR for creating the TGFF Injectors - parsers used to transform text files generated by the Task Graph for Free (TGFF) tool (DICK; RHODES; WOLF, 1998) into TGFF ecore models;
- LP Solver and LP Solver Java Wrapper for solving Implicit Path Enumeration Problem during the design evaluation.

7.1.2 Contributed tools

Figure 7.1 shows the tools used to support the MDE methodology for DSE. The tools contributed by this thesis are identified by the surrounding dashed box. The figure also shows the adopted user front-end tools and the data flow between all these tools.

The MOdel-Driven engineering for Embedded System (MODES) framework (NASCIMENTO; OLIVEIRA; WAGNER, 2007) consists of a set of metamodels and transformations to capture different views of embedded systems and to provide support for the integration of Domain Specific MDE tools. The metamodels and transformations are presented in Section 7.2. The SPEU tool provides the support for quick system evaluation by using static analysis. Details on the evaluation tool are provided in Section 7.3. The CPACO algorithm was implemented into the H-SPEX tool, which orchestrates the DSE process and the interaction with the SPEU tool. Implementation details of H-SPEX are presented in Section 7.4. Finally, a tool to extract a DSED model from a UML one is available, in order to reduce the effort to define DSE scenarios and reuse the design artifacts to generate constraints. This tool is described in Section 7.5.

Figure 7.1: Overview of Supporting Tools.



7.2 MODES: MDE framework for Embedded Systems

Originally, MODES provided the components System Designer, Application, Platform and Implementation Managers, which transform UML models into internal models conforming to metamodels proposed to represent applications, capturing functionality by means of processes communicating by ports and channels; platforms, indicating available hardware/software resources; mappings from applications into platforms; and implementations, oriented to code generation and hardware synthesis (NASCIMENTO; OLIVEIRA; WAGNER, 2007). MODES relies on the Y-Chart approach, focusing on complex models to represent the Y-Charts axises. MODES was developed in the context of the Embedded System Laboratory (acronym in Portuguese - LSE[1]) of Federal University of Rio Grande do Sul, and it was originally published with co-authors in (NASCIMENTO; OLIVEIRA; WAGNER, 2007). In the view of the author of this thesis, simple and specialized metamodels can be more effective and improve the development process. Therefore, the MODES framework was extended, by refactoring its original metamodels in smaller parts and adding other metamodels to represent different system views or development needs. Although separation of concerns is still available as multiple system views, after the extensions there is no more direct identification between MODES and the Y-Chart approach. Also included into MODES was a transformation that extracts information from front-end languages such as UML and converts it into models conforming to the internal metamodels.

Therefore, now MODES is a library of domain specific metamodels, models, and transformations, which are specified or implemented to support the development of embedded systems. MODES provides metamodels in ECORE, as well as APIs and model editors, which were generated from these metamodels by using EMF tools, thus they can be deployed as Eclipse Plug-in. Model-to-model transformations are provided to support generation of MODES models from well-adopted languages and to provide interoperation between tools, e.g., H-SPEX and SPEU. There are also model-to-text transformations to generate source code, configuration scripts, and other development artifacts, such as input file for the UPPAAL model checking tool (LARSEN; PETTERS-

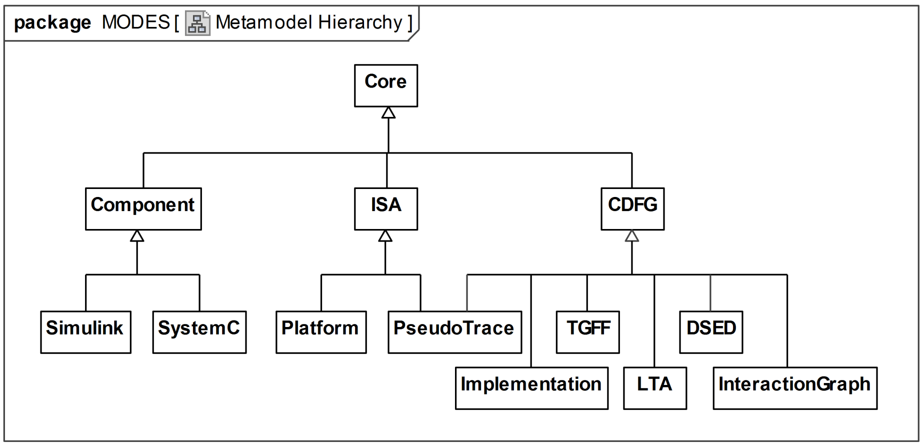
[1]http://www.inf.ufrgs.br/ lse/

SON; YI, 1997). Text-to-model transformations were included to inject models in TGFF and Simulink text file format in the framework as TGFF and Simulink models based on ECORE, so that they can be accessed by high-level Java APIs and model model-to-model transformations. Additional transformations can also be implemented, based on the provided metamodels or the generated Java APIs, in order to operate the models or perform some development tasks. As such, MODES is the integration point of multiple embedded systems development methods, which requires specialized views of the system. Nowadays, with the advances in model management, tools such as AtlanMod MegaModel Management (AM3)[2] or MoScript[3] can be used to improve the MODES framework, so that the library of models and transformations can be adequately managed, providing support for (meta)model registration, version control, (meta)model extensions, and others. The next sections present the metamodels and transformations available in the MODES framework.

7.2.1 Basic Metamodels

Two metamodels are provided to support the representation of basic concepts. Due to their simplicity, no diagrams are presented. One is the Core metamodel, which provides the basic elements shared by all other metamodels. This metamodel is composed of `NamedElement`, which has a name and a description. It is also associated with an `Annotation` element, which contains a string key and a value used to provide information about `NamedElement`. The second metamodel represents a CDFG and is the basis for many other metamodels. The CDFG metamodel extends the `NamedElement` from the Core metamodel, by defining a `Graph`, containing `Vertex` and `Edges`. Figure 7.2 illustrates the metamodels provided by the MODES framework and the specialization hierarchy as UML classes. The DSED metamodel was described in Chapters 5 and 6. It was also integrated in the MODES framework. The other metamodels provided by MODES and used in this thesis are described in the following sections.

Figure 7.2: MODES metamodel hierarchy.



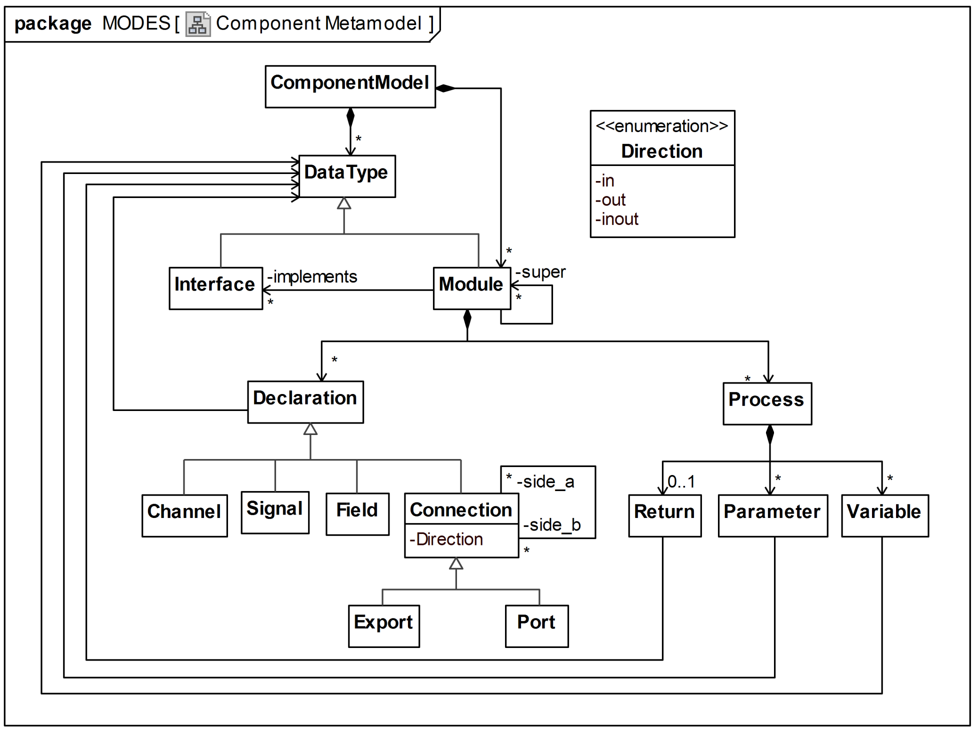
7.2.2 Component Metamodel

The application structure can be represented in different ways, however component or class models are the most common way to represent the composition structure. Moreover,

[2]http://www.wiki.eclipse.org/AM3
[3]htpp://www.eclipse.org/MoScript

components and classes are the basis for many other DSLs, such as Simulik, SCADE and SystemC. Figure 7.3 presents the MODES Component metamodel.

Figure 7.3: MODES Component Metamodel.



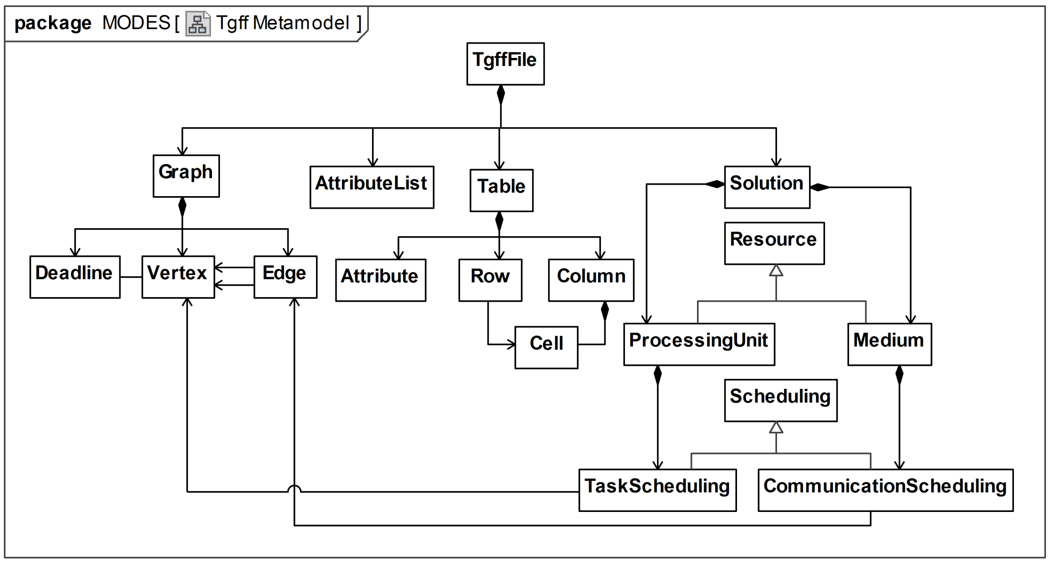
Conforming to this metamodel a `ComponentModel` captures the application structure in terms of a set of `Modules`. Each `Module` has `Declarations` and `Processes`. Specialization of `Modules` is possible due to the `super` association between `Modules`, with same class' hierarchy semantic. `Declarations` are associated with `DataTypes` and can be a `Channel`, a `Connection`, or a `Signal`. These concepts come from hardware description languages, such as VHDL, but are also used in some software MoCs. `Channels` are used by `Processes` to send or receive messages. `Connections` have a `Direction`, which can be `in`, `out` or `inout`, representing the communication direction through `Port` and `Export` used to interconnect `Modules`. `Signals` are used to specify shared memories for processes. A `Declaration` can also be a `Field`, which defines the state of a `Module`. A `Process` contains `Parameters`, a `Return` value, and `Variables`, which correspond to local memories.

The behaviors of `Processes` are associated to MoCs, which are not represented in this metamodel. This association allows the translation from an abstract behavior description to a specific MoC and the execution of algorithms to automate design tasks. Currently, two MoC's are supported, a CDFG representation defined by the Interaction Graph metamodel and a Labeled Time Automata (LTA). These metamodels are briefly described in Section 7.2.8.

7.2.3 Task Graph / TGFF Metamodel

Task Graph for Free (TGFF)(DICK; RHODES; WOLF, 1998) is a tool that generates graphs and resource tables, which represent respectively task graphs and information on system resources, in order to produce mapping, scheduling and allocation problems as input for synthesis and DSE methods. The outputs of TGFF are files in a plain text format, which represent graphs and tables. MODES provides a TGFF metamodel in order to represent the outputs of the TGFF tool and inject them into the MDE Framework. Moreover, this metamodel is also used to represent task graphs extracted from the system specification, providing in this way a common representation for this information. Figure 7.4 illustrates the TGFF metamodel.

Figure 7.4: MODES TGFF Metamodel.

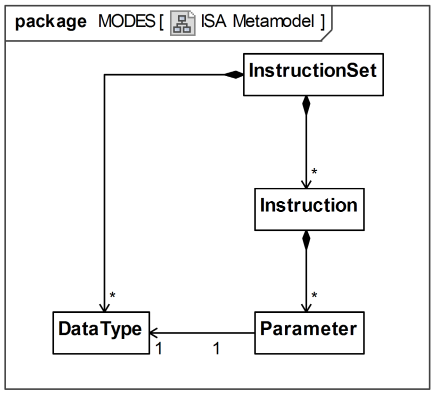


7.2.4 Instruction Set Architecture Metamodel

The Instruction Set Architecture (ISA) metamodel is aimed to represent not only the instruction set of processor architectures, but also to represent high-level instructions, such as the Symbolic Instruction defined by SPEU, DERCS (WEHRMEISTER et al., 2008) Actions and UML Actions. The `Annotation` element inherited from MODES' Core metamodel is used to provide information on instruction size, execution cycles, etc. Such information is used to characterize a processing unit's instructions or services provided by components from the platform repository (see Section 7.2.6). Figure 7.5 illustrates the MODES' ISA metamodel. The ISA metamodel is an extension of the method proposed in (OLIVEIRA et al., 2006) to represent architectural information and reuse it to improve static system analysis.

The specification of ISA models is done by using the Eclipse EMF reflective editor, and currently two models are provided by MODES. The Symbolic ISA model was contributed by the SPEU evaluation tool to represent the high-level behavior independent of platform. The Femtojava Instruction Set defines the Femtojava ISA for the Femtojava micro-controller (ITO; CARRO; JACOBI, 2001), which is a stack-based processor that implements the Java virtual machine. The Femtojava ISA contains also the instruction extensions to support the real-time API presented in(WEHRMEISTER et al., 2005). These models are used by SPEU, which requires a mapping between Symbolic ISA model into real ISA models, in order to extract estimation from UML models. Other ISA can be added to the MODES' model repository, when new processor architectures are included in the Platform Repository. Other ISAs of Virtual Machines or abstract instruction set, such as the UML ALF(OMG, 2013) can also be integrated in the repository.

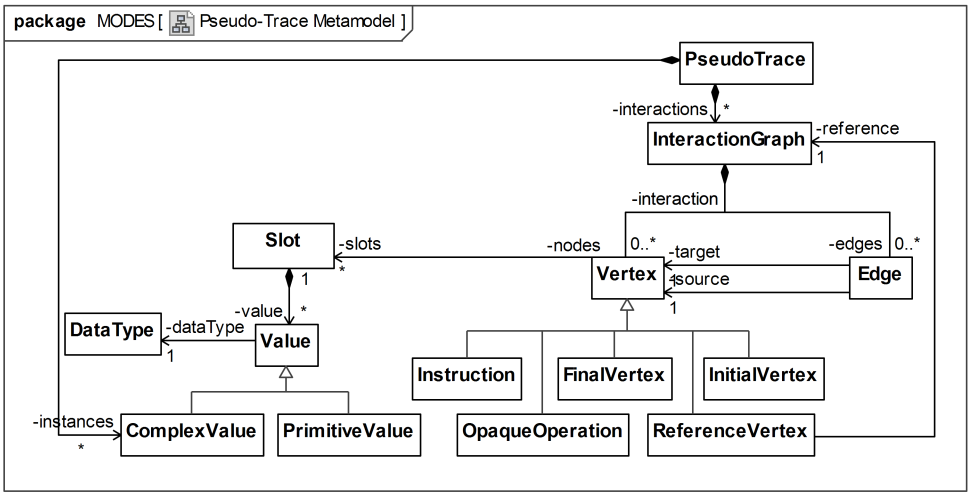
Figure 7.5: MODES ISA Metamodel.



7.2.5 Pseudo-trace Metamodel

The Pseudo-trace metamodel represents the expected application behavior at run-time, in order to allow a quick evaluation of the system. The application behavior is represented by the Interaction Graph metamodel in form of CDFGs. The Pseudo-trace metamodel extends the Interaction Graph metamodel, by adding into the metamodel elements to represent the expected run-time information, such as structural and functional constraints on the execution path and defined execution scenarios. This representation is based on the Implicit Path Enumeration method (LI; MALIK, 1995), which is used to estimate the worst-case execution path, and Software Performance Engineering (SMITH; WILLIAMS, 2003), which inspired the method to define execution scenarios. The combination of both methods in an MDE approach was presented in (OLIVEIRA et al., 2006) and extended in this thesis, in order to allow its application to systems described in different languages. Figure 7.6 illustrates the Pseudo-trace metamodel.

Figure 7.6: MODES Pseudo-trace metamodel.



A `Pseudo-trace` consists of a set of `InteractionGraphs` with `Vertex` and `Edges` representing the CDFG extracted from the application. A `Vertex` can be an `InitialVertex` or a `FinalVertex`, which represent the beginning and end of an `InteractionGraph`, respectively, and holds information about the number of times the InteractionGraph is executed. Such information is used to extract the structural constraints, explained with more detail in Section 7.3.2. A Vertex can also be a ReferenceVertex, which associates a Vertex to a complete Interaction-Graph, thus reducing the complexity by capturing the application behavior in a hierarchical way. The abstraction level can be controlled by using the OpaqueOperation, which extends Vertex, in order to represent an operation at arbitrary granularity. In this way, OpaqueOperation can be used to represent a single instruction or a full execution of a complex algorithm, which will be associated to a cost that characterizes the operation. Other control flow vertices, such as parallel execution, conditions, loops, and operations, such as assignment, creation and destruction of objects, which are required to capture the behavior, are represented by the Instruction element, which is defined in ISA models. The Annotation element from the Core metamodel is used to provide functional constraints such as lower and upper bounds for execution of loops and conditions. For example, let's consider that a vertex n_2 must execute 5 times if another vertex v_1 is executed due to a loop and conditional statements between them, than the functional constraint expression $v_2 = 5v_1$ will be stored in an Annotation associated to the vertex v_2. Moreover, each Vertex contains a list of Slots, which store dynamic Values computed at run-time. Such run-time information is not mandatory. However, if defined, they can be used by data-flow analysis, in order to improve the extraction of structural and functional constraints, and improve the accuracy of estimations.

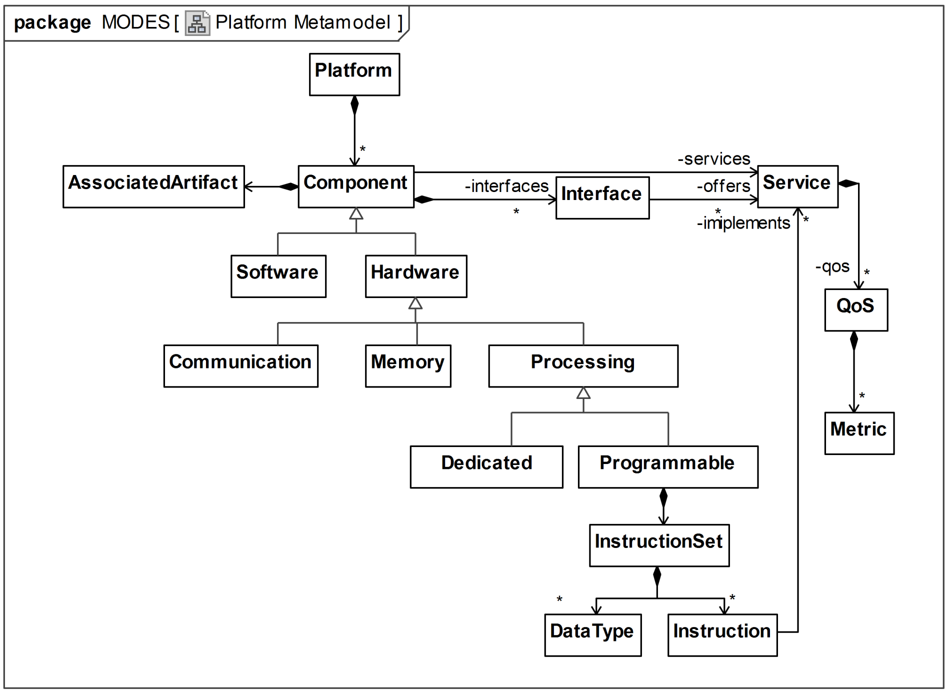
7.2.6 Platform Metamodel

In a PBD context a large number of hardware and software components are provided and can be reused in the system development. To evaluate different solutions at a high abstraction level, the reused components must be pre-characterized in terms of performance, energy, memory footprint, and others. This pre-characterized library dramatically reduces the design phases and the uncertainty about the system properties, thus improving the productivity and accuracy. The software component characterization is performed after the component code is compiled for the target architecture, since at this point in time the evaluation can capture architectural information with high accuracy. The characterization of hardware components must be performed from adequate synthesized descriptions, to obtain values that are independent of technology and frequency, such as execution cycles and gate switchings per cycle (a measure for power consumption).

In the MODES framework, the available hardware and software components, as well as the information on characterization, are stored in a platform model, conforming to the metamodel presented in Figure 7.7. Such a platform model is a repository of reusable components at different abstraction levels, which includes implementations or models. This representation is consistent with the platform definition presented in (SANGIOVANNI-VINCENTELLI, 2007).

In the Platform metamodel, a Platform contains different Components, which offer Services for the application, through a set of Interfaces. These Services must be pre-characterized in terms of Quality of Service, which is represented in the metamodel by the element QoS. Quality of Service has Metrics, which hold the values of a QoS in a specific metric. Two types of components are distinguished, namely Software and Hardware. In order to simplify the metamodel, there is no distinction between different types of software, such as application components, Operating System, and drivers, so that any type of software can be reused from the platform repository. Hardware components are classified in Communication, Memory and Processing, which repre-

Figure 7.7: Platform metamodel.



sent communication, storage and processing resources, respectively. Moreover, Processing can also be Dedicated, so that no software can be executed onto it, or Programmable, which is able to execute software, if it conforms to the Programmable's instruction set.

Currently, the platform repository conforming to the Platform Metamodel contains information on processing units of Femtojava type, which are different versions of a Java micro-controller (ITO; CARRO; JACOBI, 2001; BECK et al., 2003; WEHRMEISTER et al., 2005), scheduling, timer, and real-time specification services/API (WEHRMEISTER et al., 2005) implemented on top of Femtojava micro-controllers, hardware and software components for communication (SILVA et al., 2006, 2008), a math library, and a library of video image processing.

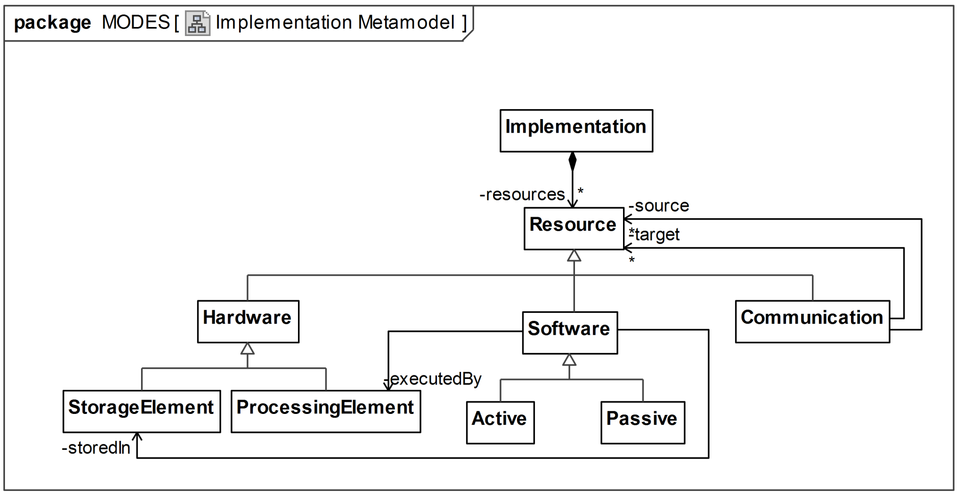
7.2.7 Implementation Metamodel

The Implementation Metamodel is presented in Figure 7.8 and represents the mapping of application into an architecture, including the allocated resources and the mapping between them. An Implementation is composed by a list of Resources, which are the Hardware, Software and Communication components required to implement the system. Software can be Active, which means it is scheduled and can initiate communication, or Passive. The metamodel represents the association between Hardware and Software, namely storedIn and executedBy, and the Communication between resources, namely source and target.

7.2.8 Other Metamodels

MODES provides also other metamodels, which are not used to support the DSE methodology. Some of these metamodels are Interaction Graph and Labeled Timed Automata presented in (NASCIMENTO; OLIVEIRA; WAGNER, 2012). Moreover, MODES

Figure 7.8: Implementation metamodel.



provides metamodels for languages commonly used for design of embedded systems, such as Simulink and SystemC, which are not described here.

The Interaction Graph metamodel is used to represent the application behavior in form of CDFGs, in order to generate code out of it. The Interaction Graph metamodel allows the composition of multiple CDFGs, so that complexity of the application can represented using different abstractions. This metamodel is the basis of the Pseudo-trace metamodel, and in this way they share common elements to represent the behavior. However, the Interaction Graph metamodel does not provide elements to represent run-time information.

MODES can extract a network of Labeled Timed Automata (LTA) (ALUR; DILL, 1994) from UML Sequence Diagrams in order to verify the system functional behavior. The LTA metamodel captures all concepts introduced by the UPPAAL model checking tool (LARSEN; PETTERSSON; YI, 1997). LTA is used in the UPPAAL model checker to perform formal verification of specified properties of the system. This feature is very useful for the designer, since the LTA model can automatically be generated from system specification and helps the designer to debug and validate it.

7.2.9 Model Transformations

A library of transformations is provided by the MODES framework, so that other DSMDETs can integrate different design views into their own internal representations and be more independent of design languages. For example, a scheduling analysis tool would made use of TGFF models and of the transformation from UML to TGFF.

Previously, the library already contained transformations from UML to Component, Interaction Graph, Implementation, and LTA models, and from these models to Java and UPAAL code (NASCIMENTO; OLIVEIRA; WAGNER, 2012). A transformation from UML to Simulink was already integrated in the framework (BRISOLARA et al., 2008). This thesis extended the library by providing transformations from UML to Pseudo-Trace, TGFF, and DSED models. Moreover, an injection transformation from TGFF text files to TGFF models in EMF was implemented, in order to use data in the TGFF format inside of the DSMDET, e.g., when loading graphs for evaluation of the methodology in Chapter 8.

7.3 SPEU: Evaluation Tool

Moving the development focus from code to model (MRAIDHA et al., 2004) suggests the support of a fast design space exploration in the early design steps, where the design effort is now concentrated. Moreover, high abstraction modeling decisions can lead to substantially superior improvements, when compared to design decisions taken at low abstraction levels (MATTOS et al., 2004; THEELEN; PUTTEN; VOETEN, 2004). However, at high abstraction levels software engineers do not have an exact idea of the impact of their decisions on essential issues such as performance, energy, and memory footprint for a given embedded platform. It would be desirable that the designer could evaluate the candidate solutions as early as possible, using the same abstraction level as in the system specification. Furthermore, at a high abstraction level the design space is large and an efficient DSE process requires quick and precise evaluation methods to rank candidate designs. However, accuracy is related to low level and inflexible designs, which are slowly evaluated by simulation tools.

Therefore, a high-level model-based estimation tool supporting a quick DSE process in early design phases is needed. H-SPEX uses the SPEU estimation tool (OLIVEIRA et al., 2006) to evaluate alternative design solutions and to guide DSE. SPEU provides analytical estimates on physical system properties. These properties are directly obtained from system specification in UML, C++, and binary code, which are transformed into CDFGs of Pseudo-Trace and Component models. The Implicit Path Enumeration method (LI; MALIK, 1995) was implemented to find out the worst case execution path in each CDFG, by using an ILP formulation. A Symbolic Instruction Set is used, in order to reuse the pseudo-trace for different architectural configurations, and improves the estimation during automatic DSE. A symbolic instruction model conforming to the ISA metamodel is used to label vertices of the pseudo-trace model. Such symbolic instructions must be mapped into services and instructions of a real platform, so that the costs of the application can be estimated. The estimation relies on a platform containing reusable components, which are characterized in terms of performance, energy, memory footprint, and others. The estimation presents errors as low as 5 % (OLIVEIRA et al., 2006), comparing to results extracted by cycle-accurate simulation, when the reuse of platform repository is largely employed by a PBD approach.

The original SPEU tool was extended, in order to better support the methodology presented in the previous chapters. Different aspects of the tool were improved, such as the mapping of instruction set, platform metamodel, and pre-characterization of platform components. Moreover the estimation method was extended to support multiple processors and communication. Besides UML, the estimation based on other system specification languages is now also supported, by integrating the SPEU method into the MODES framework.

The estimation method implemented in SPEU is divided into three steps, so that the tool can gather the most information possible to evaluate the system. In a preliminary step a platform repository containing information about the components reused in the system must be provided and the services available must be mapped into the Symbolic Instruction Set defined by SPEU. This process is defined in Section 7.3.1. In order to automate the information gathering process, the system must be modeled under certain restrictions. The modeling process was described in Section 5.3. Finally the automated estimation process is described in Section 7.3.2.

7.3.1 Platform Characterization and Instruction Mapping

In order to overcome the drawback created by the absence of the complete behavior description in UML models, and to allow more abstract evaluation, SPEU uses the information specified in the platform repository regarding costs of pre-designed components and include them in the CDFGs. This way, SPEU improves estimation accuracy, and bridges the gap between high abstraction models and platform low level details. However, the use of information about pre-characterized components from a repository by itself does not provide precision enough for DSE purposes, because such information is still dependent on the context that the system is executing. Therefore, SPEU provides a systematic method to identify and reuse information from the platform repository in the system models, by mapping symbolic instructions representing the application into real-instructions and services of platform repository.

The repository model proposed to store the required information conforms to the Platform metamodel, which was presented in Section 7.2.6. It is important to notice that the Platform model contains only (meta)information about a component, and not the component itself, so neither structural information nor behavioral information is stored. For this information one must rely on the artifacts referenced by each component in the Platform model, because if a component is available for reuse then it must have an artifact that defines it.

Providing an extensive component repository requires a significant effort. However, normally a large amount of system components can be reused from different component providers or as a sub-product from a previous system development (SHANDLE; MARTIN, 2002). Therefore, the platform repository creation is based on the accumulation of components produced or acquired in previous product developments. To add new IP resources to the platform repository, the IP provider or the user must attach this architectural information to his/her IPs. In order to provide such information the system can be simulated or the SPEU static analysis can be used on information extracted from a low level specification, instead of UML models. Simulation provides high accuracy, although it requires long interaction cycles. The pre-characterization of components by using the SPEU tool trades-off accuracy for a quick evaluation, which extracts the required information from C++ or binary code transformed in the MODES models, in order to support quick evaluation and addition of components' metadata into the platform library. Figure 7.9 illustrates both alternatives to add characterized components to the platform repository.

After population of the repository with adequate information on the platform components, a mapping is required from the provided services into the Symbolic Instructions, which is a model conforming to the ISA metamodel presented in Section 7.2.4. This method bridges the gap between the architectural independent representation of the application and the actual architecture where the application is deployed. The weaving method presented in Section 5.5 is used to improve the instruction set mapping proposed in (OLIVEIRA et al., 2006). Figure 7.10 illustrates the mapping process and the relationship between the ISA metamodel and ISA models.

The instruction mapping method consists of the creation of a weaving model that contains references from one Symbolic Instruction to one or more real instructions required to implement it, so that SPEU can add costs of all real instructions and annotate them in the pseudo-trace model. The AWM Tool is used to create an ISA Mapping model, which is used during the estimation process to determine the cost of symbolic instructions found in the Pseudo-trace model. Each `Programmable` in the repository must have an ISA

Figure 7.9: Platform Component Characterization.

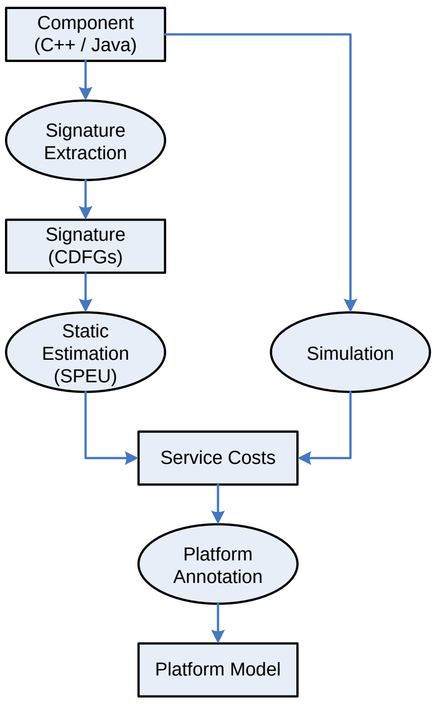
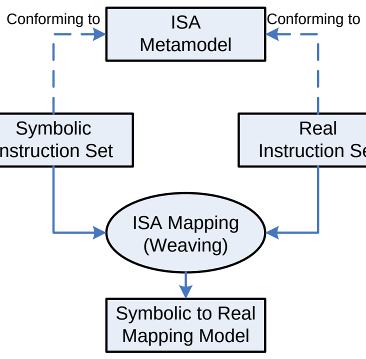


Figure 7.10: Instruction Set Architecture Model Mapping.



Mapping model associated to it.

Table 7.1 illustrates part of the mapping between SPEU's symbolic instructions into real instructions of Femtojava Multicicle. The table shows six symbolic instructions, one per line, which are mapped to one or more real instructions defined, shown in the central column. In order to calculate the final cost of a symbolic instruction, a field `costRule`, shown in the right column, was added by extending the AWM metamodel. This field indicates if the cost calculation must: i) assign directly the cost of a real to a symbolic instruction (`direct`); ii) add all values of the referenced real instructions (`add`); or use arithmetic mean of the values of all referenced real instructions (`mean`).

7.3.2 Estimation process

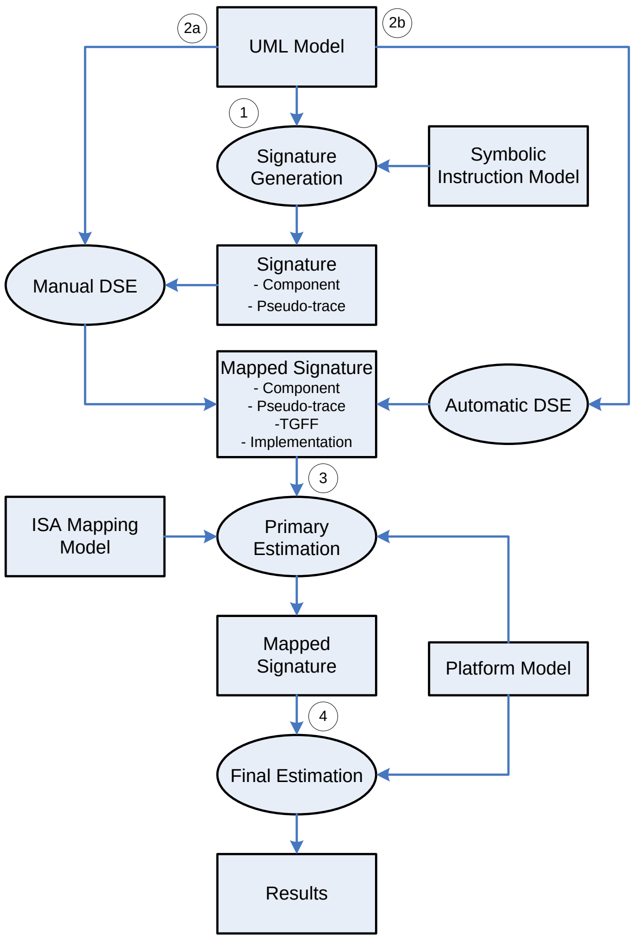
Currently the estimation approach allows the estimation of performance, energy consumption, power dissipation, throughput of communication data, and footprint of data and program memories. These system properties are the basis for other analyses, e.g. quality of service (QoS) assessment on issues such as communication, schedulability, and

Table 7.1: Sample of symbolic to real instructions of Femtojava microcontroller.

Symbolic Instruction	Real Instruction	Cost Rule
`getDynamicObjectField`	`aload_0, getfield`	add
`interactionStatic`	`invokestatic`	direct
`loadParameterByValue`	`iload_0`	direct
`conditional`	`ifgt, ifeq, ifle`	mean
`returnInteger`	`ireturn`	direct
`setStaticPrimitiveField`	`putstatic`	direct

resource usage. After these estimates, further analyses can be performed, as suggested in (PETRIU; WOODSIDE, 2003). Using an approach similar to (SMITH; WILLIAMS, 2003), the estimation focuses on use cases and scenarios that describe the system, since they provide the basics for the object-oriented methodology and provide the context for the system evaluation. The estimation method flow is illustrated in Figure 7.11 and described in the sequence.

Figure 7.11: Estimation Flow of SPEU Tool.



The evaluation flow starts generating the signature of the system, by means of transformations from UML models into Component and Pseudo-trace models conforming to the metamodels described in Section 7.2. This signature represents the structure and behavior of the system, without considering yet a platform mapping. Moreover, the reused platform services are also included in the signature in the form of symbolic instructions, whose costs are extracted from the platform repository after the mapping step. Therefore, this signature can be generated once and reused for multiple mapping solutions.

There are two alternative ways to map a signature. One way is to manually define Component diagrams with allocated components in the UML model, so that such mapping information can be extracted. This alternative is identified in Figure 7.11 as the process 2.a (Manual DSE). Another way is by using an automatic DSE tool, which automatically generates the mapping information. The automatic process is identified in Figure 7.11 as the process 2.b (Automatic DSE). The mapping is responsible for defining the allocation of platform resources and the mapping of the application into an architecture. A Task Graph and an Implementation model are added into the previously generated signature, which after this step contains also architectural decisions required for the system evaluation.

In the Primary Estimation step the first figures are gathered from the Platform model and related ISA Mapping models, which contain the link between symbolic and real instructions, in order to annotate them as intermediate values in the vertices and edges of each `InteractionGraph` from the Pseudo-trace model. After this first process, the SPEU tool can estimate the best case and worst case scenarios by handling the estimation as an optimization problem. For this purpose, SPEU formulates an ILP for each `InteractionGraph` in the the Pseudo-trace model, in order to find the worst case execution path, by using a method similar to the Implicit Path Enumeration(LI; MALIK, 1995), and based on the these paths SPEU back annotates the costs in each `InteractionGraph`. Implicit Path Enumeration determines the number of executions of each basic block, which in the SPEU tool are symbolic instructions, in the best or worst cases. These limits are calculated by minimizing or maximizing the linear expression in Equation 7.1. The result cost C is the value annotated in each graph.

$$C = \sum_{i=0}^{N} c_i x_i \tag{7.1}$$

Where:

c_i is the cost associated to the mapped symbolic instruction $inst_i$.

x_i is the number of times $inst_i$ is executed, such that $x_i \in \mathbb{Z}^+$

N is the number of symbolic instructions in an `InteractionGraph`.

Linear constraints are used to give a hint about the execution paths in `InteractionGraphs` and can represent structural and functional constraints. Structural constraints are automatically extracted from the graph structure and represent loop and conditional executions. The functional restrictions are extracted from additional information specified in the UML model. Such information are lower and upper bounds specified in the `loop` operators and restrictions on conditional execution paths specified in `alt` operators in UML Sequence or Interaction Overview diagrams. These constraints are transformed into linear constraints, by using the Java LP wrapper[4] to the LP Solver library[5], when an ILP problem is formulated and solved. The hierarchical representation of CDFG in the

[4]http://lpsolve.sourceforge.net/5.5/

[5]http://lpsolve.sourceforge.net/5.5/

Pseudo-trace allows for complexity management, so that the ILP problems formulated for CDFG at the bottom of the hierarchy are solved first, whose costs are annotated in reference vertices of CDFG at the higher levels of the hierarchy, until reaching the top-level graph that represents the Interaction Overview diagram.

Figure 7.12 illustrates the transformation from UML Sequence Diagram into CDFG for the Pseudo-trace model and the implicit path enumeration method. Figure 7.13(a) shows a Sequence diagram, which contains an execution specification of a `Navigator`. The execution is triggered by a `Scheduler`, than in a `loop` fragment the `Navigator` reads the data from `CollisionAvoidance`. An `alt` fragment is used to specify alternative executions. One alternative calls the `notify` method to send angle and speed values to the `MovementControler`. Another alternative makes reference to an `emergencyStop` scenario, which has its own specification in another diagram. A CDFG is extracted from the Sequence diagrams and transformed into an `Interaction-Graph` of a Pseudo-trace model. The transformation iterates on each UML `Collabora-tion` in order to find the Sequence diagrams, from which CDFGs are generated. Afterwards, each Sequence diagram is iterated, such that the UML `InteractionFragment` elements, such as messages, `loop`, and `alt`, found in the model are used to generate vertices of the CDFG. For each vertex generated, a variable $(x_1, ..., x_n)$, associated to the vertex by a labeling function, is created in the ILP formulation to hold the number of times the instructions is executed. Different `InteractionFragment` are specially handled, in order to map adequately the fragments to Symbolic Instruction represent by vertices. The resulting CDFG for the presented Sequence diagram is shown in Figure 7.13(b). In this graph, the labels in the vertices identify the variable associated to it, as required in the Equation 7.1, and the labels in the edge are also associated to variables used to propagate the linear constraints used in the implicit path enumeration method. In this graphs some nodes are associated to labels in order to identify the respective UML element in Figure 7.13(a).

Each CDFG must have start and finish vertices, which are used to bound the execution of the evaluated scenario, so that, e.g., if it is entered five times in one scenario and x_1 represents the start vertex of this scenario, then $x_1 = 5$. Equations 7.2 to 7.5 are examples of linear constraints. Equation 7.2 defines that the scenario must execute five times, hence the edge $d_1 = x_1$ in order to propagate constraints. Equation 7.3 defines that the first and last vertices must be executed the same number of times. For all vertices must hold that the sum of values from its incoming edges must be equal to the sum of the values from its outgoing edges, as illustrated in Equation 7.4. Equation 7.5 illustrates a functional constraint, so that the execution enters in the loop, at vertex x_3, at least the same number of times its previous vertex executes and at most 10 times the number of executions of the previous vertex. In this case, x_4 will execute 50 times inside the loop.

$$x_1 = 5 \textit{ and } x_1 = d_1 \tag{7.2}$$

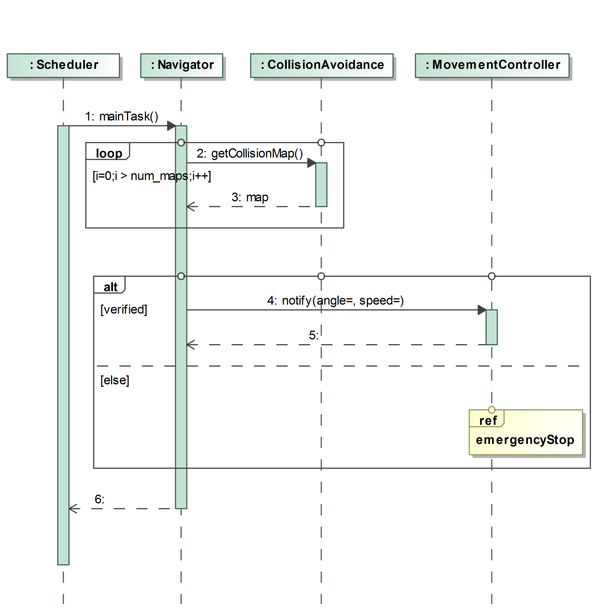
$$x_1 = x_9 \tag{7.3}$$

$$x_5 = d_5 = d_6 + d_7 \tag{7.4}$$

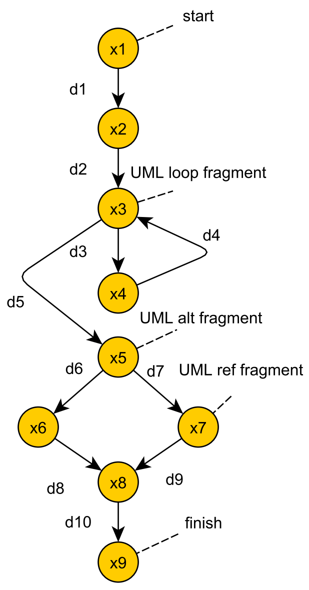
$$x_2 \leq x_3 \leq 10x_2 \tag{7.5}$$

In the Final Estimation step, SPEU calculates additional costs based on task scheduling and communication between tasks, considering the task mapping and the allocation of processors in a communication structure.

Figure 7.12: Example of Implicit Path Enumeration from UML Sequence Diagram: (a) Sequence diagram; (b) Resulting CDFG.



(a)



(b)

7.4 H-SPEX: Design Space Exploration Tool

A prototype tool named H-SPEX was implemented in order to support the automatic DSE methodology. This prototype integrates the MODES framework, the SPEU tool, and the FORMULA engine to solve DSED problems in a general way. Into H-SPEX was also implemented the CPACO-MO algorithm to solve the DSE CGP Mapping Problem. Moreover, H-SPEX coordinates the transformation between required models. The implementation is based on the OAT framework[6], which provides a graphical user interface to configure and manage optimization runs. This framework provides also a set components that make easier the development and experimentation of new optimization algorithms, such as operators, monitor, fitness functions, and graphical result presentation. OAT provides also interfaces based on the Domain-Problem-Algorithm design pattern, which splits the structures of an optimization problem, and allows easy framework extensions.

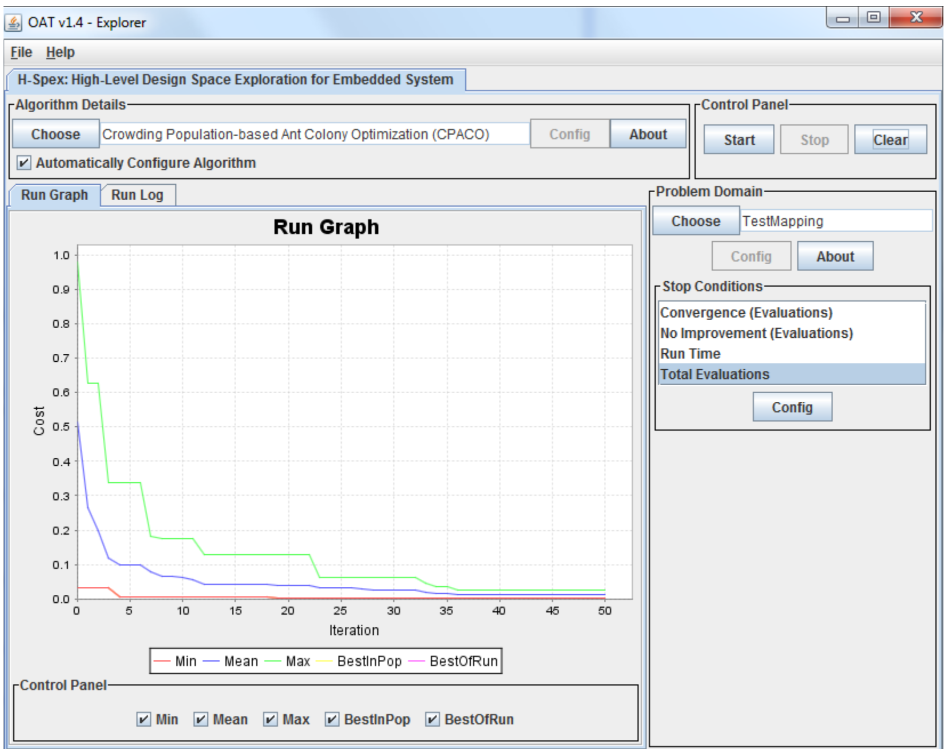
By using that design pattern, the EMF-based classes generated from the DSED metamodel were integrated into the OAT framework. A `DSEDomain` class implements the OAT's Domain interface, which collects information about the available algorithms, problems and solvers for a specific domain. The `DSEDProblem` implements the OAT's Problem interface, which provides information about the problem to be solved, such as how it must be evaluated and validated. Figure 7.13 shows the H-SPEX tool graphical user interface implemented by using the OAT framework.

7.4.1 Evaluator Integration

A proxy class was implemented to use the SPEU tool as evaluator and integrate it into the OAT framework, which calls SPEU using the `Problem` interface. The SPEU tool

[6]http://optalgtoolkit.sourceforge.net/

Figure 7.13: H-SPEX GUI implemented by using the OAT Framework.



Algorithm Details: Crowding Population-based Ant Colony Optimization (CPACO)

☑ Automatically Configure Algorithm

Problem Domain: TestMapping

Control Panel: ☑ Min ☑ Mean ☑ Max ☑ BestInPop ☑ BestOfRun

provides estimates for energy or power consumption, communication bandwidth, memory footprint, performance (execution time or cycles), or any combination of the above. The information obtained with SPEU are stored in edges, vertices and graph elements of the DSED model, which any solver algorithm can analyze and use it in its objective function. If additional metrics are need, one can implement such a proxy class and easily add another evaluator in the framework, by specifying the new solver in the DSED model with its parameters, and the fully qualified Java name (e.g. `br.ufrgs.inf.Randon-Evaluator`) to find the classes that implement the new evaluator.

7.4.2 Solver Integration

A class implementing the CPACO-MO algorithm, described in Section 6.4.1 implements also the OAT's interface `Algorithm`, and is used to solve DSE CGP Mapping Problems defined in a DSED model. In this problem, since the design space is represented as a CGP and the optimization problem always consists in finding a (sub)optimal sub-graph of the resulting CGP, the optimization algorithm is not aware of specific DSE information and of the semantics of vertices and edges in the design space. This means that the optimization algorithm is detached from the design space and from the specific DSE problem to be solved, thus it does not require a specific optimization approach. In this way, one could adopt other multi-objective optimization algorithm, such as the ones provided by the OAT framework.

The `Algorithm` interface from the OAT framework requires the implementation of a solution generation function. This function was implemented according to Algorithm 6.4 and it integrates the VIATRA II[7] transformation engine in the OAT framework. VIATRA II was used to execute the DSE rules inside the solution generation function

[7]http://www.eclipse.org/viatra2/

for DSE CGP Mapping Problems, so that after the transformation a solution is generated and made available to the optimization algorithm. This transformation engine is easily integrated into Java programs, and allows calling Java methods from the inside its transformation engine. It also provides different styles of transformation languages, including declarative and imperative rules. Such integration and flexibility were the main factors for selecting this transformation engine. Besides, experiments have shown that VIATRA II presents good performance, although it is outperformed by other transformation engines(HORVáTH et al., 2010; BERGMANN et al., 2008).

7.4.3 Library of DSE Rules

In order to alleviate the effort to implement DSE rules manually, a library of rules to support the constraints defined in the DSED metamodel must be implemented. In general, these rules verify the existence of elements of a specific `Constraint` type in the DSED model, such as `MaximumValue`, `DuplicatedMapping` and `MandatoryMapping` to verify if it is applicable to the DSE scenario in analysis. Afterwards, for each constraint found the rules create or remove elements according to the constraint semantics, which are described in Section 5.4.

There are two levels of reuse of resource in this library, namely user and developer levels. The former consists on the definition of modeling elements available in the DSED metamodel, for example by creating an `IncludeMapping` element and setting the vertices that must be mapped in the `explorables` attribute of `IncludeMapping` element. This mechanisms assumes that there is an implementation of solvers and DSE Rules that support such constraints. This is a very flexible mechanism to support reuse of DSE Rules implemented for different solvers and technologies. By extending a library and the metamodel, such mechanism can support an arbitrary number of DSE Rules and solvers, reusing a large amount of artifacts and alleviating the user effort.

The latter consists of the direct reuse of the code implemented in the concrete syntax used to define the DSE Rules, inside new transformation developed by the tool user. On this level the reuse is dependent of the implementation issues and requires a developer role from the part of tool user. The DSE Rules are implemented in the H-SPEX tool by using VIATRA II. In this language the reusable constructs are divided into Abstract State Machine (ASM) rules (`rule`), graph transformation rules (`gtrule`) and graph patterns (`patterns`). ASM rules use an imperative style to manipulate models, whereas graph transformation rules use a declarative one. Graph transformation rules have preconditions, postconditions and an optional action body, which allows for imperative constructions. Graph patterns are declarative model queries, which define pattens to be matched in the model. Such patterns are used by ASM and graph transformation rules to define which elements must be manipulated.

H-SPEX provides a library of reusable Graph transformation rules and graph patterns, which support some of the constrains defined in Section 5.4, and they support both levels of reuse. In order to reuse resources at the developer level, developers must create their own transformation to manipulate the DSED model and use the fully qualified name of rules or pattern to invoke them. For example, by using use `hspex.library.viatra.successorListOf(Dg,V,VV)`, a developer can define a call to use the `successor-ListOf(Dg,V,VV)` graph pattern to get a list of successor vertices of a specific vertex in a specific graph. The prefix `hspex.library.viatra` used before each rule and pattern name to form the fully qualified name where they are found. As an example of resources contained in the library Listing 7.1 shows the `successorListOf(Dg,V,VV)`

Listing 7.1: Sample of Graph Pattern in VIATRA II: Successor list of a vertex

```
1  shareable pattern successorListOf(Dg,V,VV) = {
2    'Graph'(Dg);
3    'Vertex'(V);
4    'Graph'.vertices(Has, Dg, V);
5    find outgoingEdge(Dg,V,E);
6    'Edge'(E);
7    'Edge'.target(TGR, E, VV);
8    'Vertex'(VV);
9  }
```

graph pattern, and Listing 7.2 shows an implementation for the `IncludeMapping` constraint element defined in the Equation 5.21.

The graph pattern shown in Listing 7.1 is used to iterate on design graphs and build the logical CGP, when solving the CGP Mapping Problem. It starts by identifying the types passed as parameters: a design graph Dg and a vertex V in Lines 2-3. Line 4 matches the composition relation `vertices` of a `Graph`, such that the vertex V is in $Dg\,(V)$. It also reuses another graph pattern, named `outgoingEdge`, identified in Line 5 and called by using the `find` keyword. The `outgoingEdge` matches the outgoing edges of vertex V and by matching the target relation between each edge and its target vertex VV in Line 7, the successor vertex VV is returned as result of the application of the complete pattern.

Listing 7.2: Sample of DSE rules: Include Mapping

```
1  gtrule createIncludedMapping(in Cs, out Sl) = {
2    precondition pattern mappingToInclude(Cs, U, V, Sl) = {
3      'IncludeMapping'(Cs);
4      'DSESolution'(Sl);
5      'Explorable'(U);
6      'Explorable'(V);
7      find explorableToInclude(Cs, U, V);
8      neg find includedMapping(Cs, NoMp, U, V);
9    }
10   postcondition pattern includedMapping(U, V, Sl, Mp) = {
11     'DSESolution'(Sl);
12     'MappingDecision'(Mp);
13     'DSESolution'.decisions(Mps, Sl, Mp);
14     find mappedExplorable(Mp, U, V);
15   }
16   action {
17     move(D, ref(fqn(Sl)) );
18     rename(D, name(Cs) );
19   }
20 }
```

A graph transformation rule is shown in Listing 7.2. This rule implements the `IncludeMapping` constraint, according to Equation 5.21. It is called at the beginning of the generation process, so that the generation of a design candidate is influenced by previous design decisions made by engineers when defining a DSE scenario. This rule defines a precondition pattern named `mappingToInclude` in Line 2. In Lines 3-6 this pattern identifies the pattern parameters: an `IncludeMapping` constraint Cs; the vertices to be mapped, U and V; the solution Sl in which the mapping decision Mp must be included and which was previously created. The precondition pattern reuses in Line 7 a graph pattern named `explorableToInclude`, which matches the vertices defined in the `explorables` attribute of the `IncludeMapping` constraint Cs. Then a negative pattern match is used in Line 8 to avoid creating the same mapping twice. A postcondition pattern, named `includedMapping` is used to defined the match in the model with the mapping to be crated. It receives as input the vertices U and V found by the precondition pattern, the solution Sl and the `MappingDecision` Mp to be created with the mapped vertices. Lines 11-12 define the inputs used in Line 13, which matches the relation between the solution Sl the decision D to be created. The pattern `mappedExplorable` defined in Line 14 matches the mapping decision Mp and the mapped vertices U, and V. If the preconditions holds, the VIATRA transformation engine manipulates the DSED model, so that the postcondition holds after the rule application. Finally, the `action` body in Lines 16-18 moves the created mapping decision to the solution and renames it to have the same name as the constraint Cs.

Currently the library does not provide DSE Rules to support all constraints. As the focus of this thesis is the mapping for PBD, only the constraints defined for the DSED Core, DSED Mapping Problem, and the constraints `Deadline` `MaximumOccupation` from the Scheduling Problem were implemented. Such an implementation supported the experiments in Chapter 8 and illustrates the methodology. Additional DSE Rules can be included in the existing library, if they are specified in VIATRA II, or a complete new library can be implemented by reusing the available patterns and rules. Alternatively, new rules in any other language can be implemented, since it supports the DSED metamodel, which is currently defined in ECORE. The DSED model provides many elements to support a broad spectrum of DSE scenarios and a developer can refer directly to them, so that rules can be implemented independently of design models.

7.5 Automatic DSED Generation From UML

The transformation of UML models into DSED is aimed to support an engineer in the task of producing DSED scenarios. The automation of such a transformation relies on transformations and metamodels provided by the MODES framework and additional transformations that extract constraint information from the UML model. The process starts by transforming UML models into Component, TGFF, InteractionGraph, and Implementation models, from which a transformation extracts design graphs and write them into a DSED model. Furthermore, the generator creates different `Constraints` elements according to patterns found in UML/MARTE model. Because some generated constraints have an implementation in the library, an engineer is not required to manually specify those DSE rules, hence improving the productivity by reusing design artifact to generate DSE ones.

In order to generate constraints from UML/MARTE constructs a mapping between patterns in the UML model to DSED one was defined. This mapping was implemented by using ATL transformations, which search for patterns in the UML/MARTE model and create the respective constraints in the DSED model. Such constraints are associated to DSED elements which provide the context where the constraints must be applied. The context is also found in the UML/MARTE pattern, by identifying the elements to which the stereotypes are applied, such as classes and objects. In this way, after creating a constraint, it is associated to `Vertex`, `Edge`, `Decisions` and other DSED elements, which will hold the values used in the DSE Rule. A simplified view of the UML/MARTE patterns and the mapping to DSED constraints is presented in Table 7.3.

The «Nfp» stereotype is used to annotate a `Property` of a UML `Class` and define a specific constraint on the annotated property. Three variations of this pattern generate the constraints `MaximumValue`, `MinimumValule`, and `AssignedValue`, which are distinguished by using the stereotype properties $staQ$ equal to min,max or $determ$, respectively, where x assigned to the stereotype property $value$ is a value speci-

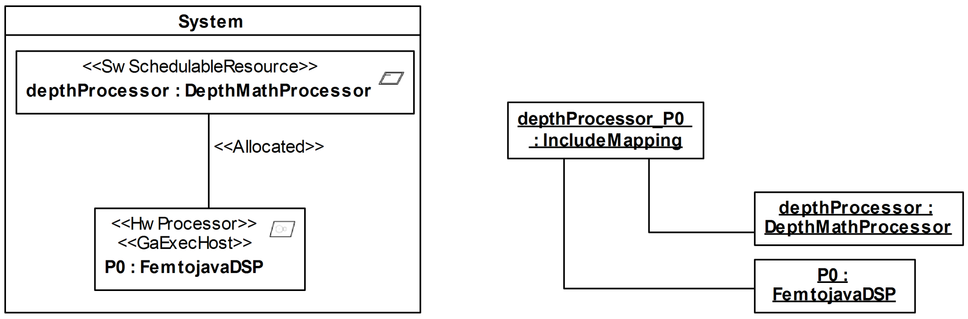
Table 7.3: Mapping of MARTE Profile to DSE rules.

UML/MARTE	DSE rule
`Property` + «Nfp» + $source = req$, $staQ = max$, $value = x$, $unit = y$	`MaximumValue`
`Property` + «Nfp» + $source = req$, $staQ = min$, $value = x$, $unit = y$	`MinimumValue`
`Property` + «Nfp» + $source = req$, $staQ = determ$, $value = x$, $unit = y$	`AssignedValue`
«HwProcessor» + $frequency = x$, $unit = y$, $source = req$	`PropertyValue`
«SwSchedulableResource» + «HwClock» + $frequency = x$, $unit = y$, $source = req$	`PropertyValue`
«HwProcessor» + «GaExecHost»+ $source = req$, $staQ = max$, $utilization = x$	`MaximumOccupation`
«SwSchedulableResource» + «Allocated» + «HwProcessor»	`IncludeMapping`
«HwProcessor» + «HwBus» + Port Connection	`IncludeMapping`
«SwSchedulableResource» + «RtFeature» + «RtSpecification» + $relDl.value = x$, $unit = y$	`Deadline`

fied by the user. The annotated `Property` must be one of the `NFP_Real` type provided in the MARTE profile library, such as `NFP_Power`, `NFP_Area` and others types, or a `NFP_CommonType` without defining a *unit* property, such as `NFP_Boolean` and `NFP_Integer`. The stereotype «Nfp» is required to specify performance, power, memory, and other NFRs that is used to define constraints in the DSED model and trigger DSE Rules to remove design candidates, if they do not fulfill the NFRs or to use such values in some heuristic search implemented by a solver. A `AssignedValue` constraint is generated to define the frequency property of a processor, when the stereotype «HwProcessor» and its $frequency$ property are found in the UML model. Another configuration constraint of type `AssignedValue` is generated for elements annotated with the stereotype «SwSchedulableReseource», which defines a task, and «HwClock» with a $frequency$ property defined. In this case, the processing unit that is going to execute the task must be configured with the specified frequency. A `MaximumOccupation` constraint is created when elements annotated with the stereotype «GaExecHost» and its properties $source = req$, $staQ = max$, and $utilization = x$ are found in combination with the stereotype «HwProcessor». The constraint `IncludedMapping` is created when elements annotated with «SwSchedulableReseource» are associated to elements annotated with «HwProcessor» and the association is annotated with the «Allocated» stereotype, which defines that a task must be executed in a specific processor. The mapping of a processor to a specific bus segment is defined by connecting a port of a processor annotated with the stereotype «HwProcessor» to a port of an element annotated with the stereotype «HwBus». This pattern creates an `IncludeMapping` constraint, mapping these elements. Finally, in order to verify if a task fulfills the `Deadline` constraint, an element representing this task must be annotated with the stereotype «SwSchedulableResource» and have `Message`associated to a task's `Operation` and it is annotated with the stereotype «RtFeature». The «RtFeature» must be associated to a «RtSpecification», whose property `relDl` is assigned with the relative deadline.

As an example, lets consider a system with a task named `depthProcessor`, which implements a heavy image processing function. An engineer may want to constrain the DSE by defining that this task must execute in a DSP microcontroler, which is a hardware resource more adequate to this function. Hence, he/she defies a UML Composite diagram, in which the `depthProcessor` task is annotated the stereotype «SwSchedulableResource» and associated to the processor `P0` annotated with the stereotype «HwProcessor». This association is also annotated to define association semantic, by applying the stereotype «Allocated». Figure 7.15(a) illustrates the resulting UML Composite diagram. Such model patterns leads to the generation of a constraint in the DSED model of type `IncludeMapping`, which is associated to two `Vertex` elements that represents the UML elements `depthProcessor` and `P0`. The DSE Rule previously presented in the Listing 7.2 is applied, when the instance of `IncludeMapping` is found in the DSED model.

Figure 7.14: Pattern for generation of IncludeMapping constraint: (a) UML Composite diagram; (b) Created elements in the DSED model (shown using UML Instances notation).


System
<<Sw SchedulableResource>>
depthProcessor : DepthMathProcessor
<<Allocated>>
<<Hw Processor>>
<<GaExecHost>>
P0 : FemtojavaDSP
depthProcessor_P0 : IncludeMapping
depthProcessor : DepthMathProcessor
P0 : FemtojavaDSP

(a) (b)

The automatic generation of the DSED model can be improved by defining new patterns UML/MARTE and the respective transformations into DSED model, for example, by considering the MARTE data type package to generate configuration constraints. Maybe not all DSED elements can be generated from UML/MARTE models, hence additional standard profiles such as the UML Profile for System on a Chip[8] or private profiles, such as the UML COMPLEX profile for DSE(GRUTTNER et al., 2012). However, due to the expressiveness of MARTE and other standard profiles, this thesis abstained to define yet another profile to cover the generation of some simple elements, such as `MandatoryMapping` that do not require any association with other elements.

[8]http://www.omg.org/spec/SoCP/

7.6 Discussion

This chapter presented the tools adopted to implement and support the proposed methodology. *De facto* standard tools from the Eclipse Modeling Project were used as MDE Framework, such as EMF, ATL and AMW. Different activities were supported by well adopted tools such as Magic Draw for UML modeling, ANTLR for parser generation, and LP Solve to solve ILPs during the estimation process.

Moreover, some tools were extended to fulfill the methodology requirements. The MODES frameworks was extended with more metamodels and transformations, which improved the support of DSMDETs. The methods implemented in the SPEU tool for quick estimation were improved by updating its metamodels, such as platform, ISA, pseudo-trace metamodels. New technologies were also integrated in the SPEU tool that improved the mapping between Symbolic Instruction Set and real ISAs models

A DSE tools named H-SPEX was implemented to integrate solvers, evaluators and the DSED model, so that DSE can be automatically performed. It orchestrates the DSE process and provides a GUI based on the OAT framework, in order to ease the user interaction. It was also implemented a library of DSE rules that supports some of the constraints defined in the DSED metamodel. Finally, a tool to automatically generate DSED models from UML/MARTE was developed that increases the productivity when repetitive DSE scenarios must be evaluated.

Although the DSE methodology flow is not fully automated for all DSE problems and different use cases, many steps were automated and computational support was provided to all steps, in order to ease the integration of the methodology into development process and improve productivity.

8 EVALUATION

This chapter presents an evaluation and an example of the methodology presented in Chapters 5, 6, and 7. It starts by presenting a case study, in which synthetic graphs are used to produce input data with different complexities, so that the step-wise search induced by iterating on the CGP of the design graphs can be evaluated. In a second case study a realistic DSE scenario for a real-life application is used to provide a complete example of the DSE flow presented in Chapter 5. At the end of this chapter final remarks are presented.

8.1 Case Study I: Synthetic Graphs

This case study aims to evaluate the scalability of the CGP method to represent the design space when solving the DSE mapping problem. This method induces a step-wise iteration on the CGP of the design graphs. Therefore, the execution time of Algorithm 6.4, adopting a logical implementation of the CGP, was measured under different scenarios, in order to evaluate how the execution time grows when the problem size increases. The proposed method provides interfaces for iteration, constraint specification and node selection, so that a user can customize the heuristic applied during the search on the design space. Due to this flexibility, only the design space iteration, with random search selection of nodes and minimal constraints is evaluated. In this way, the method behavior is stressed, without the influence of optimization methods, which can be customized by the user.

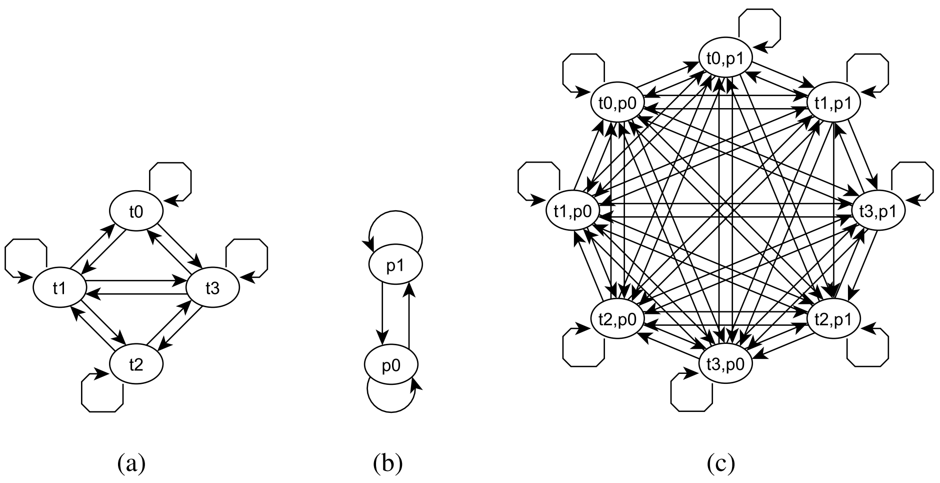
The behavior of Algorithm 6.4 depends on three factors: the size of the graphs, the connectivity between vertices, which increases the number of alternatives, and the number of graphs used in the CGP. In order to evaluate how these factors influence the method, three different experiments were built. For this experiment, synthetic graphs were generated by a Java application especially implemented for this purpose. These graphs are generated in TGFF models, which are transformed into DSED models by an ATL transformation, so that the evaluated method can use the same type of input as in a real DSE scenario. Moreover, graphs extracted from a benchmark in TGFF file format were also adopted in this case. All graphs are explained in each experiment. The transformations and the evaluated method ran on a Java virtual machine 1.6. The machine used in this case study has the following setup: Intel Xeon CPU with 2 cores at 2.40 GHz; Windows 7 Professional 64 Bit; 6 Gb installed RAM. In order to reduce the noise in the measurements, which ran concurrently with other tasks in the operating system, the average time in microseconds to generate 1000 solutions for all generated problems was measured.

8.1.1 Experiment 1: Application size

By increasing the size of the graph, which represents an application also the number of vertices to be mapped onto an architecture increases. These vertices can represent functions, tasks, objects or any other application units. In this experiment the application size is measured in number of tasks, which must be mapped to an architecture. Three cases were evaluated, namely worst case, best case and average case, the last one represented by a benchmark. These cases consist in the generation of candidate solutions for DSE mapping problems formed by pairs of design graphs T and P, such that T represents a task graph and P represents a processor graph onto which tasks must be mapped. During the generation of solutions Constraints 5.20 and 5.18 must be fulfilled, so that all vertices of graph T are mapped and a design decision $d \in S$ does not repeat in the solution S. The experiment generated 26 problems for the worst and best cases, in such a way that the first two measurements were done with 5 and 10 tasks and the following measurements were increased by 10 tasks up to 250 tasks. The number of vertices in P was arbitrarily fixed to 2, in order to control the factors that influence the complexity.

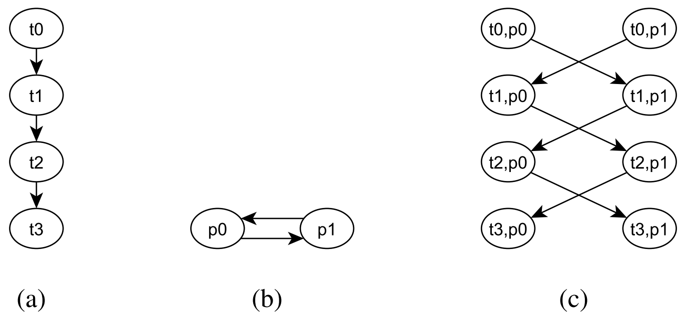
The worst case reproduces a scenario where design graphs are used to produce a CGP with the worst complexity. This means that the resulting CGP has many edges, enforcing the evaluation of many alternatives and the pruning of all already selected vertices after each iteration. Such scenario can be reproduced when the graphs T and P are fully connected, so that $T \otimes P$ is also fully connected. Figure 8.1 illustrates these graphs for the DSE mapping problem in the worst case, consisting of a task graph T with 4 vertices, a processor graph P with 2 vertices and the graph $T \otimes P$.

Figure 8.1: Example of Graphs used for evaluation of the worst case: (a) Task Graph T; (b) Processor Graph P; (c) Design Space $T \otimes P$;



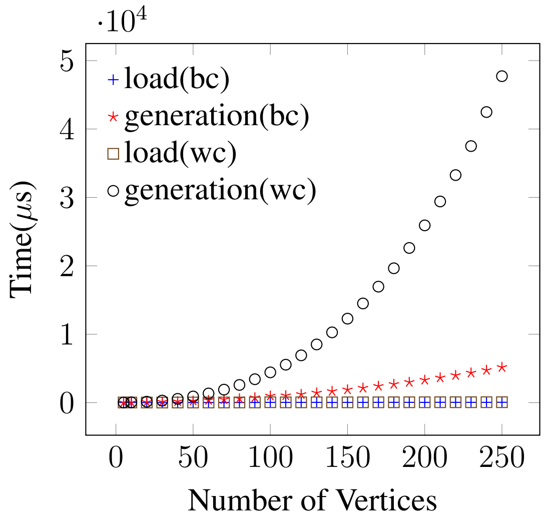
The best case reproduces a scenario where the task graph has a small number of connections and there are not many alternatives for mapping, which leads to a straightforward iteration on the CGP. In order to produce this scenario, the generated task graphs T contain vertices with a maximum degree of 1, resulting in a chain of vertices. The processor graph, which contains 2 processors, contains edges to connect one processor to another in both directions, but without edges connecting a vertex with itself. In this way, the graph $T \otimes P$ enforces that each task must be mapped onto a processor that is different from the previous one, and at a specific vertex there is only one alternative design decision available. Figure 8.2 illustrates sample graphs used to evaluate the best case scenario containing a task graph T with 4 vertices, graph P with 2 vertices, and the graph $T \otimes P$. The resulting graph $T \otimes P$ was explicitly drawn by crossing edges between vertices, in order to highlight the interchange of processors. However, notice that it is a graph with 2 components.

Figure 8.2: Example of Graphs used for evaluation of the best case: (a) Task Graph T; (b) Processor Graph P; (c) Design Space $T \otimes P$;



The results measured for the worst case (wc) and best case (bc) are shown in Figure 8.3. This figure shows the measured average time in microseconds to load a DSED model for each problem containing task graphs with increasing number of vertices and the average time to generate a sub-graph from $T \otimes P$ that represents a candidate solution.

Figure 8.3: Evaluation of solution generation time by increasing the application size (number of vertices)

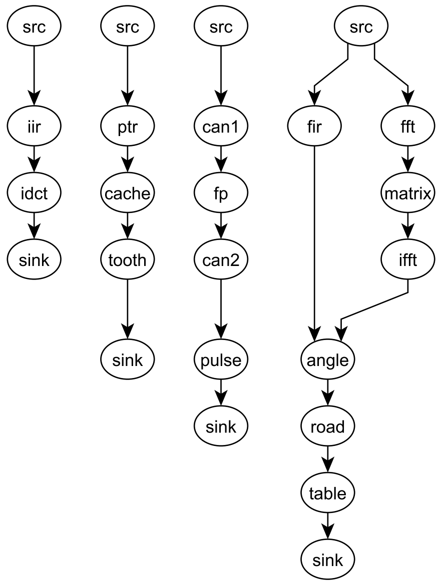


The results shows that the time required to load a DSED model grows linearly, and does not contribute significantly to the total execution of the algorithm. The smallest and the largest load times are 23 and 69 microseconds, respectively. Moreover, the difference between load times for worst and best cases are insignificant. However, the growth of the generation time for worst and best cases diverges significantly as the number of vertices in the task graph increases. Such a growth divergence is due to the difference on the degree of vertices, which increases with the number of vertices in the worst case, whereas it remains constant in the best case. Notice that the high connectivity of the graph $T \otimes P$ for the worst case is not due to the number of alternatives, because for all problems one task can only be mapped onto processor p_0 or p_1. Instead, the connectivity reflects the high dependency between vertices in the task graph, and, as it is shown in the third case, such a design is not usual in real-life applications.

The third case reproduces real-life applications and does not stress the method limits. It represents the average case, where independent sets of tasks with limited number of connections are mapped onto an architecture, reflecting the nature of embedded systems. Such task sets are provided by the Embedded System Synthesis Benchmarks Suite (E3S)[1], which contains task graphs representing real-life applications from different domains, and resource sets containing meta-information about memories, buses, and processors. The E3S benchmark suite was created to aid the evaluation of embedded system synthesis methods and is largely based on the data extracted from the Embedded Microprocessor Benchmark Consortium (EEMBC)[2].

The average case consists of five DSE mapping problems, one for each application domain provided by EEMBC: automotive/industrial automation, consumer electronics, office automation, networking, and telecommunications. Each problem has one task set with one or more task graphs, in a way that the resulting CGP contains multiple components, and stresses additional features of the method, such as jumping from one component to another by using the root and sink lists, as discussed in Section 6.5. Considering high-connected architectures of today, such as those based on NoCs, task sets were mapped onto a fully connected processor graph P. However, this graph contains only two vertices, so that the results can be compared to the results of previous cases. Figure 8.4 illustrates the task set for the automotive/industrial automation application provided by E3S. The others graph illustrations can be found in Appendix 10, together with the list of 45 tasks that compose the benchmark.

Figure 8.4: E3S task graph set representing an automotive application.



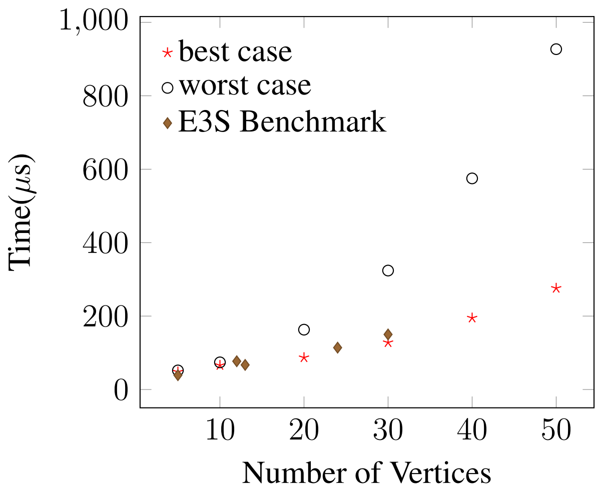
The average total time required to load and generate solutions for DSE mapping problems using the task graphs provided by the E3S benchmark are shown in Figure 8.5.

[1]http://ziyang.eecs.umich.edu/ dickrp/e3s/

[2]http://www.eembc.org/

Besides the average case, the figure also shows the average total time to load and generate solutions measured in the previous experiment for worst and best cases.

Figure 8.5: Evaluation of solution generation time by using the E3S Benchmark.



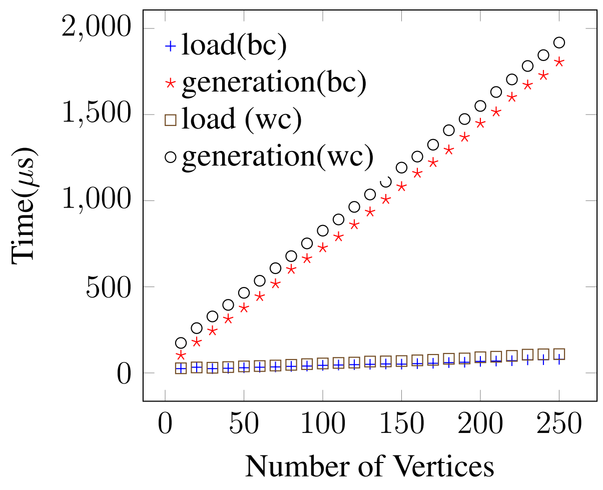
The results shows that for task graphs with small number of vertices, the difference between the execution time of all cases is not significant, whereas from graphs with more than 20 vertices the difference between best and worst cases grows. However, the average total time to generate solutions for the application with 24 and 30 tasks is very close to the best case. Therefore, the time to generate solutions from real-life problems will grow polynomially and closer to the best case than to the worst case, because in real-life problems the growth of the number of vertices does not imply a growth of the degree. Such a disconnection between task graph size and connectivity comes from the fact that reducing the coupling between application components and increasing the coherence of components is a best practice in system development(OLIVEIRA et al., 2008).

8.1.2 Experiment 2: Number of alternatives

The number of alternatives depends on the size and flexibility of an architecture. The size of the architecture is defined by the number of vertices in the architectural graph, and the number of edges defines its flexibility. In order to evaluate the scalability while increasing the number of alternatives in this experiment setup, two sets of 25 problems for worst and best cases were produced. In both cases the task graph T is mapped onto a fully connected architectural graph containing a vertex set varying from 10 up to 250, with step of 10 vertices. A fully connected task graph for the worst case and a task graph containing vertices with maximum degree 1 for the best case were defined and the size of the vertex set was arbitrarily fixed in 20. In this way, the number of alternatives increases with the number of vertices in the architectural graph. Figure 8.6 shows the average time to load and generate solutions for the DSE mapping problem produced in this experiment.

Like other experiments, the time to load a DSED model was not significant. However, an important result is that the time to generate solutions in both cases grows linearly when the number of alternatives increases. Because the iteration on the design space is bounded by the size of the task graph, i.e., it does not matter how large is the architecture vertex set, the algorithm will iterate only $|T(V)|$ times and the observed growth is due to the evaluation of adjacent vertices of the latest selected vertex in the design space. Therefore, the number of alternatives grows with n, where n is number of vertices added in P, when

Figure 8.6: Evaluation of solution generation time by increasing the number of alternatives.



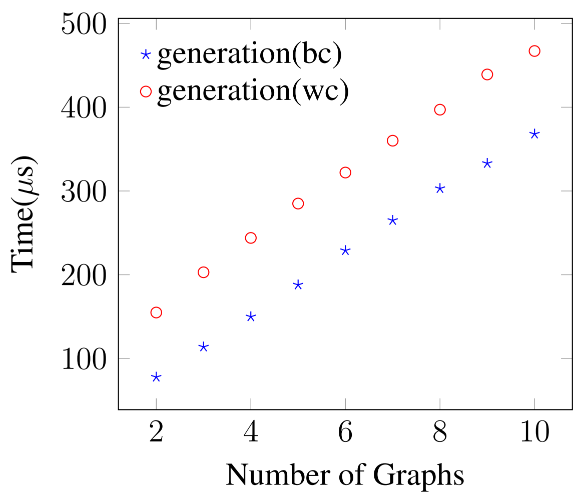
evaluating the adjacent list, whereas the actual design space grows with $|T(V)| \times n$, with no influence on the performance. The difference between the worst and the best case is due to the step-wise iteration, which evaluates only alternatives that are reachable from a specific vertex, instead of evaluating all combinations, which occurs in the worst case. Therefore, a reduced generation time is expected when the resulting CGP has less edges connecting vertices with references to different tasks, as occurs with the best case. Such a fact reinforces the previous observation that the algorithm performance depends much more on the task dependencies than on the number of alternatives, i.e. on the size and flexibility of an architecture.

8.1.3 Experiment 3: Number of design graphs

The mapping between multiple layers produced by the CGP also increases the generation time. Each layer is represented by a design graph, and the mapping between layers represents different design activities. This experiment has the goal to evaluate the scalability of the algorithm when increasing the number of design graphs participating in the CGP, so that more design activities can be solved simultaneously. The setup for this experiment uses a task graph arbitrarily fixed to 20 tasks. Again two sets of problems were produced, one set using a fully connected task graph for the worst case and the other one using a graph with vertex degree limited to 1 for the best case. These graphs were used to produce nine problems, increasing the number of design graphs up to 10. Each additional graph besides the task graph has a fixed size of five vertices, and they are fully connected. Figure 8.7 shows the average time to generate one solution for the generated problems.

Again the difference in performance between the worst and best cases is due to the difference in the connectivity presented in the task graphs used for each case. In spite of the exponential growth in the number of alternatives by combining multiple graphs, the results show a linear growth of the generation time for both the best and worst cases. The number of alternatives grows with $|Dg_n(V)|$, which means that the adjacent list off a vertex grows with the size of the vertex set of the design graph n, whereas the actual design space grows with $|Ds(V)| \times |Dg_n(V)|$, with no influence on the performance. Therefore, the generation algorithm presents a very good performance, allowing the simultaneous DSE of multiple activities by exploiting the CGP method to represent the

Figure 8.7: Evaluation of solution generation time by increasing the number of design graphs.



design space.

8.2 Case Study II: Electronic Wheelchair System

In order to illustrate the proposed methodology, this case study presents a DSE scenario for a real-time embedded system dedicated to the automation and control of an intelligent wheelchair that helps people with special needs. In this case study all steps presented in Figure 5.1 are shown, and it illustrates how the tools presented in Chapter 7 are used to automate the DSE flow. This case study focuses on the DSE CGP Mapping Problem presented in Definition 6.6, in order to highlight the method proposed in Chapter 6. All other DSE problems discussed in Chapter 5 can be solved in a similar way by using the DSED metamodel and the generated API to implement the engineer's heuristic or to integrate global optimizers into the framework. Such solving methods were discussed in Section 5.3.

8.2.1 DSE Scenario

The wheelchair system has several functions, such as movement control, collision avoidance based on ultrasound and stereo vision, navigation, target pursuit, battery control, system supervision, task scheduling, and automatic movement. The system application is modeled by using UML, as described in Section 5.2. PBD is considered in this case study, thus it is expected that a large amount of software and hardware components can be reused to develop the system. The reusable components are specified in the platform repository, as described in Section 7.2.6.

The DSE scenario consists in solving the DSE mapping problem extended with the CGP representation of the design space, presented in Section 6.3, so that the following design activities are performed:

- definition of which objects are active or passive (runnables), among the 17 behaviors defined in the interaction graphs;
- mapping of the active objects to available processors (up to 6 processors);

- allocation of the selected processors into a hierarchical bus with two segments;
- processor voltage scaling with four distinct voltage levels.

In order to illustrate how an engineer can customize the DSE methodology to specific scenarios by including his/her own heuristic, a cluster graph Cg was created from the UML sequence and interaction overview diagrams. By analyzing the dependencies between sequence diagrams, referenced in the interaction overview diagram, a cluster graph was produced to define if an interaction must be implemented by an active or passive object. In this way, the cluster graph contains vertices, which represents alternative clusters of interactions. The edges of the cluster graph represents dependencies between clusters, which emerges by grouping interactions with dependencies in different clusters. Selected vertices from the cluster graph represents active objects that wrap passive behavior of sequential interactions.

A fully connected processor graph Dg_p is defined to represent the processors available in the platform and the way they how can communicate. The graph $Cg \otimes Dg_p$ represents the possible mappings of vertices from the cluster graph onto to available processors.

The communication structure graph Dg_c represents a hierarchical bus with two segments, thus this graph contains two fully connected vertices to represent each segment. Processors mapped to the same segment can communicate without overhead, hence only two vertices are required. The graph $Dg_p \otimes Dg_c$ represents the possible mapping of processors onto the bus.

A graph Dg_v representing the four distinct voltage levels is defined. Each vertex represents a level and the edges represent the transitions between levels. In order to increase the benefits of voltage scaling, each active object can execute in a processor with a voltage level defined according to its computational and time requirements. The graph $Dg_c \otimes Dg_v$ represents the alternative assignments of voltage levels for each active object.

The design space for this DSE scenario is the graph $Cg \otimes Dg_p \otimes Dg_c \otimes Dg_v$ resulting from the CGP between those four design graphs. The design space graph contains 2,064 vertices and 334,080 edges, from where a set of up to 17 vertices (the maximum active task distribution is equal to the number of interaction graphs) must be selected to define a sub-graph, which represents a candidate design solution. The unveiled design space presents 5.89×10^{41} alternative designs, considering a unrestricted design space, when the step-wise iteration on the design space presented in Section 6.4 is not applied. However, in the proposed methodology, edges guide the available alternatives and constraints are locally applied between the current vertex and its neighbors, thus pruning the design space and speeding up the DSE process.

8.2.2 DSE Flow

8.2.2.1 Design Modeling

The methodology flow starts by modeling the wheelchair system as prescribed in Subsections 5.3 and 7.3. The UML model specifies the wheelchair movement control, collision avoidance, and navigation functions, which are essential to the system and incorporate critical hard real-time constraints. It consists of a Class model, 18 interaction diagrams, 1 interaction overview diagram, and one composite diagram. These UML diagrams were produced in the Magic Draw tool and can be found in Appendix 11.

The platform library was presented in Section 7.2.6 and provides software and hardware components to be reused during the implementation of the wheelchair control system. A UML model library mirrors the platform repository, so that an engineer can use the library components in the UML model.

8.2.2.2 DSED Modeling

A basic DSED model is available containing the information about solvers, evaluators, and the metrics that can be evaluated, thus reducing the manual effort for the specification. Moreover, part of the DSED model is automatically generated, by extending the basic DSED model with design graphs extracted from UML models, such as interaction, task, processor, and communication graphs.

When some information cannot be directly specified in or extracted from UML models, an engineer can use the DSED editor, generated by EMF, and specify the required information directly in the DSED model. In this case study the cluster and the voltage scale graphs were included in the DSED using the DSED editor.

8.2.2.3 Design-DSED Weaving

This step combines multiple design models with DSED, in order to support the link between variability and the design, back annotation, and other tasks. The Design-DSED Weaving model is created when data are automatically extracted from UML. The weaving with additional models, such as a Simulink model, is not supported yet. Moreover, manually inserted information in the DSED model requires manual weaving using the multi-pane editor provided by AMW weaving tool.

8.2.2.4 DSE Rules Generation

Two rules are standard in the DSE mapping problem and were used in the previous case study, namely `MandatoryMapping`, corresponding to Constraint 5.20 and `OneToManyMapping`, corresponding to Constraint 5.18, and are also used in this DSE scenario, so that all tasks must be mapped and tasks do not repeat in the solution. Additionally, mappings that lead to computational times greater than deadlines of real-time tasks must be rejected after the evaluation, hence the DSE rule `MaximumValue`, which refers to Constraint 5.1, is also specified. Moreover, processors can only be mapped to one segment of the bus, hence after the selection of the first mapping for a specific processor all other design decisions must obey the same mapping. Such a constraint is implemented by the `ManyToOneMapping`, which implements Constraint 5.19. Some DSE rules can be automatically generated from the UML models, such as `MaximumValue`, while the other rules are selected prior to the execution of the exploration.

8.2.2.5 Design Space Exploration and Evaluation

Before starting running the DSE process, an engineer can select the Solver and evaluation methods to be used, from the ones registered in the DSED model. In this case study we use the `CPACO Solver`, which implements the optimization algorithm and solution generation procedure described in Chapter 6, and the SPEU Evaluator, used to estimate the values of objectives from the generated solutions. H-SPEX was configured to optimize the system in terms of performance (cycles), power (μWatt), energy (μJoules), total memory (bytes), and communication volume (bytes transmitted on the bus).

The DSE process is executed automatically by H-SPEX until reaching a stop condition, which was set to 5,000 evaluations. The number of candidates generated at each iteration was set to 50 and the population size was set to 20.

8.2.2.6 Final Solution Selection and Back Annotation

The DSE result is the final population of non-dominated candidate designs. The best overall candidate must be selected after a trade-off analysis between the obtained estimations, based on some criteria, such as weights for the optimized objectives, or any other design feature. The outputs of the H-SPEX tool are a text report and a DSED model containing the final population characterized with the values estimated by SPEU. These outputs can be used in some analysis tools to aid the engineer in the task of selecting the solution(s) to be refined and implemented. The next section presents the results for this DSE scenario.

The resulting DSED model then can be used to back annotate the results into the original design models, by following the links contained in the Design-DSED Weaving model. Although the DSED metamodel and API are provided also for back annotation, this is not automated yet.

8.2.3 DSE Results

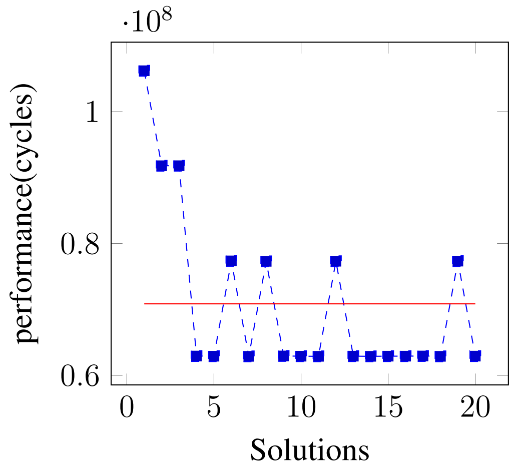
The candidate population was found after 5,000 evaluations, which was the stop condition. The DSE process ran 3 hours and evaluated proximately 1,666 candidates/hour. Such result shows the performance of the DSE and evaluation tools in a real-life application, which contains a huge design space. Figure 8.8 and Figure 8.9 show five plots with values of the objectives estimated for each solution. The horizontal line without marks shows the average value.

The first observation in Figure 8.9 is that the communication strongly influences the system performance, so that reducing data transfers is important to optimize the system. However, as all solutions fulfill the system performance requirements, other solutions may provide better gains in other objectives than the solution with best performance.

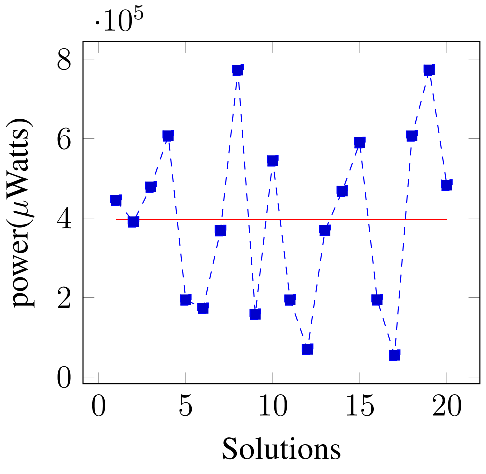
The results show that solutions 3 and 19 are above the average for all objectives, hence they are not good candidates for further developments. These solutions do not presented an adequate load balance, because they require more cycles to execute and dissipate more power to fulfill the requirements. From the set of solutions that present at least 4 values below the average, six solutions, namely 5, 7, 11, 13, 16 and 13, present results above the average for energy consumption, and, from them, solution 7 and 13 present the highest figures. These solutions present two clusters, which agglomerate functions with many dependencies under the same task. Such an approach reduces the communication and execution cycles required by data transfer and scheduling, hence these solutions presented good performance figures. However, these clusters are responsible for most of the execution cycles, which are executed at higher power in order to fulfill the computation requirements, therefore increasing the energy consumption.

Solutions 4, 9, 14, 15, 17, and 20 present figures closer to the average. They have a smaller number of processors and a smart distribution of the 17 behaviors in task clusters. Not as a coincidence, these solutions present the highest number of design decisions that appear more frequently in the population. This fact illustrates the convergence of the search, which can be improved by changing the algorithm parameters, such as number of evaluations, size of the population, and crowding factor. These parameters were empirically selected, based on the time to evaluate a solution, previous runs and recommended parameters (ANGUS, 2007).

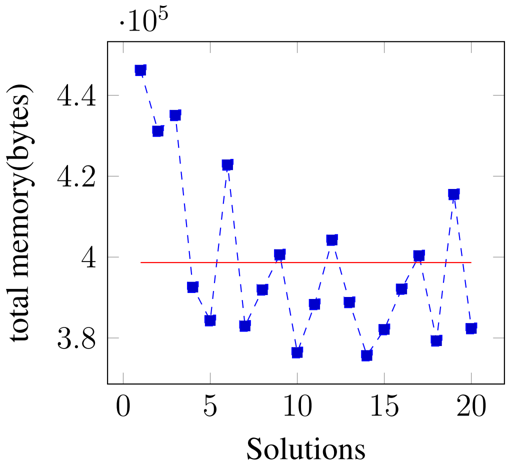
Figure 8.8: DSE Results for the Wheelchair System: (a) performance (cycles); (b) power (μWatts); (c) total memory (bytes).



(a)

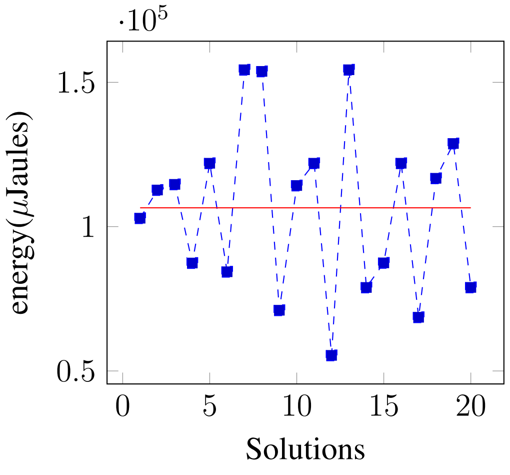


(b)

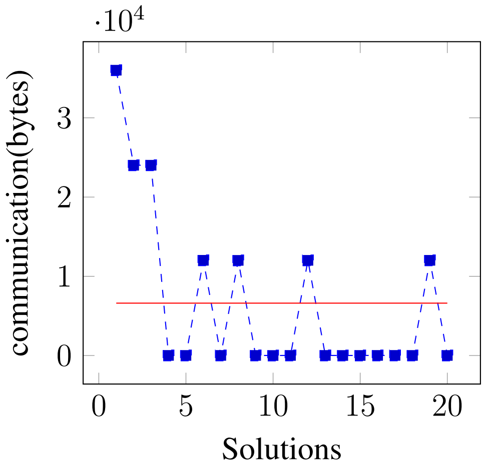


(c)

Figure 8.9: DSE Results for the Wheelchair System: (a) energy (μJaules); (b) communication (bytes).



(a)



(b)

8.3 Discussion

This chapter presented two case studies, one aiming at the evaluation of the method proposed to improve mapping in PBD and another one to show a complete flow of the DSE methodology applied to a real-life application.

The first case study demonstrates the scalability of the method proposed to generate candidate designs, when executed for problems with different sizes and complexities. Three experiments were performed. The first one showed the average time to generate candidate designs, when increasing the application size, considering graphs with connectivity in best, average, and worst cases. Although the generation time grows exponentially in the worst case, the time grows polynomially in the best and average cases. Moreover, the generation time for problems extracted from an industrial benchmark grows close to the best case. The second experiment showed the growth of generation time, when the number of alternatives in an architectural graph increased. The growth in this case is linear with the growth of the architectural graph in the best and worst cases. The last experiment showed the growth of the execution time when design graphs are included in the CGP. As expected, the logical implementation of the CGP favors the simultaneous mapping of multiple graphs, which grows linearly with the addition of new graphs, although the number of possible combination grows exponentially. Finally, by evaluating all three experiments one notices that the connectivity of the graph representing the application has more influence on the performance of the method than the number of alternatives and the number of layers to be mapped. Moreover, it was shown that the proposed method is efficient and scalable, hence it can be applied to larger problems - as it was demonstrated in the second case study.

The second case study illustrates the methodology by applying it to the design of an electronic wheelchair control system. This case study shows with examples each process of the methodology: system modeling with UML, DSE domain modeling, Design-DSED weaving, exploration rule generation, design space exploration and evaluation, and the selection of the final solution. In order to illustrate the flexibility of the proposed methodology an heuristic was adopted, such that a cluster graph is used to replace the task graph and indicates some possible clusters of tasks. This heuristic aims to guide the DSE process by reducing the communication and scheduling costs. However, it was intended to increase the number of combinations after the CGP calculation, which resulted in a huge design space of 5.89×10^{41} design alternatives, and illustrates the scalability of the method for real-life problems. Although the resulting design space was much larger than the design spaces in the first case study, the method for design candidate generation demonstrates good performance, by evaluating proximately 1,666 candidates/hour, including a static analysis of the candidates.

9 CONCLUSION

This thesis proposes a DSE methodology for embedded systems, which consists in the exploitation of the MDE approach to improve different deficiencies in the current DSE methodologies, such as: *i*) restricted support to multiple DSE activities; *ii*) fixed method to generate solutions; *iii*) tightly coupling to evaluation tools; *iv*) lack of integration with the development process. Moreover, this thesis proposes methods to overcome some challenges faced by these methods, namely the increasing number of alternative decisions in a design space with reduced number of feasible designs due to stringent requirements, as well as the inter-dependencies between such decisions. Furthermore, in the state-of-the-art the trade-off between heuristics for specific DSE problems and global optimizers prevents the development of tools that are simultaneously efficient and flexible.

The DSE methodology was completely implemented for an MDE process. The result was a Domain Specific MDE Tool, which better integrates the DSE activities into a development process. A simple development process was defined, in order to provide a real development context for the DSE methodology, and the flow of data in the process was automated by using model transformations, which alleviate the development effort. The process identifies methods, tools, and artifacts required to promote an easy deployment of the methodology.

A lightweight modeling method was proposed to specify a system by using UML and MARTE. This method collects common practices in modeling, so that the development flow could be automated without enforcing stringent modeling requirements. Because it focuses in the most common modeling elements from UML and MARTE, the method can be used in conjunction with other tools, such as different UML editors, GenERTiCA for code generation, and Cheddar for schedulability analysis.

After review of the current DSE methods, a DSE Domain metamodel was defined. It represents the important elements of a DSE process, such as available solver and evaluation tools and metrics to be optimized or used to guide the DSE process, and represent these elements in a concise and uniform way. Moreover, DSED can represent four different DSE problems identified from the studied literature and classified according to (SAXENA; KARSAI, 2011), namely construction, configuration, mapping, and scheduling. Each DSE problem contributes to the DSED metamodel, which provides abstractions to represent problem elements, solutions, and constraints specialized for each problem. Such abstractions improve the specification of DSE models, which require less specialization of an engineer in optimization and formal techniques, usually required by general optimization frameworks.

In order to improve the flexibility of the DSE methodology, so that models in different languages can be used, a model weaving method was adopted. This method is supported by the AWM tool, which weaves design and DSED model elements at the beginning of the

DSE process. Besides multiple input models, the weaving method aids the automation of DSE activities, such as back annotation of DSE results in the design models, and improves the traceability and separation of concerns, namely design and design and DSE.

Models conforming to DSED models are handled in every step by model-to-model transformations, which extract required information from development models and fill the DSE models. By including domain specific knowledge, namely DSE, in the transformations rules, these rules could also be used to implement DSE rules, in order to guide the automatic DSE process and prune the design space. Moreover, model-to-model transformations are an adequate mechanism to specify configurable, reusable, and complex DSE rules, if an engineer needs additional constraints according to a specific DSE scenario.

A method to represent the design space as a Categorical Graph Product (CGP) was developed, in order to improve the mapping between layers in PBD approaches. The CGP is adequate to represent such a mapping, because it maps automatically multiple design graphs, improves the abstraction, and represents multiple design decisions involved in the mapping as one single unified decision. Therefore, CGP is appropriate for representing simultaneous and interdependent design alternatives. Moreover, CGP exposes element dependencies through all graphs, which is especially important to optimize the communication in different systems aspects (e.g. tasks, processors, and buses).

Based on the CGP representation of the design space, a method to generate solutions for the DSE Mapping problem was implemented. It induces a step-wise search in the design space. This search is guided by constraints applied locally at each vertex's adjacency as the search algorithm iterates on graphs resulting from CGP, in order to select the sub-graph which represents an alternative design. Such an approach avoids the enumeration of all possibilities, by removing alternative vertices that do not fulfill the specified constraints. Experiments have shown the generation method being able to deal with large design spaces, because the time to generate a solution grows linearly, despite of the exponential growth of combinations, by increasing the number of alternatives or graphs. The results have also shown that the time to generate solutions grows exponentially in the worst case, when the number of vertices in the application grows. However, this time grows polynomially in the best case. Moreover, experiments with an industrial benchmark have shown that the time to generate solutions grows also polynomially for real-life applications, and close to the best case.

The proposed methodology is supported by a set of tools developed during this PhD work. This set includes the MODES framework, which provides domain specific (meta)-models and transformations for embedded systems development. Through MODES all other tools were integrated, such as UML model editor, the H-SPEX DSE tool, and the SPEU estimation tool, so that a complete automated process could be supported. The MODES framework was extended, by refactoring original metamodels in smaller parts and adding other metamodels and transformations.

The design space exploration of a large design space requires quick evaluation tools, hence the estimation tool SPEU was developed to support quick evaluation of embedded systems specified in UML models. SPEU provides analytical estimates about physical system properties, such as performance, power dissipation, and volume of communication. The estimation is performed by extracting structural and behavioral information from UML models and generating a CDFG. In this graph information of pre-designed components from a platform repository is annotated and used to improve the estimation accuracy. An ILP formulation for each CDFG generated is solved, in order to identify the worst-case execution paths and calculate the final estimates. By using SPEU, HSPEX can rapidly evaluate candidate solutions during the DSE process, without depending on costly synthesis-and-simulation evaluation cycles.

The H-SPEX tool was implemented to orchestrate and support the MDE approach for automated DSE. H-SPEX provides an implementation of the methods proposed in Chapters 5 and 6. It relies on the Java API generated from the DSED metamodel to configure and execute the DSE process. It coordinates also the transformation of models from and to tools required in the process.

H-SPEX implements CPACO-MO, which is an implementation of a global optimization algorithm based on Ant Colony Systems and Evolutionary Algorithms. This algorithm was adapted to exploit the proposed design space representation through a logic implementation of a CGP inside the procedure for the generation of candidate designs. By combining CGP and CPACO-MO this proposal balances the trade-off between global optimization and heuristic optimization. CPACO-MO represents the global optimization methods, which are flexible and incorporate few knowledge of the problem domain, in detriment of performance. The generation of candidate designs and CGP allow the specification of heuristics to guide the design space dealing with simple abstractions - vertices and edges, representing the mapping between layers in the PBD approach. Moreover, model transformations can be used to customize the heuristic and prune the design space, by dealing with the concepts defined in the DSED metamodel.

9.1 Future work

Development, integration, and adaption of solvers were not the main contribution of this thesis, because the methodology was developed to be applied with different solvers. However, CPACO-MO algorithm was implemented and adapted to solve the DSE Mapping Problem with a logical implementation of the CGP. A general proposal to to solve the other three DSE problem is the integration of the FORMULA tool, by transforming the specification of DSED metamodel in ECORE to a specification in FORMULA. Moreover, CPACO-MO requires further evaluation and comparison with state-of-the-art algorithms applied for similar problems, because each algorithm is better suited to a specific class of problems.

This thesis strongly relies on transformation engines and transformation languages. In that MDE gets more importance in the system development, more and more engines and languages are proposed. Because each one has different characteristics, studies show that they can have completely different performance figures, from an exponential to a linear growth of execution time. Therefore, further studies of engines and languages would improve the knowledge on specific application niches for which each one is more appropriate and on their influence in domain specific MDE tools, and in particular in tools for DSE.

Now that the fundamentals for Model-driven Design Space Exploration are settled, by representing the DSE domain in an adequate way through two metamodels, namely DSED and a transformation language, it is interesting to bridge the requirements specified in the early development phase to the DSE rules defined as a transformation language. Since requirements could be formalized as constraints, such as in OCL, and most transformation languages use OCL as basis to define expressions, such a constraint forwarding process seems to be natural, even if it may require intermediate steps, such as the transformation of requirements into design models.

In this thesis six DSE problems were specified in the DSED metamodel. Although a general method using FORMULA to find a solution for all specified problems can be implemented, not all of these problems were specifically approached. Similarly to the DSE Mapping Problem, which was improved by using CGP to represent the design space and a combination of heuristic search with global optimization algorithm (CPACO-MO), other problems can also be approached with specific heuristics.

An effort that is always valuable is to automate steps during the development process. In particular, the extraction of data from design models can be improved or extended. For example, design graphs could be extracted from SystemC models or SysML. Another example would be the automation of the specification of parameter interdependency. This was originally proposed in (GIVARGIS; VAHID, 2002), and the DSED metamodel is able to represent configuration problems by using this method. The weaving of design and DSED models facilitates the back annotation of the results into the design models. This step can be automated in the future.

During the evolution of this thesis many tools were developed. However, all of them are prototypes and can be improved. A tool for (meta)model management is required, namely mega-model management tools, such as AtlanMod MegaModel Management (AM3)[1] or MoScript[2]. Metamodels and transformations must also evolve to reflect the community requirements, as well as the MODES framework requires more (meta)models and transformations to completely fulfill its goals. SPEU can exploit the UML Test profile and the MARTE Analysis profile, in order to improve the evaluation scenario specification and the estimation. Besides the automation of more activities in the H-SPEX tool, its GUI and usability can be improved, as well as its integration with other analysis, solver, and estimation tools.

[1]http://www.wiki.eclipse.org/AM3

[2]htpp://www.eclipse.org/MoScript

10 GRAPHS OF E3S BENCHMARKS SUITE (E3S)

10.1 List of E3S Tasks

Automotive/Industrial

0 Angle to Time Conversion

1 Basic floating point

2 Bit Manipulation

3 Cache Buster

4 CAN Remote Data Request

5 Fast Fourier Transform (Auto/Indust. Version)

6 Finite Impulse Response Filter (Auto/Indust. Vers)

7 Infinite Impulse Response Filter

8 Inverse discrete cosine transfom

9 Inverse Fast Fourier Transform (Auto/Indust. Vers)

10 Matrix arithmetic

11 Pointer Chasing

12 Pulse Width Modulation

13 Road Speed Calculation

14 Table Lookup and Interpolation

15 Tooth To Spark

Consumer

16 Compress JPEG

17 Decompress JPEG

18 High Pass Grey-scale filter

19 RGB to CYMK Conversion

20 RGB to YIQ Conversion

Networking

21 OSPF/Dijkstra

22 Route Lookup/Patricia

23 Packet Flow - 512 kbytes

24 Packet Flow - 1 Mbyte

25 Packet Flow - 2 Mbytes

Office automation

26 Dithering

27 Image Rotation

28 Text Processing

Telecom

29 Autocorrelation - Data1 (pulse)

30 Autocorrelation - Data2 (sine)

31 Auto-Correlation - Data3 (speech)

32 Convolutional Encoder - Data1 (xk5r2dt)

33 Convolutional Encoder - Data2 (xk4r2dt)

34 Convolutional Encoder - Data3 (xk3r2dt)

35 Fixed-point Bit Allocation - Data2 (typ)

36 Fixed-point Bit Allocation - Data3 (step)

37 Fixed Point Bit Allocation - Data6 (pent)

38 Fixed Point Complex FFT - Data1 (pulse)

39 Fixed point Complex FFT - Data2 (spn)

40 Fixed Point Complex FFT - Data3 (sine)

41 Viterbi GSM Decoder - Data1 (get)

42 Viterbi GSM Decoder - Data2 (toggle)

43 Viterbi GSM Decoder - Data3 (ones)

44 Viterbi GSM Decoder - Data4 (zeros)

45 Placeholder task (sink / src)

10.2 E3S Graphs

Figure 10.1: E3S task graph set representing an office automation application.

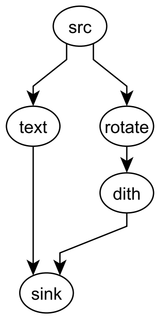


Figure 10.2: E3S task graph set representing a consumer electronics application.

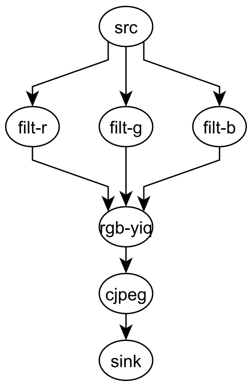


Figure 10.3: E3S task graph set representing a networking application.

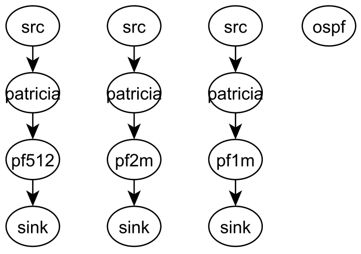


Figure 10.4: E3S task graph set representing an automotive application.

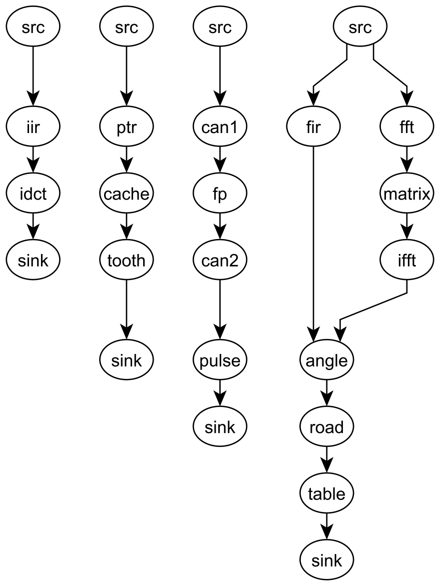
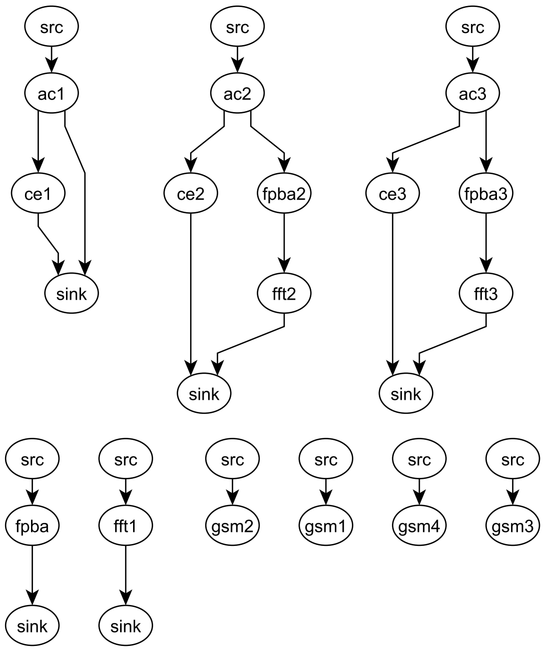


Figure 10.5: E3S task graph set representing a telecommunication application.



11 WHEELCHAIR SYSTEM: UML DIAGRAMS

Figure 11.1: Wheelchair System: Sequence Diagram - Movement control.

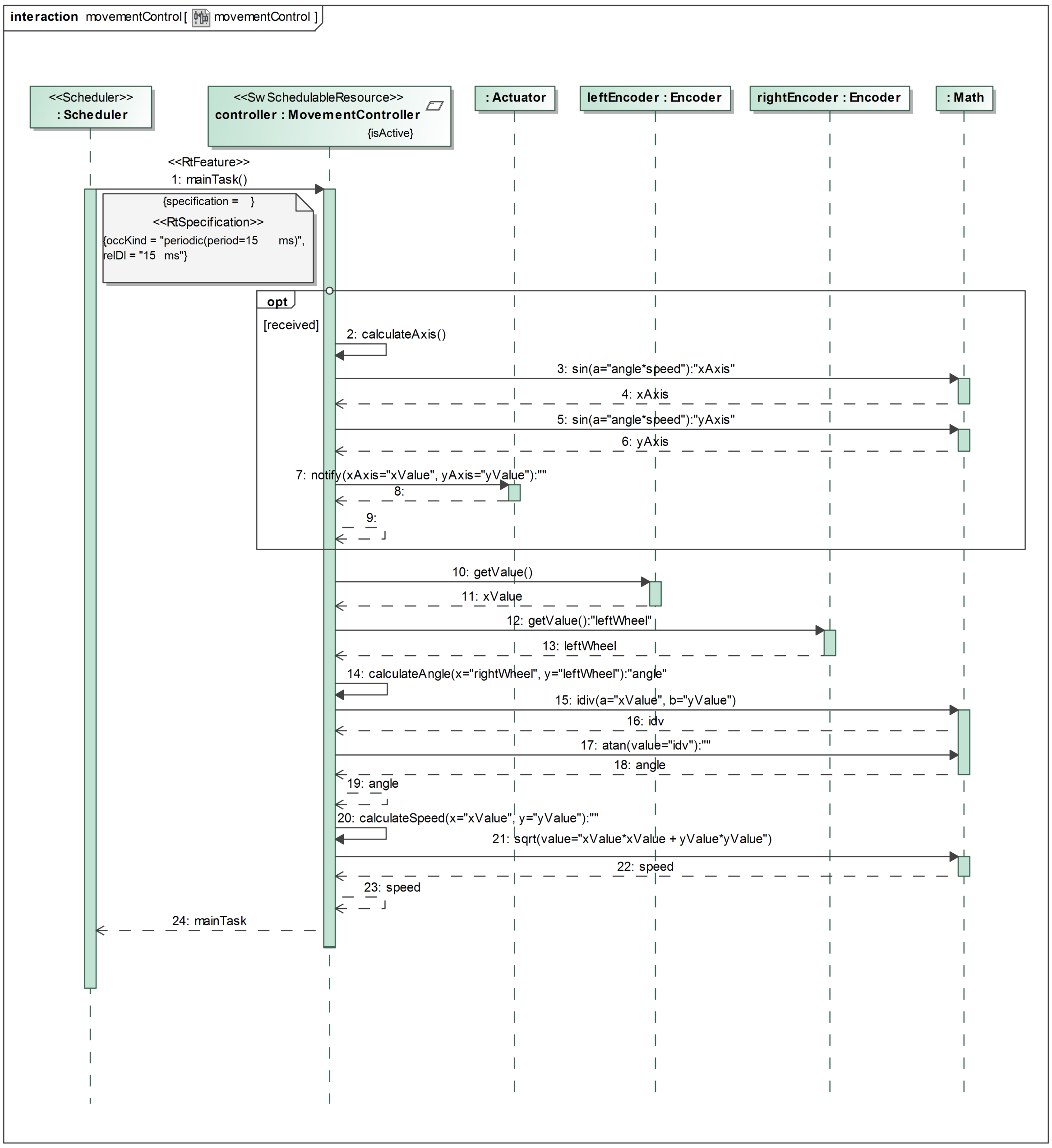

interaction movementControl [movementControl]
<<Scheduler>> : Scheduler
<<Sw SchedulableResource>> controller : MovementController {isActive}
: Actuator
leftEncoder : Encoder
rightEncoder : Encoder
: Math
<<RtFeature>> 1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=15 ms)", relDl = "15 ms"}
opt
[received]
2: calculateAxis()
3: sin(a="angle*speed"):"xAxis"
4: xAxis
5: sin(a="angle*speed"):"yAxis"
6: yAxis
7: notify(xAxis="xValue", yAxis="yValue"):""
8:
9:
10: getValue()
11: xValue
12: getValue():"leftWheel"
13: leftWheel
14: calculateAngle(x="rightWheel", y="leftWheel"):"angle"
15: idiv(a="xValue", b="yValue")
16: idv
17: atan(value="idv"):""
18: angle
19: angle
20: calculateSpeed(x="xValue", y="yValue"):""
21: sqrt(value="xValue*xValue + yValue*yValue")
22: speed
23: speed
24: mainTask

Figure 11.2: Wheelchair System: Sequence Diagram - Movement interface reading.

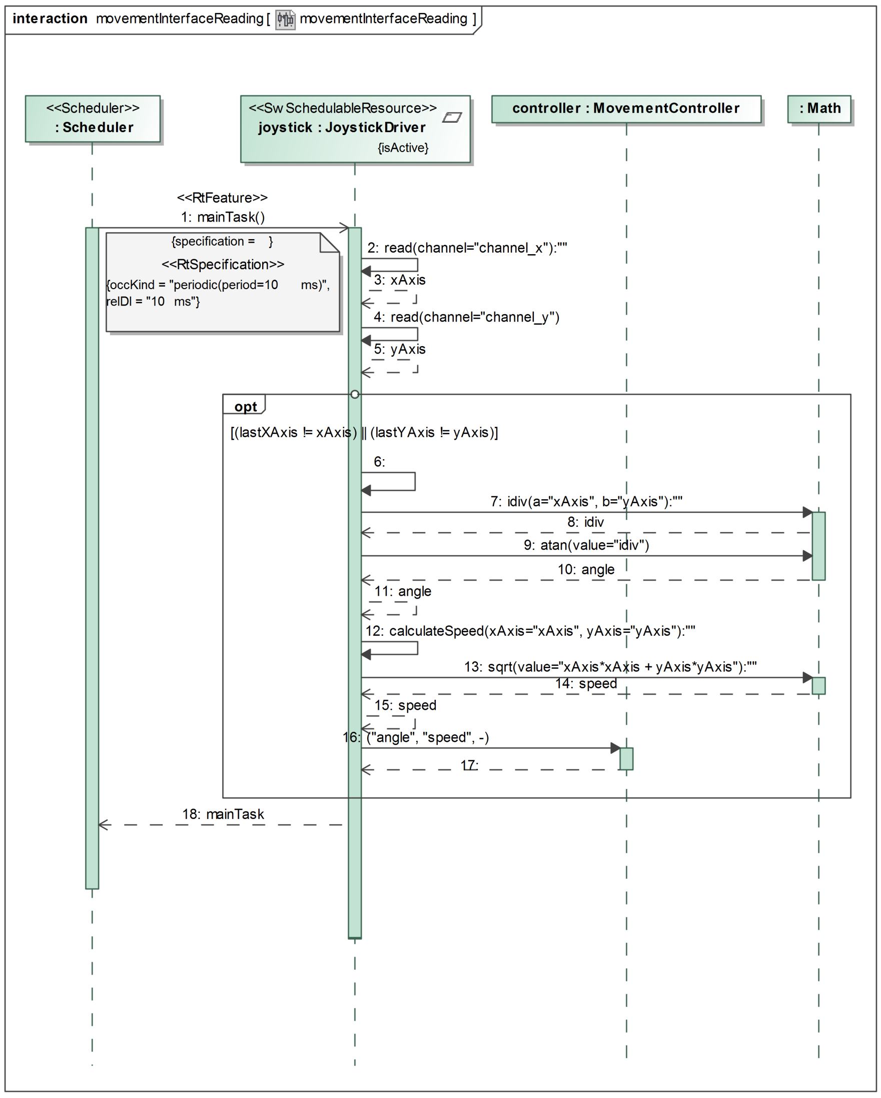

interaction movementInterfaceReading [movementInterfaceReading]
<<Scheduler>>
: Scheduler
<<SwSchedulableResource>>
joystick : JoystickDriver
{isActive}
controller : MovementController
: Math
<<RtFeature>>
1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=10 ms)",
relDl = "10 ms"}
2: read(channel="channel_x"):""
3: xAxis
4: read(channel="channel_y")
5: yAxis
opt
[(lastXAxis != xAxis) || (lastYAxis != yAxis)]
6:
7: idiv(a="xAxis", b="yAxis"):""
8: idiv
9: atan(value="idiv")
10: angle
11: angle
12: calculateSpeed(xAxis="xAxis", yAxis="yAxis"):""
13: sqrt(value="xAxis*xAxis + yAxis*yAxis"):""
14: speed
15: speed
16: ("angle", "speed", -)
17:
18: mainTask

Figure 11.3: Wheelchair System: Sequence Diagram - Actuator interface writing.

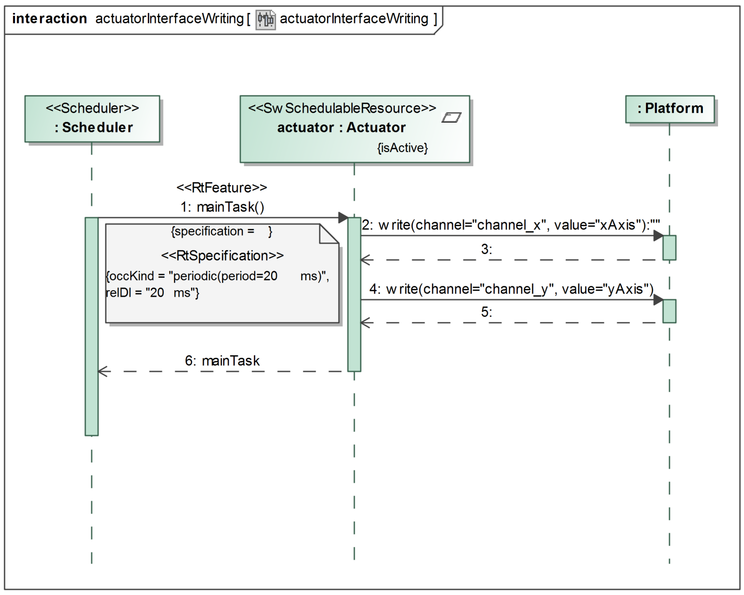

interaction actuatorInterfaceWriting [actuatorInterfaceWriting]
<<Scheduler>>
: Scheduler
<<Sw SchedulableResource>>
actuator : Actuator
{isActive}
: Platform
<<RtFeature>>
1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=20 ms)", relDl = "20 ms"}
2: write(channel="channel_x", value="xAxis"):""
3:
4: write(channel="channel_y", value="yAxis")
5:
6: mainTask

Figure 11.4: Wheelchair System: Sequence Diagram - Generate depth map.

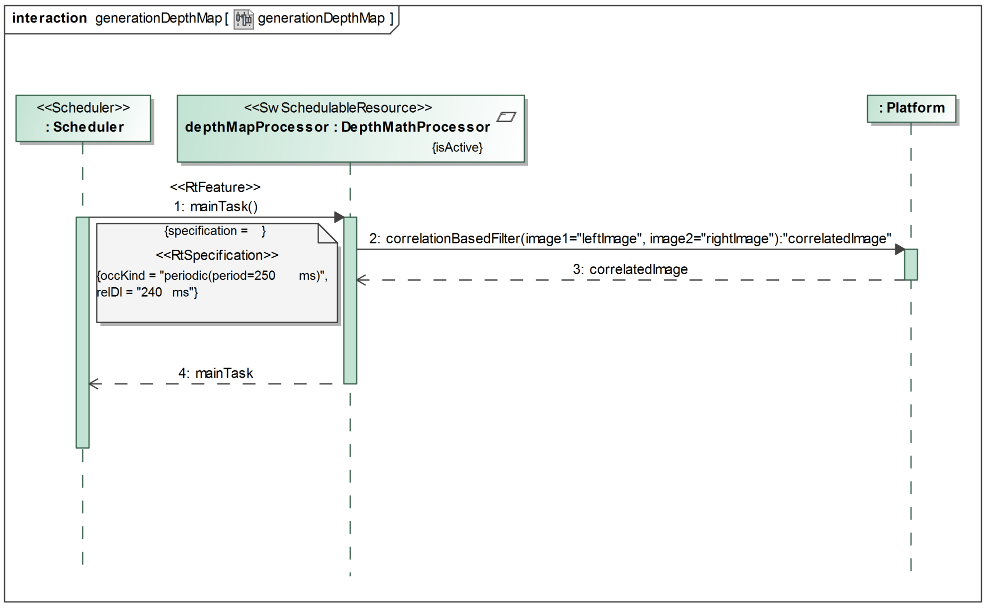

interaction generationDepthMap [generationDepthMap]
<<Scheduler>>
: Scheduler
<<Sw SchedulableResource>>
depthMapProcessor : DepthMathProcessor
{isActive}
: Platform
<<RtFeature>>
1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=250 ms)", relDl = "240 ms"}
2: correlationBasedFilter(image1="leftImage", image2="rightImage"):"correlatedImage"
3: correlatedImage
4: mainTask

Figure 11.5: Wheelchair System: Sequence Diagram - Left image processing.

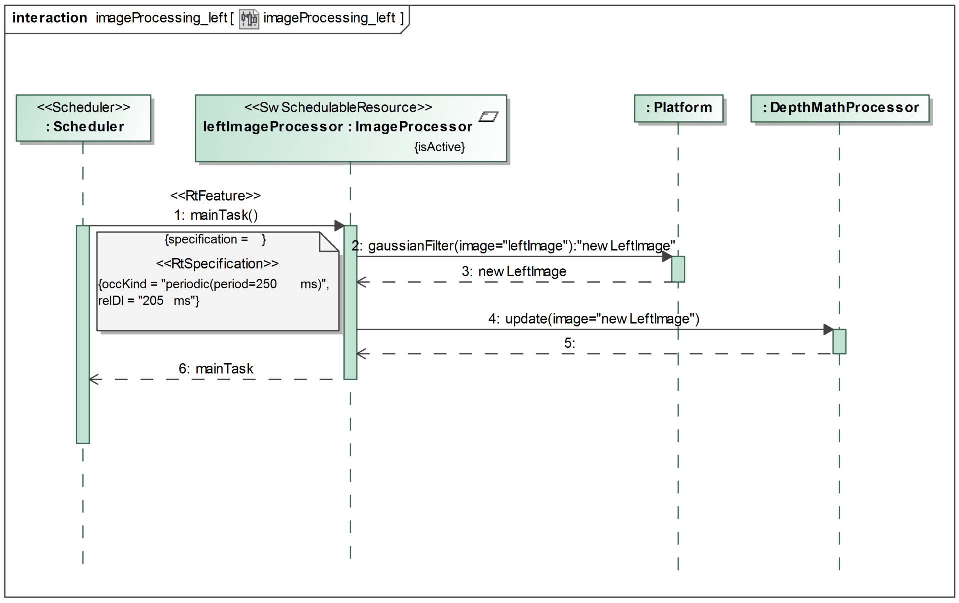

interaction imageProcessing_left [imageProcessing_left]
<<Scheduler>>
: Scheduler
<<Sw SchedulableResource>>
leftImageProcessor : ImageProcessor
{isActive}
: Platform
: DepthMathProcessor
<<RtFeature>>
1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=250 ms)",
relDl = "205 ms"}
2: gaussianFilter(image="leftImage"):"new LeftImage"
3: new LeftImage
4: update(image="new LeftImage")
5:
6: mainTask

Figure 11.6: Wheelchair System: Sequence Diagram - Right image processing.

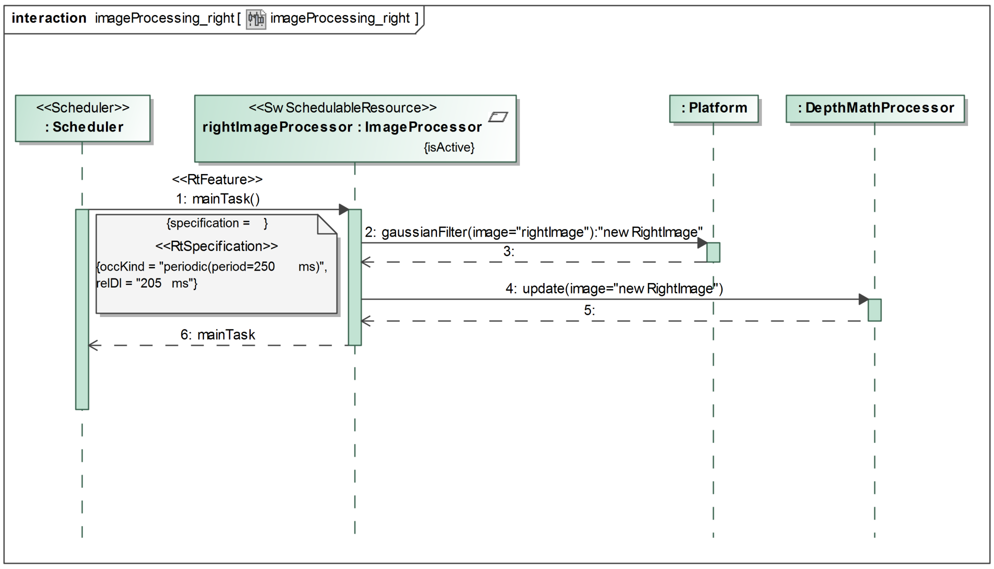

interaction imageProcessing_right [imageProcessing_right]
<<Scheduler>>
: Scheduler
<<Sw SchedulableResource>>
rightImageProcessor : ImageProcessor
{isActive}
: Platform
: DepthMathProcessor
<<RtFeature>>
1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=250 ms)",
relDl = "205 ms"}
2: gaussianFilter(image="rightImage"):"new RightImage"
3:
4: update(image="new RightImage")
5:
6: mainTask

Figure 11.7: Wheelchair System: Sequence Diagram - Right image sensoring.

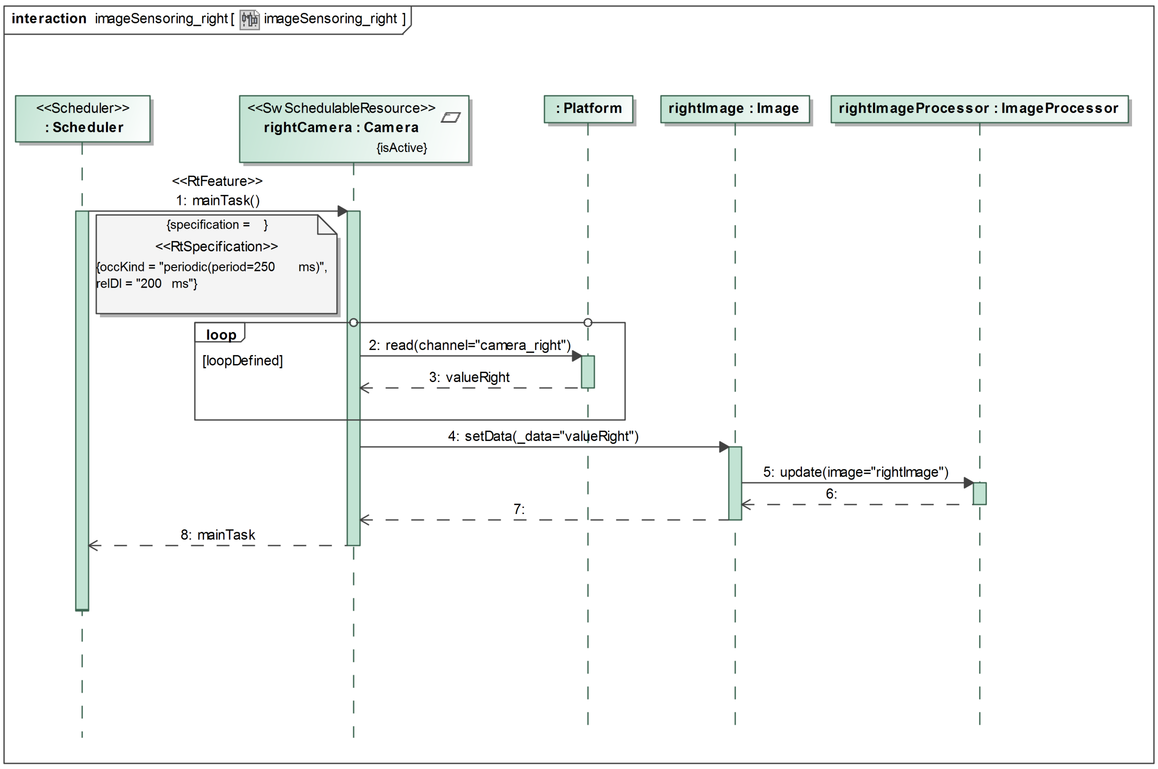

interaction imageSensoring_right [imageSensoring_right]
<<Scheduler>>
: Scheduler
<<Sw SchedulableResource>>
rightCamera : Camera
{isActive}
: Platform
rightImage : Image
rightImageProcessor : ImageProcessor
<<RtFeature>>
1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=250 ms)",
relDl = "200 ms"}
loop
[loopDefined]
2: read(channel="camera_right")
3: valueRight
4: setData(_data="valueRight")
5: update(image="rightImage")
6:
7:
8: mainTask

Figure 11.8: Wheelchair System: Sequence Diagram - Left image sensoring.

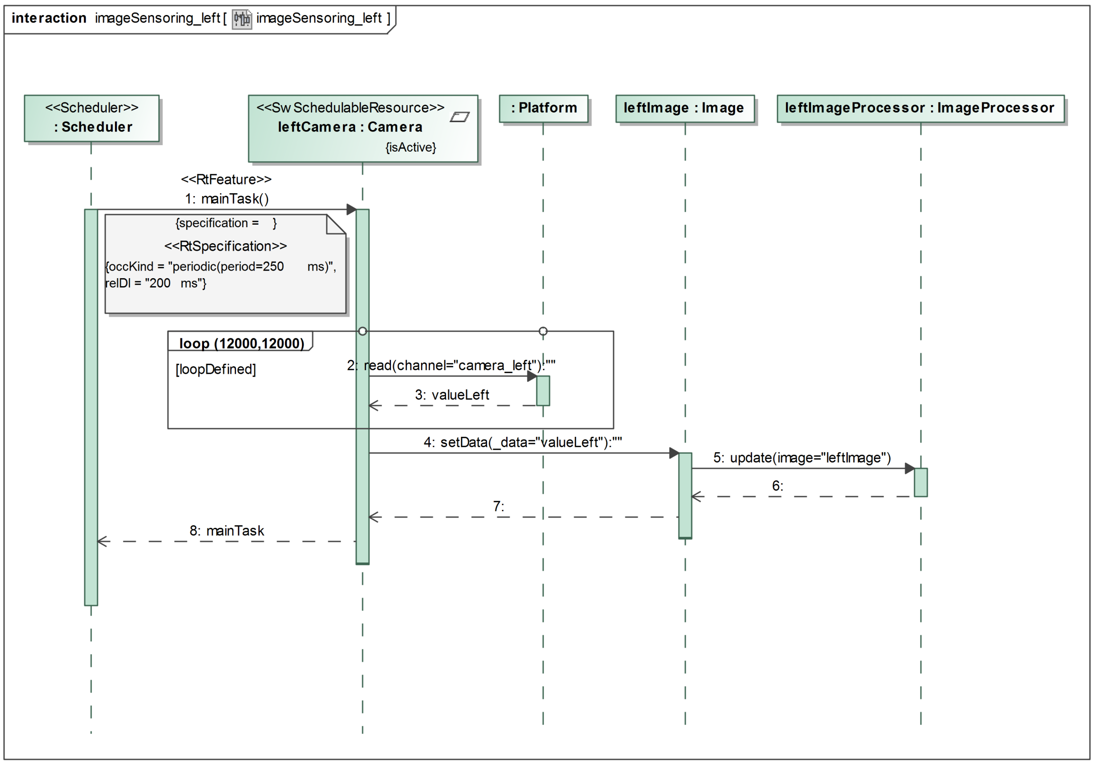

interaction imageSensoring_left [imageSensoring_left]
<<Scheduler>>
: Scheduler
<<Sw SchedulableResource>>
leftCamera : Camera
{isActive}
: Platform
leftImage : Image
leftImageProcessor : ImageProcessor
<<RtFeature>>
1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=250 ms)",
relDl = "200 ms"}
loop (12000,12000)
[loopDefined]
2: read(channel="camera_left"):""
3: valueLeft
4: setData(_data="valueLeft"):""
5: update(image="leftImage")
6:
7:
8: mainTask

Figure 11.9: Wheelchair System: Sequence Diagram - Right encoder reading.

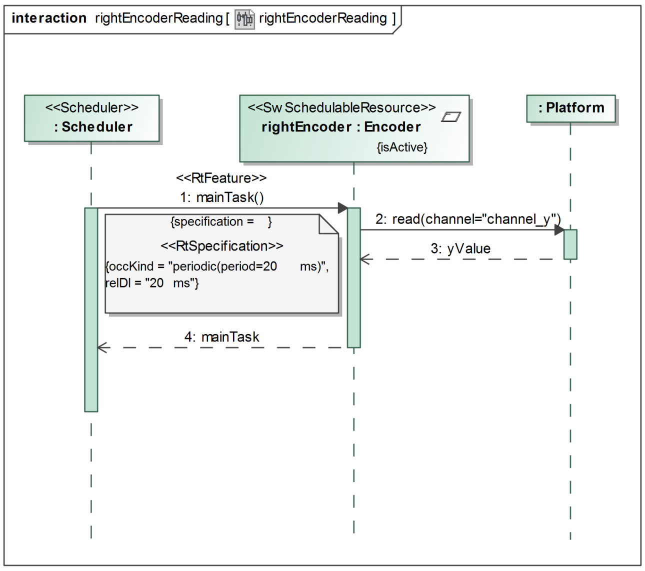


Figure 11.10: Wheelchair System: Sequence Diagram - Left encoder reading.

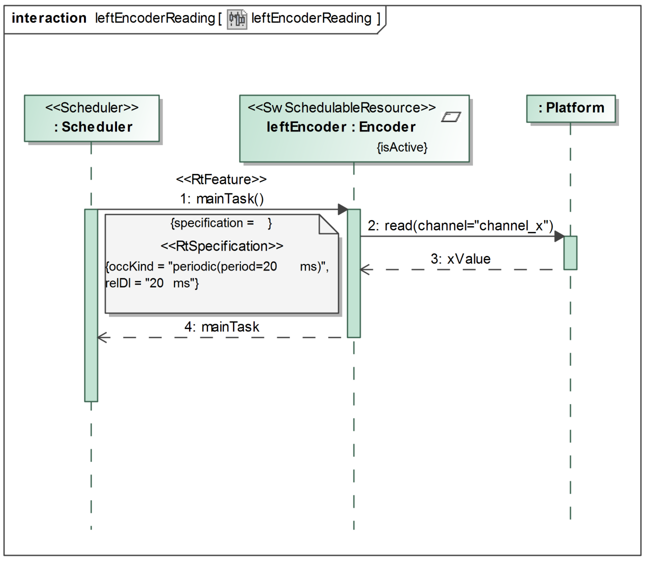


Figure 11.11: Wheelchair System: Sequence Diagram - Navigation.

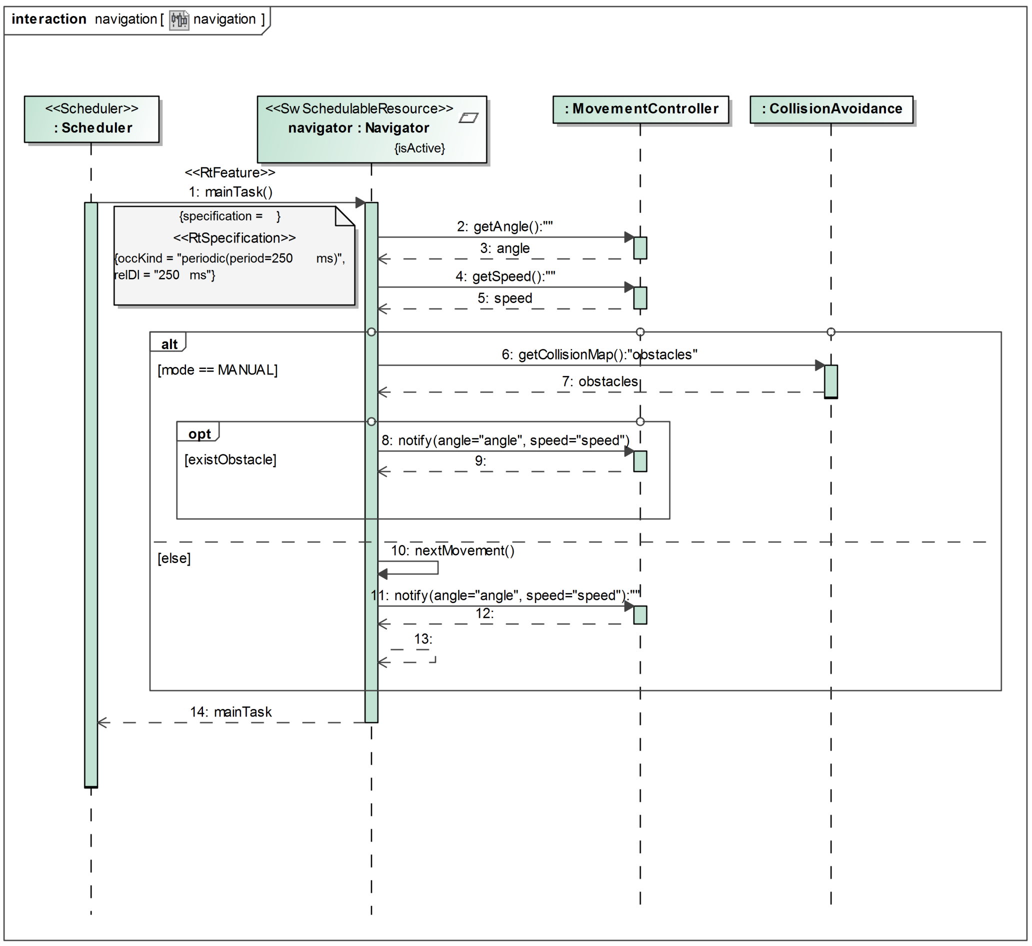


Figure 11.12: Wheelchair System: Sequence Diagram - Sytem initialization.

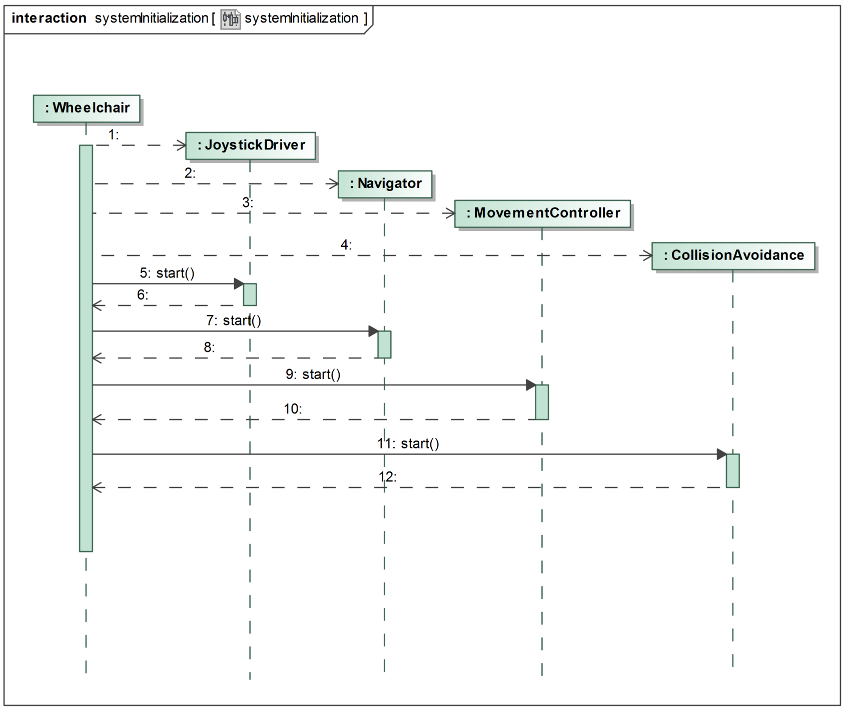


Figure 11.13: Wheelchair System: Sequence Diagram - Ultrasound processing.

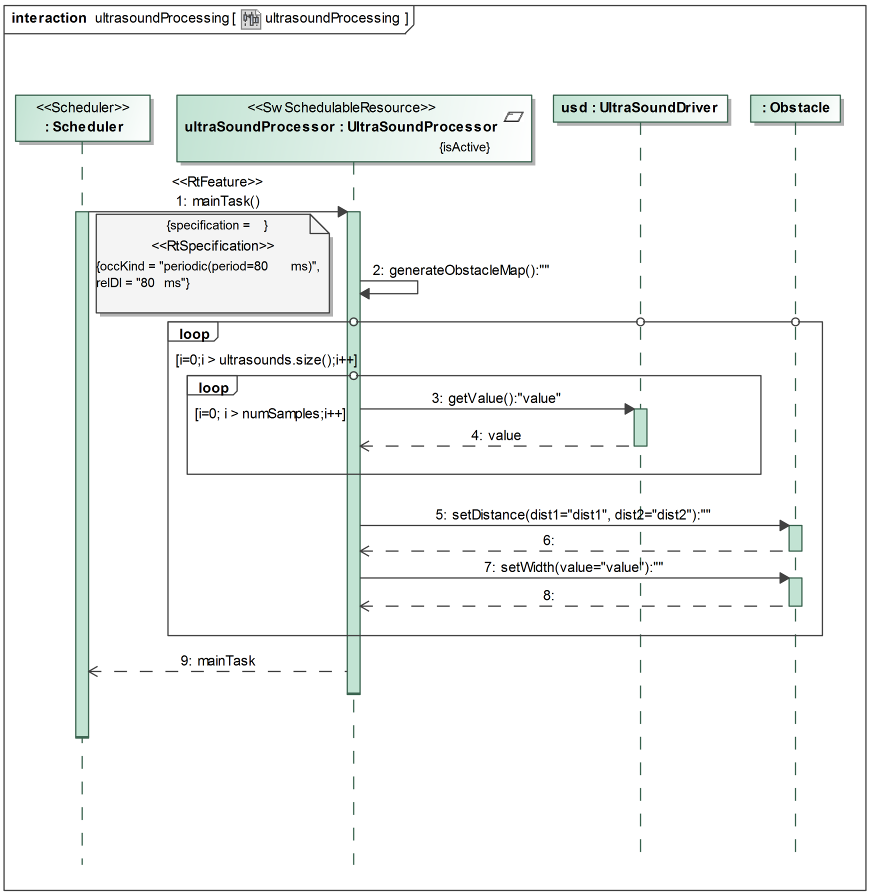

interaction ultrasoundProcessing [ultrasoundProcessing]
<<Scheduler>>
: Scheduler
<<Sw SchedulableResource>>
ultraSoundProcessor : UltraSoundProcessor
{isActive}
usd : UltraSoundDriver
: Obstacle
<<RtFeature>>
1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=80 ms)",
relDl = "80 ms"}
2: generateObstacleMap():""
loop
[i=0;i > ultrasounds.size();i++]
loop
[i=0; i > numSamples;i++]
3: getValue():"value"
4: value
5: setDistance(dist1="dist1", dist2="dist2"):""
6:
7: setWidth(value="value"):""
8:
9: mainTask

Figure 11.14: Wheelchair System: Sequence Diagram - Ultrasound reading 1.

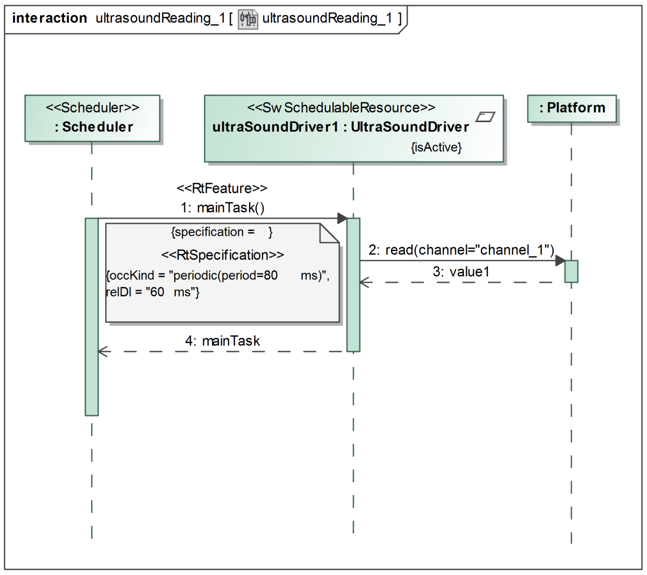

interaction ultrasoundReading_1 [ultrasoundReading_1]
<<Scheduler>>
: Scheduler
<<Sw SchedulableResource>>
ultraSoundDriver1 : UltraSoundDriver
{isActive}
: Platform
<<RtFeature>>
1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=80 ms)",
relDl = "60 ms"}
2: read(channel="channel_1")
3: value1
4: mainTask

Figure 11.15: Wheelchair System: Sequence Diagram - Ultrasound reading 2.

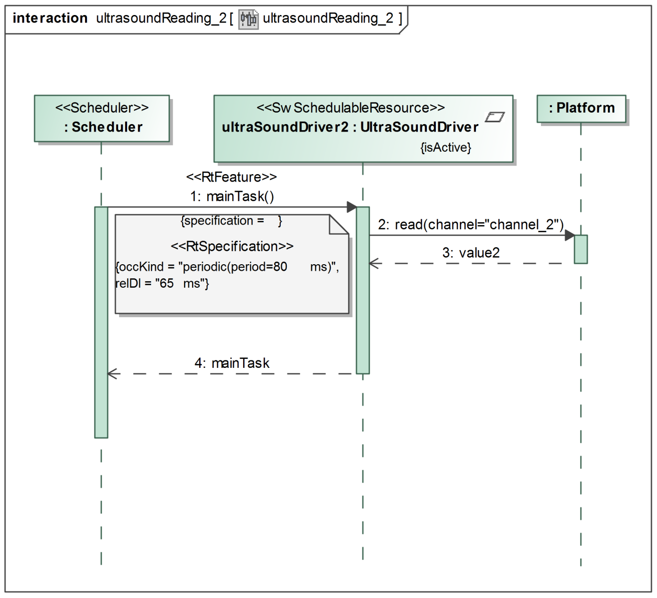

interaction ultrasoundReading_2 [ultrasoundReading_2]
<<Scheduler>>
: Scheduler
<<Sw SchedulableResource>>
ultraSoundDriver2 : UltraSoundDriver
{isActive}
: Platform
<<RtFeature>>
1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=80 ms)",
relDl = "65 ms"}
2: read(channel="channel_2")
3: value2
4: mainTask

Figure 11.16: Wheelchair System: Sequence Diagram - Ultrasound reading 3.

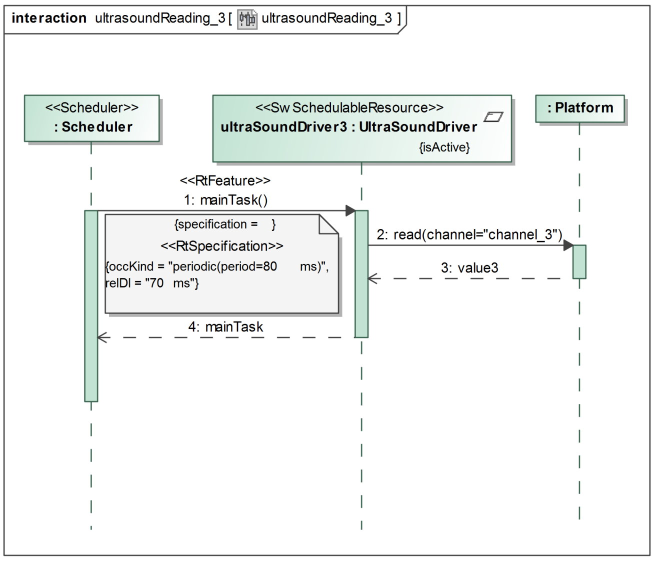

interaction ultrasoundReading_3 [ultrasoundReading_3]
<<Scheduler>>
: Scheduler
<<Sw SchedulableResource>>
ultraSoundDriver3 : UltraSoundDriver
{isActive}
: Platform
<<RtFeature>>
1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=80 ms)",
relDl = "70 ms"}
2: read(channel="channel_3")
3: value3
4: mainTask

Figure 11.17: Wheelchair System: Sequence Diagram - Ultrasound reading 4.

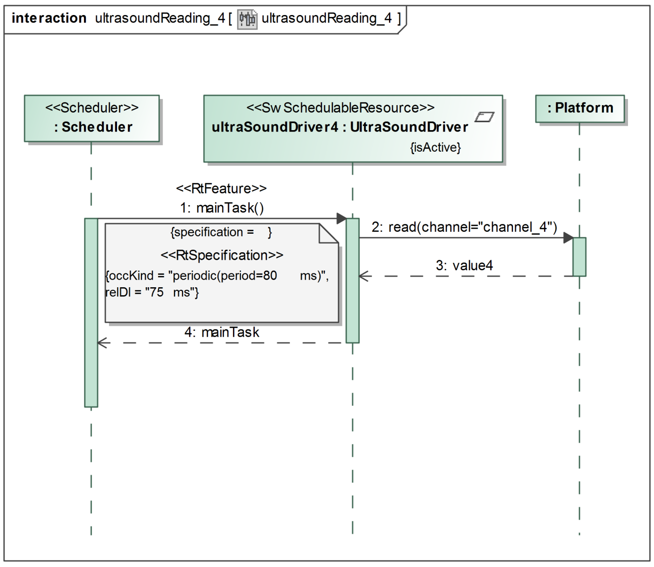

interaction ultrasoundReading_4 [ultrasoundReading_4]
<<Scheduler>>
: Scheduler
<<Sw SchedulableResource>>
ultraSoundDriver4 : UltraSoundDriver
{isActive}
: Platform
<<RtFeature>>
1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=80 ms)",
relDl = "75 ms"}
2: read(channel="channel_4")
3: value4
4: mainTask

Figure 11.18: Wheelchair System: Sequence Diagram - Verify obstacle.

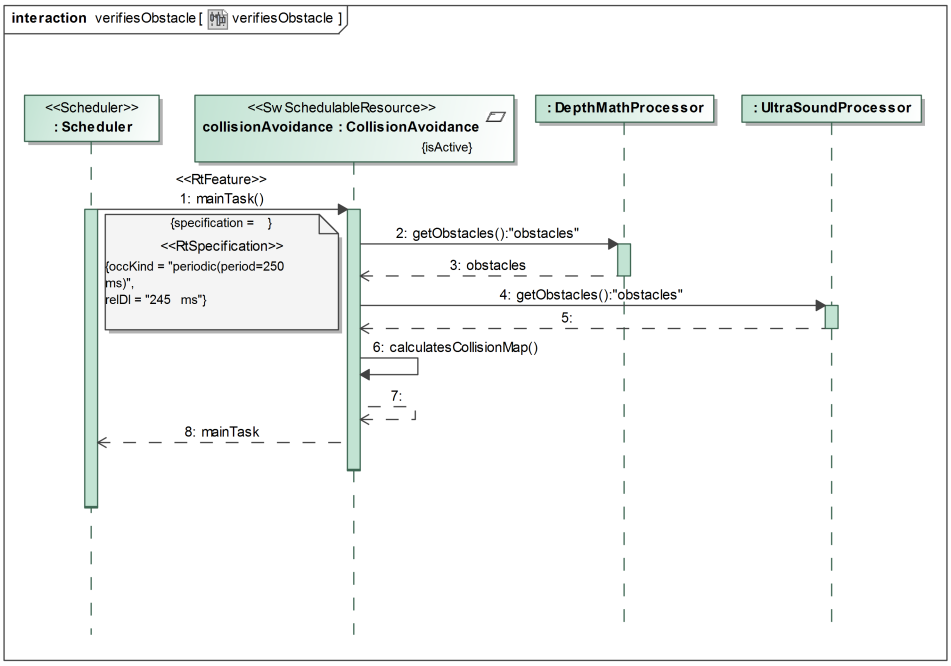

interaction verifiesObstacle [verifiesObstacle]
<<Scheduler>>
: Scheduler
<<Sw SchedulableResource>>
collisionAvoidance : CollisionAvoidance
{isActive}
: DepthMathProcessor
: UltraSoundProcessor
<<RtFeature>>
1: mainTask()
{specification = }
<<RtSpecification>>
{occKind = "periodic(period=250 ms)",
relDl = "245 ms"}
2: getObstacles():"obstacles"
3: obstacles
4: getObstacles():"obstacles"
5:
6: calculatesCollisionMap()
7:
8: mainTask

Figure 11.19: Wheelchair System: Composite structure diagram.

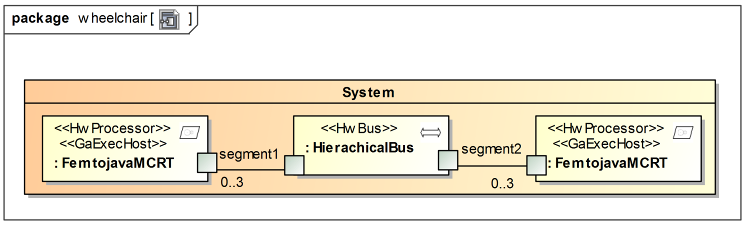


Figure 11.20: Wheelchair System: Interaction overview diagram.

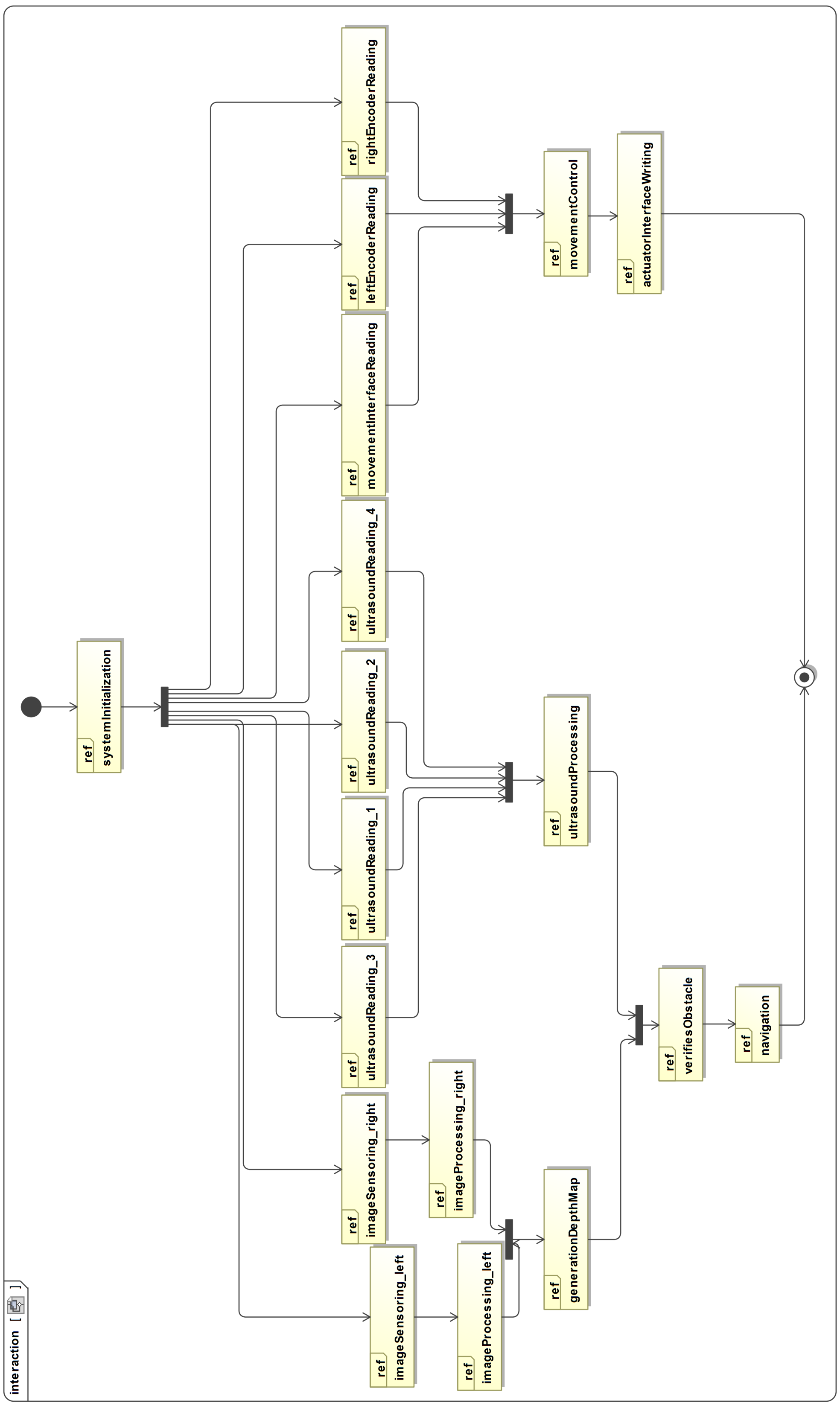
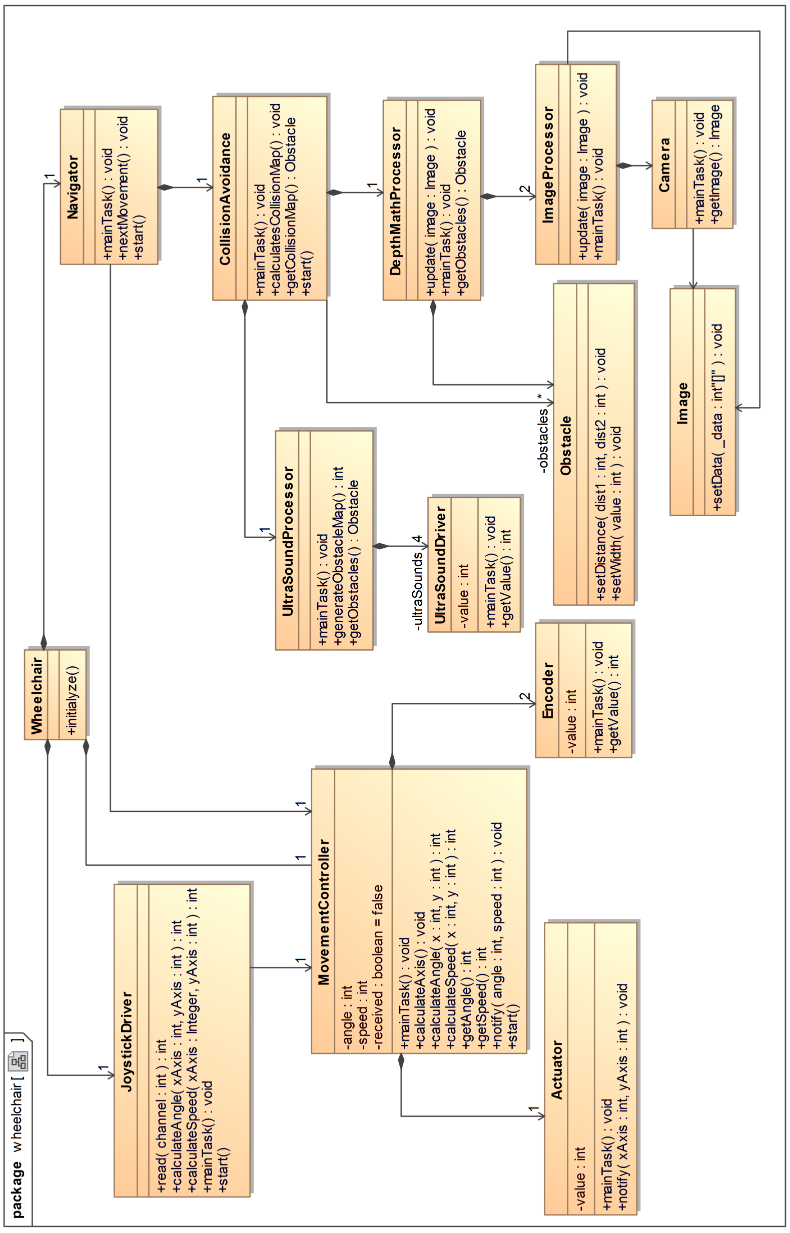


Figure 11.21: Wheelchair System: Class diagram.



LIST OF OWN PUBLICATIONS

BRISOLARA, L. B. et al. Using UML as front-end for heterogeneous software code generation strategies. In: DESIGN, AUTOMATION AND TEST IN EUROPE, New York, NY, USA. **Proceedings...** ACM, 2008. p.504–509. (DATE '08).

NASCIMENTO, F. A. M. et al. MDA-based approach for embedded software generation from a UML/MOF repository. In: INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 19., New York, NY, USA. **Proceedings...** ACM, 2006. p.143–148. (SBCCI '06).

NASCIMENTO, F. A. M.; OLIVEIRA, M. F. S.; WAGNER, F. A model-driven engineering framework for embedded systems design. **Innovations in Systems and Software Engineering**, [S.l.], v.8, n.1, p.19–33, Mar. 2012.

NASCIMENTO, F. A. M.; OLIVEIRA, M. F. S.; WAGNER, F. R. ModES: embedded systems design methodology and tools based on mde. In: INTERNATIONAL WORKSHOP ON MODEL-BASED METHODOLOGIES FOR PERVASIVE AND EMBEDDED SOFTWARE, 4., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2007. p.67–76.

NASCIMENTO, F. A. M.; OLIVEIRA, M. F. S.; WAGNER, F. R. Using MDE for the formal verification of embedded systems modeled by UML sequence diagrams. In: ANNUAL SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN: CHIP ON THE DUNES, 22., New York, NY, USA. **Proceedings...** ACM, 2009. (SBCCI '09).

NASCIMENTO, F. A.; OLIVEIRA, M. F. S.; WAGNER, F. R. Formal Verification for Embedded Systems Design Based on MDE. In: RETTBERG, A. et al. (Ed.). **Analysis, Architectures and Modelling of Embedded Systems**. Berlin, Heidelberg: Springer Boston, 2009. p.159–170. (IFIP Advances in Information and Communication Technology, v.310).

OLIVEIRA, M. F. S. et al. Early Embedded Software Design Space Exploration Using UML-Based Estimation. In: SEVENTEENTH IEEE INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2006. p.24–32.

OLIVEIRA, M. F. S. et al. Model driven engineering for MPSOC design space exploration. In: INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 20., New York, NY, USA. **Proceedings...** ACM, 2007. p.81–86. (SBCCI '07).

OLIVEIRA, M. F. S. et al. Model driven engineering for MPSOC design space exploration. **Journal of Integrated Circuits and Systems**, [S.l.], v.3, n.1, p.13–22, 2008.

OLIVEIRA, M. F. S. et al. Software Quality Metrics and their Impact on Embedded Software. In: INTERNATIONAL WORKSHOP ON MODEL-BASED METHODOLOGIES FOR PERVASIVE AND EMBEDDED SOFTWARE, 2008., Washington, DC, USA. **Proceedings. . .** IEEE Computer Society, 2008. p.68–77.

OLIVEIRA, M. F. S. et al. Exploiting the model-driven engineering approach to improve design space exploration of embedded systems. In: ANNUAL SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN: CHIP ON THE DUNES, 22., New York, NY, USA. **Proceedings. . .** ACM, 2009. (SBCCI '09).

OLIVEIRA, M. F. S. et al. High-Level Design Space Exploration of Embedded Systems Using the Model-Driven Engineering and Aspect-Oriented Design Approaches. In: GOMES, L.; FERNANDES, J. a. M. (Ed.). **Behavioral Modeling for Embedded Systems and Technologies**. [S.l.]: IGI Global, 2010. p.114–146.

OLIVEIRA, M. F. S. et al. Design space abstraction and metamodeling for embedded systems design space exploration. In: INTERNATIONAL WORKSHOP ON MODEL-BASED METHODOLOGIES FOR PERVASIVE AND EMBEDDED SOFTWARE, 7., New York, NY, USA. **Proceedings. . .** ACM, 2010. p.29–36. (MOMPES '10).

REDIN, R. M. et al. On the Use of Software Quality Metrics to Improve Physical Properties of Embedded Systems. In: KLEINJOHANN, B.; WOLF, W.; KLEINJOHANN, L. (Ed.). **Distributed Embedded Systems**: design, middleware and resources. Boston, MA: Springer Boston, 2008. p.101–110. (IFIP International Federation for Information Processing, v.271).

REFERENCES

ACCELLERA. **SystemC Language**. [S.l.]: Accellera Systems Iniciative, 2011.

ADRION, W. R.; BRANSTAD, M. A.; CHERNIAVSKY, J. C. Validation, Verification, and Testing of Computer Software. **ACM Comput. Surv.**, New York, NY, USA, v.14, n.2, p.159–192, June 1982.

ALUR, R.; DILL, D. L. A theory of timed automata. **Theoretical Computer Science**, Essex, UK, v.126, n.2, p.183–235, Apr. 1994.

AMSTEL, M. F. V.; LANGE, C. F. J.; BRAND. Metrics for Analyzing the Quality of Model Transformations. In: ECOOP WORKSHOP ON QUANTITATIVE APPROACHES ON OBJECT ORIENTED SOFTWARE ENGINEERING, 12. **Proceedings...** [S.l.: s.n.], 2008. p.41–51.

ANDERSEN, N. et al. Efficient synthesis of feature models. In: INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE, 16., New York, NY, USA. **Proceedings...** ACM, 2012. v.1, p.106–115.

ANGUS, D. Crowding Population-based Ant Colony Optimisation for the Multi-objective Travelling Salesman Problem. In: IEEE SYMPOSIUM ON COMPUTATIONAL INTELLIGENCE IN MULTI-CRITERIA DECISION-MAKING. **Proceedings...** [S.l.: s.n.], 2007. p.333–340.

ASCIA, G.; CATANIA, V.; PALESI, M. A GA-Based Design Space Exploration Framework for Parameterized System-On-A-Chip Platforms. **IEEE Transactions on Evolutionary Computation**, [S.l.], v.8, n.4, p.329–346, Aug. 2004.

ASCIA, G. et al. Performance evaluation of efficient multi-objective evolutionary algorithms for design space exploration of embedded computer systems. **Appl. Soft Comput.**, [S.l.], v.11, n.1, p.382–398, Nov. 2011.

ATITALLAH, R. B. et al. Multilevel MPSoC Performance Evaluation Using MDE Approach. In: INTERNATIONAL SYMPOSIUM ON SYSTEM-ON-CHIP, 2006. **Proceedings...** IEEE, 2006. p.1–4.

AXELSSON, J. Architecture Synthesis and Partitioning of Real-Time Systems: a comparison of three heuristic search strategies. In: INTERNATIONAL WORKSHOP ON HARDWARE/SOFTWARE CO-DESIGN, 5., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1997.

BAKSHI, A.; PRASANNA, V. K.; LEDECZI, A. MILAN: a model based integrated simulation framework for design of embedded systems. In: ACM SIGPLAN WORKSHOP ON LANGUAGES, COMPILERS AND TOOLS FOR EMBEDDED SYSTEMS, New York, NY, USA. **Proceedings...** ACM, 2001. p.82–93.

BALARIN, F. et al. **Hardware-software co-design of embedded systems**: the polis approach. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

BALARIN, F. et al. Metropolis: an integrated electronic system design environment. **Computer**, Los Alamitos, CA, USA, v.36, n.4, p.45–52, Apr. 2003.

BECK, A. C. S. et al. CACO-PS: a general purpose cycle-accurate configurable power simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 16., Los Alamitos, CA, USA. **Proceedings...** IEEE, 2003. p.349–354.

BECKER, L. B.; WEHRMEISTER, M. A.; PEREIRA, C. E. Power and performance tuning in the synthesis of real-time scheduling algorithms for embedded applications. In: INTEGRATED CIRCUITS AND SYSTEM DESIGN, 17., New York, NY, USA. **Proceedings...** ACM, 2004. p.169–174.

BERGER, A. S. **Embedded Systems Design**: an introduction to processes, tools and techniques. [S.l.]: Mcgraw-Hill Professional, 2001.

BERGMANN, G. et al. A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In: EHRIG, H. et al. (Ed.). **Graph Transformations**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p.396–410. (Lecture Notes in Computer Science, v.5214).

BEZIVIN, J. On the unification power of models. **Software and Systems Modelling**, [S.l.], v.4, n.2, p.171–188, May 2005.

BLICKLE, T.; TEICH, J.; THIELE, L. System-Level Synthesis Using Evolutionary Algorithms. **Design Automation for Embedded Systems**, [S.l.], v.3, n.1, p.23–58, Jan. 1998.

BONDé, L.; DUMOULIN, C.; DEKEYSER, J.-L. Metamodels and MDA Transformations for Embedded Systems. In: BOULET, P. (Ed.). **Advances in Design and Specification Languages for SoCs**. Boston: Springer US, 2005. p.89–105.

BONTEMPI, G.; KRUIJTZER, W. A Data Analysis Method for Software Performance Prediction. In: DESIGN, AUTOMATION AND TEST IN EUROPE, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2002. (DATE ’02).

BOULET, P. et al. MDA for SoC Design, Intensive Signal Processing Experiment. In: FORUM ON DESIGN LANGUAGES. **Proceedings...** ECSI, 2003. p.309–317. ((FDL’03)).

BRISOLARA, L. B. et al. Using UML as front-end for heterogeneous software code generation strategies. In: DESIGN, AUTOMATION AND TEST IN EUROPE, New York, NY, USA. **Proceedings...** ACM, 2008. p.504–509. (DATE ’08).

BUTTAZZO, G. C. **Hard Real-Time Computing Systems**: predictable scheduling algorithms and applications. 3.ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. (Real-Time Systems Series).

COFFLAND, J. E.; PIMENTEL, A. D. A software framework for efficient system-level performance evaluation of embedded systems. In: ACM SYMPOSIUM ON APPLIED COMPUTING, 2003., New York, NY, USA. **Proceedings...** ACM, 2003. p.666–671. (SAC '03).

CORMEN, T. H. et al. **Introduction to Algorithms**. 3.ed. Cambridge, MA, USA: The MIT Press, 2009.

CZARNECKI, K. **Generative Programming**: principles and techniques of software engineering based on automated configuration and fragment-based component models. 1998. Tese (Doutorado em Ciência da Computação) — Technical University of Ilmenau.

CZARNECKI, K.; HELSEN, S. Feature-based survey of model transformation approaches. **IBM Systems Journal**, Riverton, NJ, USA, v.45, n.3, p.621–645, July 2006.

DEB, K. et al. A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimisation: nsga-ii. In: INTERNATIONAL CONFERENCE ON PARALLEL PROBLEM SOLVING FROM NATURE, 6., London, UK, UK. **Proceedings...** Springer-Verlag, 2000. p.849–858. (PPSN VI).

DEL FABRO, M. D. et al. AMW: a generic model weaver. **Proceeedings d' 1ères Journées sur l'Ingénierie Dirigée par les Modèles**, [S.l.], 2005.

DENSMORE, D.; PASSERONE, R.; SANGIOVANNI-VINCENTELLI, A. A Platform-Based Taxonomy for ESL Design. **Design & Test of Computers, IEEE**, Los Alamitos, CA, USA, v.23, n.5, p.359–374, May 2006.

DICK, R. P.; JHA, N. K. MOGAC: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v.17, n.10, p.920–935, Oct. 1998.

DICK, R. P.; RHODES, D. L.; WOLF, W. TGFF: task graphs for free. In: SIXTH INTERNATIONAL WORKSHOP ON HARDWARE/SOFTWARE CODESIGN. (CODES/-CASHE'98). **Proceedings...** IEEE Comput. Soc, 1998. p.97–101.

DORIGO, M.; MANIEZZO, V.; COLORNI, A. Ant system: optimization by a colony of cooperating agents. **Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on**, [S.l.], v.26, n.1, p.29–41, Feb. 1996.

DOUGLASS, B. P. **Real-Time Design Patterns**: robust scalable architecture for real-time systems. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

DWIVEDI, B. K.; KUMAR, A.; BALAKRISHNAN, M. Automatic synthesis of system on chip multiprocessor architectures for process networks. In: IEEE/ACM/IFIP INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, 2., New York, NY, USA. **Proceedings...** ACM, 2004. p.60–65. (CODES+ISSS '04).

ERBAS, C.; ERBAS, S. C.; PIMENTEL, A. D. A multiobjective optimization model for exploring multiprocessor mappings of process networks. In: IEEE/ACM/IFIP INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, 1., New York, NY, USA. **Proceedings...** ACM, 2003. p.182–187. (CODES+ISSS '03).

FERNANDES, J. M.; MACHADO, R. J.; SANTOS, H. D. Modeling industrial embedded systems with UML. In: HARDWARE/SOFTWARE CODESIGN, New York, NY, USA. **Proceedings...** ACM, 2000. p.18–22. (CODES ’00).

FERRARI, A.; SANGIOVANNI-VINCENTELLI, A. System design: traditional concepts and new paradigms. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTERS AND PROCESSORS (CAT. NO.99CB37040), 1999. **Proceedings...** IEEE Comput. Soc, 1999. p.2–12.

FRANCE, R.; RUMPE, B. Model-driven Development of Complex Software: a research roadmap. In: FUTURE OF SOFTWARE ENGINEERING, 2007., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2007. p.37–54. (FOSE ’07).

FREITAS, E. P. **Metodologia orientada a aspectos para a especificação de sistemas tempo-real embarcados distribuídos**. 2007. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul.

GAJSKI, D. D. et al. **Specification and Design of Embedded Systems**. [S.l.]: Prentice Hall PTR, 1994.

GAJSKI, D. D. et al. **Embedded System Design**: modeling, synthesis and verification. 1.ed. [S.l.]: Springer, 2009.

GASEVIC, D.; DJURIC, D.; DEVEDZIC, V. **Model Driven Engineering and Ontology Development**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

GEILEN, M. Object-oriented modelling and specification using SHE. **Computer Languages**, [S.l.], v.27, n.1-3, p.19–38, Oct. 2001.

GIVARGIS, T.; VAHID, F. Platune: a tuning framework for system-on-a-chip platforms. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v.21, n.11, p.1317–1327, Nov. 2002.

GIVARGIS, T.; VAHID, F.; HENKEL, J. System-level exploration for Pareto-optimal configurations in parameterized system-on-a-chip (December 2002). **IEEE Trans. Very Large Scale Integr. Syst.**, Piscataway, NJ, USA, v.10, p.416–422, Aug. 2002.

GOERING, R. **Platform-based design**: a choice, not a panacea. Available at: `http://www.eetimes.com/story/OEG20020911S0061`, Accessed on: April, 2012.

GOMAA, H. **Designing Concurrent, Distributed, and Real-Time Applications with Uml**. 1st.ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

GRATTAN, B.; STITT, G.; VAHID, F. Codesign-extended applications. In: HARDWARE/SOFTWARE CODESIGN, New York, NY, USA. **Proceedings...** ACM, 2002. p.1–6. (CODES ’02).

GREEN, P. N.; EDWARDS, M. D. The modelling of embedded systems using HASoC. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXHIBITION. **Proceedings...** IEEE, 2002. p.752–759.

GREENFIELD, J.; SHORT, K. Software factories: assembling applications with patterns, models, frameworks and tools. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 18., New York, NY, USA. **Proceedings...** ACM, 2003. p.16–27. (OOPSLA '03).

GRIES, M. Methods for evaluating and covering the design space during early design development. **VLSI Journal on Integration**, Amsterdam, Netherlands, v.38, n.2, p.131–183, Dec. 2004.

GRUTTNER, K. et al. COMPLEX: codesign and power management in platform-based design space exploration. In: EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN (DSD), 15. **Proceedings...** IEEE, 2012. p.349–358.

HAAN, J. **The place of Architecture in Model-Driven Engineering**. Available at: `http://www.theenterprisearchitect.eu/archive/2008/11/27/the-place-of-architecture-in-model-driven-engineering`, Accessed on: April, 2012.

HABIBI, A.; TAHAR, S. Towards an efficient assertion based verification of SystemC designs. In: HIGH-LEVEL DESIGN VALIDATION AND TEST WORKSHOP, 2004. NINTH IEEE INTERNATIONAL, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2004. p.19–22. (HLDVT '04).

HEGEDUS, A. et al. A model-driven framework for guided design space exploration. In: IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), 26. **Proceedings...** IEEE, 2011. p.173–182.

HORVáTH, A. et al. Experimental assessment of combining pattern matching strategies with VIATRA2. **International Journal on Software Tools for Technology Transfer (STTT)**, [S.l.], v.12, n.3-4, p.211–230, Apr. 2010.

ITO, S. A.; CARRO, L.; JACOBI, R. P. Making Java work for microcontroller applications. **Design & Test of Computers, IEEE**, [S.l.], v.18, n.5, p.100–110, Sept. 2001.

JACKSON, E. et al. Specifying and Composing Non-functional Requirements in Model-Based Development. In: BERGEL, A.; FABRY, J. (Ed.). **Software Composition**. [S.l.]: Springer Berlin Heidelberg, 2009. p.72–89. (Lecture Notes in Computer Science, v.5634).

JACKSON, E. K.; SCHULTE, W. Model Generation for Horn Logic with Stratified Negation. In: IFIP WG 6.1 INTERNATIONAL CONFERENCE ON FORMAL TECHNIQUES FOR NETWORKED AND DISTRIBUTED SYSTEMS, 28., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2008. p.1–20. (FORTE '08).

JERRAYA, A. A. et al. (Ed.). **Embedded Software for SoC**. [S.l.]: Springer, 2003.

JOUAULT, F.; BéZIVIN, J. KM3: a dsl for metamodel specification formal methods for open object-based distributed systems. In: GORRIERI, R.; WEHRHEIM, H. (Ed.). **Proceedings of the 8th IFIP WG 6.1 international conference on Formal Methods for Open Object-Based Distributed Systems**. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2006. p.171–185. (Lecture Notes in Computer Science, v.4037).

KAHN, G. The Semantics of Simple Language for Parallel Programming. In: IFIP CONGRESS. **Proceedings...** [S.l.: s.n.], 1974.

KALAVADE, A.; LEE, E. A. Hardware/Software Co-design Using Ptolemy: a case study. In: IFIP INTERNATIONAL WORKSHOP ON HARDWARE/SOFTWARE CO-DESIGN. **Proceedings...** [S.l.: s.n.], 1992.

KANG, E.; JACKSON, E.; SCHULTE, W. An Approach for Effective Design Space Exploration. In: CALINESCU, R.; JACKSON, E. (Ed.). **Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems**. [S.l.]: Springer Berlin Heidelberg, 2011. p.33–54. (Lecture Notes in Computer Science, v.6662).

KANGAS, T. et al. UML-based multiprocessor SoC design framework. **ACM Trans. Embed. Comput. Syst.**, New York, NY, USA, v.5, n.2, p.281–320, May 2006.

KENT, S. Model Driven Engineering. In: THIRD INTERNATIONAL CONFERENCE ON INTEGRATED FORMAL METHODS, London, UK, UK. **Proceedings...** Springer-Verlag, 2002. p.286–298. (IFM '02).

KEUTZER, K. et al. System-level design: orthogonalization of concerns and platform-based design. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v.19, n.12, p.1523–1543, Dec. 2000.

KIENHUIS, B. et al. An approach for quantitative analysis of application-specific dataflow architectures. In: IEEE INTERNATIONAL CONFERENCE ON APPLICATION-SPECIFIC SYSTEMS, ARCHITECTURES AND PROCESSORS. **Proceedings...** [S.l.: s.n.], 1997. p.338–349.

KIENHUIS, B. et al. A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach. In: EMBEDDED PROCESSOR DESIGN CHALLENGES: SYSTEMS, ARCHITECTURES, MODELING, AND SIMULATION - SAMOS. **Proceedings...** Springer, 2002. p.18–37.

KOCK, E. A. D. et al. YAPI: application modeling for signal processing systems. In: DESIGN AUTOMATION CONFERENCE, Los Alamitos, CA, USA. **Proceedings...** IEEE, 2000. v.00, p.402–405.

LAHIRI, K.; RAGHUNATHAN, A.; DEY, S. Efficient exploration of the SoC communication architecture design space. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 2000., Piscataway, NJ, USA. **Proceedings...** IEEE Press, 2000. p.424–430. (ICCAD '00).

LARSEN, K. G.; PETTERSSON, P.; YI, W. Uppaal in a nutshell. **International Journal on Software Tools for Technology Transfer (STTT)**, [S.l.], v.1, n.1-2, p.134–152, Dec. 1997.

LEDECZI, A. et al. Modeling methodology for integrated simulation of embedded systems. **ACM Trans. Model. Comput. Simul.**, New York, NY, USA, v.13, p.82–103, Jan. 2003.

LI, Y. T.; MALIK, S. Performance analysis of embedded software using implicit path enumeration. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 32., New York, NY, USA. **Proceedings...** ACM, 1995. p.456–461. (DAC '95).

LIEFOOGHE, A. et al. ParadisEO-MOEO: a framework for evolutionary multi-objective optimization. In: OBAYASHI, S. et al. (Ed.). **Evolutionary Multi-Criterion Optimization**. [S.l.]: Springer Berlin Heidelberg, 2007. p.386–400. (Lecture Notes in Computer Science, v.4403).

LIEVERSE, P. et al. A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems. **The Journal of VLSI Signal Processing**, [S.l.], v.29, n.3, p.197–207, Nov. 2001.

LINEHAN, E.; CLARKE, S. An aspect-oriented, model-driven approach to functional hardware verification. **Journal of Systems Architecture**, [S.l.], v.58, n.5, p.195–208, Apr. 2012.

LU, C. et al. Performance Specifications and Metrics for Adaptive Real-Time Systems. **Real-Time Systems Symposium, IEEE International**, Los Alamitos, CA, USA, v.0, p.13+, 2000.

LUKASIEWYCZ, M. et al. Opt4J: a modular framework for meta-heuristic optimization. In: GENETIC AND EVOLUTIONARY COMPUTATION, 13., New York, NY, USA. **Proceedings...** ACM, 2011. p.1723–1730. (GECCO '11).

MARWEDEL, P. **Embedded System Design**. 1.ed. Boston, USA: Kluwer Academic Publishers, 2003.

MATTOS, J. C. et al. Design Space Exploration with Automatic Generation of IP-Based Embedded Software. In: DESIGN METHODS AND APPLICATIONS FOR DISTRIBUTED EMBEDDED SYSTEMS, New York, NY, USA. **Proceedings...** Springer, 2004. p.237–246. (IFIP International Federation for Information Processing, v.150).

MELLOR, S. J.; BALCER, M. **Executable UML**: a foundation for model-driven architectures. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

MIHAL, A. et al. Developing architectural platforms: a disciplined approach. **IEEE Design & Test of Computers**, [S.l.], v.19, n.6, p.6–16, Nov. 2002.

MOHANTY, S.; PRASANNA, V. K. Rapid system-level performance evaluation and optimization for application mapping onto SoC architectures. In: ANNUAL IEEE INTERNATIONAL ASIC/SOC CONFERENCE, 15. **Proceedings...** IEEE, 2002. p.160–167.

MOHANTY, S.; PRASANNA, V. K. A hierarchical approach for energy efficient application design using heterogeneous embedded systems. In: COMPILERS, ARCHITECTURE AND SYNTHESIS FOR EMBEDDED SYSTEMS, 2003., New York, NY, USA. **Proceedings...** ACM, 2003. p.243–254. (CASES '03).

MRAIDHA, C. et al. MDA Platform for Complex Embedded Systems Development. In: KLEINJOHANN, B. et al. (Ed.). **Design Methods and Applications for Distributed Embedded Systems**. [S.l.]: Springer US, 2004. p.1–10. (IFIP International Federation for Information Processing, v.150).

MURILLO, L.; MURA, M.; PREVOSTINI, M. MDE Support for HW/SW Codesign: a uml-based design flow. In: BORRIONE, D. (Ed.). **Advances in Design Methods from Modeling Languages for Embedded Systems and SoC's**. [S.l.]: Springer Netherlands, 2010. p.19–37. (Lecture Notes in Electrical Engineering, v.63).

NASCIMENTO, F. A. M. et al. MDA-based approach for embedded software generation from a UML/MOF repository. In: INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 19., New York, NY, USA. **Proceedings...** ACM, 2006. p.143–148. (SBCCI '06).

NASCIMENTO, F. A. M.; OLIVEIRA, M. F. S.; WAGNER, F. A model-driven engineering framework for embedded systems design. **Innovations in Systems and Software Engineering**, [S.l.], v.8, n.1, p.19–33, Mar. 2012.

NASCIMENTO, F. A. M.; OLIVEIRA, M. F. S.; WAGNER, F. R. ModES: embedded systems design methodology and tools based on mde. In: INTERNATIONAL WORKSHOP ON MODEL-BASED METHODOLOGIES FOR PERVASIVE AND EMBEDDED SOFTWARE, 4., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2007. p.67–76.

NEEMA, S. et al. Constraint-Based Design-Space Exploration and Model Synthesis. In: ALUR, R.; LEE, I. (Ed.). **Embedded Software**. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2003. p.290–305. (Lecture Notes in Computer Science, v.2855).

OLIVEIRA, M. F. S. et al. Early Embedded Software Design Space Exploration Using UML-Based Estimation. In: SEVENTEENTH IEEE INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2006. p.24–32.

OLIVEIRA, M. F. S. et al. Model driven engineering for MPSOC design space exploration. In: INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 20., New York, NY, USA. **Proceedings...** ACM, 2007. p.81–86. (SBCCI '07).

OLIVEIRA, M. F. S. et al. Model driven engineering for MPSOC design space exploration. **Journal of Integrated Circuits and Systems**, [S.l.], v.3, n.1, p.13–22, 2008.

OLIVEIRA, M. F. S. et al. Software Quality Metrics and their Impact on Embedded Software. In: INTERNATIONAL WORKSHOP ON MODEL-BASED METHODOLOGIES FOR PERVASIVE AND EMBEDDED SOFTWARE, 2008., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2008. p.68–77.

OLIVEIRA, M. F. S. et al. Exploiting the model-driven engineering approach to improve design space exploration of embedded systems. In: ANNUAL SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN: CHIP ON THE DUNES, 22., New York, NY, USA. **Proceedings...** ACM, 2009. (SBCCI '09).

OLIVEIRA, M. F. S. et al. High-Level Design Space Exploration of Embedded Systems Using the Model-Driven Engineering and Aspect-Oriented Design Approaches. In: GOMES, L.; FERNANDES, J. a. M. (Ed.). **Behavioral Modeling for Embedded Systems and Technologies**. [S.l.]: IGI Global, 2010. p.114–146.

OMG. **Technical Guide to Model Driven Architecture**: the mda guide v1.0.1. [S.l.]: Object Management Group (OMG), 2003.

OMG. **Object Constraint Language, OMG Available Specification, Version 2.0**. [S.l.]: Object Management Group (OMG), 2006.

OMG. **Unified Modeling Language (UML), Infrastructure, V2.1.2**. [S.l.]: Object Management Group (OMG), 2007.

OMG. **UML Profile for MARTE**: modeling and analysis of real-time embedded systems v1.1. [S.l.]: Object Management Group (OMG), 2011.

OMG. **Action Language for Foundational UML (Alf)**: concrete syntax for a uml action language. [S.l.]: Object Management Group (OMG), 2013.

ORTEGA, R. B.; BORRIELLO, G. Communication synthesis for distributed embedded systems. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1998., New York, NY, USA. **Proceedings...** ACM, 1998. p.437–444. (ICCAD '98).

PAULIN, P. G.; PILKINGTON, C.; BENSOUDANE, E. StepNP: a system-level exploration platform for network processors. **IEEE Design & Test of Computers**, [S.l.], v.19, n.6, p.17–26, Nov. 2002.

PERALDI-FRATI, M.-A.; DEANTONI, J. Scheduling Multi Clock Real Time Systems: from requirements to implementation. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT/COMPONENT/SERVICE-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 14., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2011.

PETRIU, D. C.; WOODSIDE, C. M. Performance analysis with UML: layered queueing models from the performance profile. In: LAVAGNO, L.; MARTIN, G.; SELIC, B. (Ed.). **UML for Real**. Norwell, MA, USA: Kluwer Academic Publishers, 2003. p.221–240.

PETTER, A.; BEHRING, A.; MüHLHäUSER, M. Solving Constraints in Model Transformations. In: PAIGE, R. (Ed.). **Theory and Practice of Model Transformations**. [S.l.]: Springer Berlin Heidelberg, 2009. p.132–147. (Lecture Notes in Computer Science, v.5563).

PIEL, E. et al. Gaspard2: from marte to systemc simulation. In: DATE'08 WORKSHOP ON MODELING AND ANALYSIS OF REAL-TIME AND EMBEDDED SYSTEMS WITH THE MARTE UML PROFILE. **Proceedings...** [S.l.: s.n.], 2008.

PIMENTEL, A. D. The Artemis workbench for system-level performance evaluation of embedded systems. **International Journal of Embedded Systems**, [S.l.], v.3, n.3, p.181+, 2008.

PIMENTEL, A. D.; ERBAS, C.; POLSTRA, S. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. **IEEE Transactions on Computers**, Los Alamitos, CA, USA, v.55, n.2, p.99–112, Feb. 2006.

PREVOSTINI, M.; GANESAN, S. Bridging the Gap Between SysML and Design Space Exploration. In: FORUM ON SPECIFICATION & DESIGN LANGUAGES. **Proceedings...** ECSI, 2006. p.389–395.

RAMMIG, F. J. **Systematischer Entwurf digitaler Systeme**. [S.l.]: Teubner, Stuttgart, 1989.

RATIONAL. **Rational Unified Process for Systems Engineering (RUP-SE) 1.1**. [S.l.]: Rational Software Corporation, 2002. (TP 165A, 5/02).

REYNERI, L. M. et al. A hardware/software co-design flow and IP library based on simulink. In: DESIGN AUTOMATION CONFERENCE, 38., New York, NY, USA. **Proceedings. . .** ACM, 2001. p.593–598. (DAC '01).

ROTHENBERG, J. The nature of modeling. In: **AI, Simulation, and Modeling**. [S.l.]: John WIley and Sons, 1989. p.75–92.

ROTHLAUF, F. **Design of Modern Heuristics**: principles and application. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. 1–4p. (Natural Computing Series).

SANGIOVANNI-VINCENTELLI, A. Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design. **Proceedings of the IEEE**, [S.l.], v.95, n.3, p.467–506, 2007.

SAXENA, T.; KARSAI, G. MDE-Based Approach for Generalizing Design Space Exploration. In: PETRIU, D.; ROUQUETTE, N.; HAUGEN, Ø. (Ed.). **Model Driven Engineering Languages and Systems**. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2010. p.46–60. (Lecture Notes in Computer Science, v.6394).

SAXENA, T.; KARSAI, G. A Meta-Framework for Design Space Exploration. In: IEEE INTERNATIONAL CONFERENCE AND WORKSHOPS ON ENGINEERING OF COMPUTER BASED SYSTEMS, 18., Washington, DC, USA. **Proceedings. . .** IEEE Computer Society, 2011. p.71–80.

SCHATTKOWSKY, T.; MUELLER, W.; RETTBERG, A. A Generic Model Execution Platform for the Design of Hardware and Software. In: MARTIN, G.; MüLLER, W. (Ed.). **UML for SOC Design**. [S.l.]: Springer US, 2005. p.63–88.

SCHATZ, B.; HOLZL, F.; LUNDKVIST, T. Design-Space Exploration through Constraint-Based Model-Transformation. In: IEEE INTERNATIONAL CONFERENCE AND WORKSHOPS ON ENGINEERING OF COMPUTER BASED SYSTEMS, 17., Washington, DC, USA. **Proceedings. . .** IEEE Computer Society, 2010. p.173–182.

SCHLICHTER, T. et al. Improving System Level Design Space Exploration by Incorporating SAT-Solvers into Multi-Objective Evolutionary Algorithms. In: IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, Los Alamitos, CA, USA. **Proceedings. . .** IEEE Computer Society, 2006. p.309–316.

SCHMIDT, D. C. Guest Editor's Introduction: model-driven engineering. **Computer**, Los Alamitos, CA, USA, v.39, n.2, p.25–31, Feb. 2006.

SCIUTO, D. et al. Metrics for design space exploration of heterogeneous multiprocessor embedded systems. In: HARDWARE/SOFTWARE CODESIGN, New York, NY, USA. **Proceedings. . .** ACM, 2002. p.55–60. (CODES '02).

SELIC, B. The pragmatics of model-driven development. **Software, IEEE**, [S.l.], v.20, n.5, p.19–25, 2003.

SEMICONDUCTORS, I. International Technology Roadmap for. **International Technology Roadmap for Semiconductors Edition 2001**. [S.l.: s.n.], 2011.

SHANDLE, J.; MARTIN, G. **Making embedded software reusable for SoCs**. Available at: `http://eetimes.com/electronics-news/4141946/Making-embedded-software-reusable-for-SoCs`, Accessed on: April, 2012.

SHIN, Y.; CHOI, K. Power conscious fixed priority scheduling for hard real-time systems. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 36., New York, NY, USA. **Proceedings...** ACM, 1999. p.134–139. (DAC '99).

SHIN, Y.; CHOI, K.; SAKURAI, T. Power optimization of real-time embedded systems on variable speed processors. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, Piscataway, NJ, USA. **Proceedings...** IEEE Press, 2000. p.365–368.

SILVA, E. T. et al. Java framework for distributed real-time embedded systems. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT AND COMPONENT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 9., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2006. p.85–92.

SILVA, E. T. et al. An Infrastructure for Hardware-Software Co-Design of Embedded Real-Time Java Applications. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 11., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2008. p.273–280.

SILVANO, C. et al. MULTICUBE: multi-objective design space exploration of multi-core architectures. In: IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2010. p.488–493.

SMITH, C. U.; WILLIAMS, L. G. Software performance engineering. In: LAVAGNO, L.; MARTIN, G.; SELIC, B. (Ed.). **UML for real**. Norwell, MA, USA: Kluwer Academic Publishers, 2003. p.343–365.

SOMMERVILLE, I. **Software Engineering**. 7.ed. [S.l.]: Addison Wesley, 2004.

SZTIPANOVITS, J.; KARSAI, G. Model-integrated computing. **Computer**, Riverton, NJ, USA, v.30, n.4, p.110–111, Apr. 1997.

TERRIER, F.; GéRARD, S. MDE Benefits for Distributed, Real Time and Embedded Systems. In: KLEINJOHANN, B. et al. (Ed.). **From Model-Driven Design to Resource Management for Distributed Embedded Systems**. Boston, MA: Springer Boston, 2006. p.15–24. (IFIP International Federation for Information Processing, v.225).

THEELEN, B.; PUTTEN, P.; VOETEN, J. Using the SHE Method for UML-Based Performance Modeling. In: VILLAR, E.; MERMET, J. (Ed.). **Proceedings of Forum on System Specification and Design Languages**. Boston, USA: Springer USA, 2004. p.143–160.

TIWARI, V.; MALIK, S.; WOLFE, A. Power analysis of embedded software: a first step towards software power minimization. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, Piscataway, NJ, USA, v.2, n.4, p.437–445, Dec. 1994.

LAVAGNO, L.; MARTIN, G.; SELIC, B. (Ed.). **UML and platform-based design**. Norwell, MA, USA: Kluwer Academic Publishers, 2003. p.107–126.

VAHID, F.; GAJSKI, D. D.; GONG, J. A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning. In: EUROPEAN DESIGN AUTOMATION, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society Press, 1994. p.214–219. (EURO-DAC '94).

VAHID, F.; GIVARGIS, T. D. **Embedded System Design**: a unified hardware/software introduction. [S.l.]: wiley, 2001.

VANDERPERREN, Y.; DEHAENE, W. From UML/SysML to Matlab/Simulink: current state and future perspectives. In: DESIGN, AUTOMATION AND TEST IN EUROPE: PROCEEDINGS, 3001 Leuven, Belgium, Belgium. **Proceedings...** European Design and Automation Association, 2006. p.93. (DATE '06).

VARATKAR, G.; MARCULESCU, R. Communication-Aware Task Scheduling and Voltage Selection for Total Systems Energy Minimization. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 2003., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2003. (ICCAD '03).

VEGA-RODRIGUEZ, M. A. Energy-aware design space exploration of embedded systems. **Journal of Systems Architecture**, [S.l.], v.59, n.8, p.601–602, 2013.

WEHRMEISTER, M. A. et al. An Object-Oriented Platform-based Design Process for Embedded Real-Time Systems. In: EIGHTH IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2005. p.125–128. (ISORC '05).

WEHRMEISTER, M. A. et al. GenERTiCA: a tool for code generation and aspects weaving. In: IEEE SYMPOSIUM ON OBJECT ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 2008., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2008. p.234–238. (ISORC '08).

WEHRMEISTER, M. A.; PACKER, J. G.; CERON, L. M. Support for early verification of embedded real-time systems through UML models simulation. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.46, n.1, p.73–81, Feb. 2012.

WEICHSEL, P. M. The Kronecker Product of Graphs. **Proceedings of the American Mathematical Society**, [S.l.], v.13, n.1, 1962.

WOLF, W. A decade of hardware/software codesign. **Computer**, [S.l.], v.36, n.4, p.38–43, Apr. 2003.

ZIVKOVIC, V. D. et al. Fast and Accurate Multiprocessor Architecture Exploration with Symbolic Programs. In: DESIGN, AUTOMATION AND TEST IN EUROPE, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2003.