



PhD-FSTC-2017-54  
The Faculty of Sciences, Technology and Communication

## DISSERTATION

Defence held on 29/09/2017 in Luxembourg

to obtain the degree of

## DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

by

**Nico HOCHGESCHWENDER**

Born on 17 June 1984 in Tübingen, (Germany)

## MODEL-BASED SPECIFICATION, DEPLOYMENT AND ADAPTATION OF ROBOT PERCEPTION SYSTEMS

### Dissertation defence committee

Dr Holger Voos, dissertation supervisor  
*Professor, Université du Luxembourg*

Dr Gerhard Kraetzschmar  
*Professor, Bonn-Rhein-Sieg University of Applied Sciences*

Dr Nicolas Navet, Chairman  
*Professor, Université du Luxembourg*

Dr Herman Bruyninckx  
*Professor, KU Leuven*

Dr Pierre Kelsen, Vice Chairman  
*Professor, Université du Luxembourg*

# **Model-Based Specification, Deployment and Adaptation of Robot Perception Systems**

Nico Hochgeschwender  
of University of Luxembourg



## Abstract

As robots are becoming ubiquitous and more capable, the need for introducing solid robot software development methods is pressing to increase robots' task spectrum. This thesis is concerned with improving software engineering of robot perception systems. The presented research employs a model-based approach to provide the means to represent knowledge about robotics software. The thesis is divided into three parts, namely research on the specification, deployment and adaptation of robot perception systems.

The first part contributes the design and development of two domain-specific languages, namely RPSL and DepSL. Those languages provide suitable notations and abstractions to enable domain experts to express, compose and explore functional, architectural and deployment design decisions of robot perception systems. The resulting models are interpretable, thus they can be used not only to communicate design decisions to stakeholders, but also to verify them in an early development stage.

The second part contributes means for deploying perception systems on real robot systems even in the presence of varying resource conditions. To this end, functional, architectural and deployment models are composed in a graph-structure. Such a graph enables not only humans, but also robots to derive implicitly defined information about their software both at design time and run time. The second part also contributes a reference architecture for deploying robot perception systems. The architecture provides a template solution for integrating not only the models required for deployment, but also all the other means required to carry out deployment.

The third part utilizes both RPSL, DepSL and the reference architecture to specify, implement and evaluate three different robot perception systems. Those are capable to satisfy changing requirements induced, for example, by the robot's tasks or environment. This is achieved by proposing algorithms which derive adaptation actions based on models and varying requirements.





## Acknowledgements

Back in 2009 I enjoyed my life, I lived in Munich, on any weekend I was either climbing or skiing and I had a interesting, well-paid job as a software engineer. However, I missed something, namely the liberty to pursue my research interests and to play with robots like long ago when I was still a student.

By chance I became aware of a PhD position in **Gerhard Kraetzschmar's** group at the Bonn-Rhein-Sieg University in Sankt Augustin, Germany. His group was involved in the EU-funded project BRICS. In BRICS the main topic was to establish a solid, robot software engineering process including models, tools and frameworks. The topic immediately raised my interest and I convinced Gerhard that I am the right person to work on the project. Thank you Gerhard for your valuable feedback on my research and for reminding me to think outside the box!

The BRICS project was an amazing experience and I had the chance to get actively involved not only in the project, but also in the European robotics research community. The PIs of the project trusted me and gave me the opportunity to lead the BRICS Component Model Task Force. The Task Force harmonized and consolidated important developments of the BRICS project. In particular I would like to thank **Davide Brugali** and **Herman Bruyninckx** not only for their support, but also for keeping me busy with new scientific challenges. I am very happy that Herman also serves as a member of my thesis committee. Thank you! I am very grateful to Davide Brugali for giving me the opportunity to serve as a guest editor for the Journal of Software Engineering in Robotics, it was an exhausting, but extremely educational exercise.

While working on the BRICS project I collaborated with many PhD students, in particular those part of the Component Model Task Force, namely **Markus Klotzbücher**, **Luca Gherardi**, **Peter Soetens**, **Sebastian Blumenthal** and many others. The intense and constructive discussions in the Task Force certainly influenced my work. In fact, I believe that without BRICS this thesis would not have been possible.

When I moved to Bonn I got the chance not only to work on the BRICS project, but also to become a member of the MAS group jointly headed by **Paul Plöger**, **Erwin Prassler** and **Gerhard Kraetzschmar**. I would like to thank all of them for running such a successful Master program and for their trust and support in all my activities ranging from teaching and research to RoboCup. Speaking of RoboCup I am very proud and thankful that I was a member of both

the RoboCup@Work and RoboCup@Home team and that I had the chance to work with so many good students. Thank you!

As a long-time member of the MAS group I had the luck to work with many great colleagues, namely **Anastassia Küstenmacher**, **Alex Mitrevski**, **Alexander Moriarty**, **Argentina Ortega**, **Azamat Shakhirmardanov**, **Björn Kahl**, **Christian Mueller**, **Frederik Hegger**, **Iman Awaad**, **Jan Paulus**, **Jose Sanchez**, **Matthias Füller**, **Maximilian Schöbel**, **Mike Reckhaus**, **Ronny Hartanto**, **Santosh Thoduka** and **Sven Schneider**. Thank you all for contributing to a productive, yet lovely atmosphere with entertaining coffee breaks and conference travels.

During all the years I shared an office with **Azamat Shakhirmardanov** and it was so much fun. We had fierce, yet constructive discussions about our research and football. On my first day in office he asked me to carry a mattress through Bonn and I knew this guy is as straight forward as I am; we became good friends. Thank you Azamat you are the best FF!

After Azamat left, I shared an office with **Sven Schneider** and again it was awesome. Sven not only tolerated my sloppy office style, but he had always time to chat about my research. I am grateful for many insightful hints, comments and criticism and by the way thanks for preventing me from using TikZ.

During all the years I often traveled to Luxembourg to meet **Holger Voos**, my main PhD advisor, who was also my Diploma thesis advisor. I am very grateful that this relationship has lasted so long and in a way we have come full circle now. Thank you Holger for your trust, patience and scientific support!

I also much appreciate the feedback on my research, support in collecting data and hospitality of **Miguel A. Olivares-Mendez**. Thank you! I would like to thank **Pierre Kelsen** for serving on my thesis committee and for providing valuable suggestions and feedback on my research. In this connection I would like to thank **Loïc Gammaitoni** for having a truly interdisciplinary scientific collaboration on model-based approaches. Thank you! Also, I would like to thank **Nicolas Navet** for serving on my thesis committee.

I want to thank **Sebastian Wrede**, **Arne Nordmann** and **Dennis Wiegand** for the inspiring collaboration on the *Robotics DSL Zoo*.

I would like to acknowledge the financial support which I received by various sources, namely by the European Commission through the FP7 projects BRICS (FP7-ICT-231940) and RoCKIn (FP7-ICT-601012), by the Graduate Institute of the Bonn-Rhein-Sieg University through a PhD scholarship and by the generously support of the Applied Sciences Institute at the Bonn-Aachen International Center for Information Technology (b-it).

I want to sincerely thank my parents **Jörg** and **Inge Hochgeschwender** for their support and for educating me by defining and achieving my own desires.

I would like to thank my parents-in-law, namely **Uschi, Günther, Gaby** and **Bernd** for their support and hospitality. In particular, I am grateful to Uschi and Gaby for the many visits to help with the kids.

Last but not least I would like to thank my wife **Nicole** for her love and for reminding me what is important in life. Thank you **Helen** and **Hanne** for enriching my life and for making me the proudest father on earth.



# Contents

<b>1. Introduction</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Problem Statement . . . . .	2
1.3. Thesis Objectives . . . . .	3
1.4. Solution Approach . . . . .	4
1.5. Contributions . . . . .	5
1.6. Outline . . . . .	6
1.7. Publications . . . . .	7
1.8. Collaborations . . . . .	9
<b>2. Model-driven Engineering in Robotics</b>	<b>11</b>
2.1. Introduction . . . . .	11
2.2. Model-driven Engineering . . . . .	12
2.3. Domain-specific Languages . . . . .	14
2.4. Domain-specific Languages in Robotics . . . . .	15
2.5. Summary . . . . .	23
<b>3. Specifying Robot Perception Systems</b>	<b>25</b>
3.1. Introduction . . . . .	25
3.2. DSL Development Process . . . . .	26
3.3. Description of Domain Examples . . . . .	32
3.4. Analysis of Architectural Views . . . . .	33
3.5. Identification of Common Concepts . . . . .	34
3.6. Formalization of Concepts . . . . .	40
3.7. Development of DSLs and Tooling . . . . .	48
3.8. Related Work and Discussion . . . . .	51
3.9. Summary . . . . .	57
<b>4. Exploring the Design Space of Robot Perception Systems</b>	<b>59</b>
4.1. Introduction . . . . .	59
4.2. Motivation . . . . .	60
4.3. Approach . . . . .	61
4.4. Case Study . . . . .	66

4.5. Related Work and Discussion . . . . .	69
4.6. Summary . . . . .	71
<b>5. Implementing Semantic Queries about Domain Models</b>	<b>73</b>
5.1. Introduction . . . . .	73
5.2. Motivation . . . . .	74
5.3. Graph-based Storage and Composition of Domain Models . . . . .	76
5.4. Semantic Querying of Domain Models . . . . .	79
5.5. Case Study . . . . .	79
5.6. Related Work and Discussion . . . . .	84
5.7. Summary . . . . .	86
<b>6. Deploying Robot Perception Systems</b>	<b>87</b>
6.1. Introduction . . . . .	87
6.2. Reference Architecture . . . . .	89
6.3. Case Study . . . . .	92
6.4. Related Work and Discussion . . . . .	99
6.5. Summary . . . . .	100
<b>7. Adapting Robot Perception Systems</b>	<b>101</b>
7.1. Introduction . . . . .	101
7.2. Task-based Adaptation . . . . .	102
7.3. Platform-based Adaptation . . . . .	113
7.4. Environment-based Adaptation . . . . .	121
7.5. Summary . . . . .	126
<b>8. Conclusion</b>	<b>129</b>
8.1. Contributions . . . . .	129
8.2. Suggestions for Future Work . . . . .	130
<b>A. Alloy Model of Perception Features</b>	<b>133</b>
<b>Bibliography</b>	<b>135</b>
<b>List of Figures</b>	<b>153</b>
<b>List of Tables</b>	<b>155</b>

# Abbreviations

<b>BNF</b>	Backus-Naur Form
<b>BRICS</b>	Best Practice in Robotics
<b>CB</b>	Capability Building
<b>DSE</b>	Design Space Exploration
<b>DSL</b>	Domain-specific Language
<b>DSML</b>	Domain-specific Modeling Language
<b>EDA</b>	Electronic Design Automation
<b>FD</b>	Functional Design
<b>GDDL</b>	Grasp Domain Definition Language
<b>GPML</b>	General Purpose Modeling Language
<b>MARTE</b>	Modeling and Analysis of Real-time and Embedded Systems
<b>MDE</b>	Model-Driven Engineering
<b>MBE</b>	Model-based Engineering
<b>MDD</b>	Model-driven Development
<b>MDSD</b>	Model-based Software Development
<b>MDA</b>	Model-driven Architecture
<b>MOF</b>	Meta Object Facility
<b>OMG</b>	Object Management Group
<b>OCL</b>	Object Constraint Language
<b>PB</b>	Platform Building
<b>PD</b>	Product Deployment
<b>PM</b>	Product Maintenance
<b>RAP</b>	Robot Application Development Process
<b>SB</b>	Scenario Building



**SBM** System Benchmarking

**SD** System Deployment

**SysML** System Modeling Language

**UML** Unified Modeling Language

**Z** Z Specification Language



# Chapter 1.

## Introduction

*“ If you believe too much you’ll never notice the flaws; if you doubt too much you won’t get started. It requires a lovely balance.”*

— Richard Hamming, 1915 – 1998

### 1.1. Motivation

With the advent of mobile, general-purpose and versatile robot platforms the development of increasingly complex applications in dynamic, unstructured environments will become a reality. Robots are being exploited to ever-expanding application domains such as service robots in households performing human-scale manipulation activities [1] [2] [3], robots in warehouses transporting goods [4] [5] [6] or robots supporting search and rescue operations [7] [8]. In those scenarios, robots are expected to robustly perform a wide variety of tasks over a long period of time even in the presence of changing requirements caused by varying environmental, task and resource conditions.

In order to realize such sophisticated applications, domain experts need to perform a knowledge-intensive process that reflects, involves and builds upon decisions from complex, heterogenous fields of research and engineering – reaching from hardware design, domains such as control, perception or planning to software engineering.

Although the latest advancements in those fields contributed significantly to the development of sophisticated applications, robots’ task spectrum remains limited to carefully engineered applications.

One of the reasons is that in robotics software engineering in particular, the challenging and interdisciplinary integration of those fields is all too often solved in an ad-hoc manner for very specific problems, where knowledge and assumptions about the robot’s software remains



**Figure 1.1.** Two general-purpose, mobile and versatile robot platforms considered in this thesis. On the left-hand side a Care-O-bot 3 [9] service robot deployed in a household environment. On the right-hand side a KUKA youBot [10] service robot deployed in an industrial environment.

implicit. Such a development approach is not sustainable and will not scale for more advanced, complex robotic applications deployed in real-world environments.

This thesis aims to contribute to improving robotics software engineering by providing the means to explicitly represent knowledge about robotics software. Such knowledge will be used both by humans and robots to analyze and reflect on robots' software both at design and run time.

## 1.2. Problem Statement

In order to perform purposeful tasks, robots need to extract knowledge about the world from the data perceived through their sensors. To do so, robot perception systems need to be equipped with a broad set of sophisticated perception capabilities interpreting the sensory data.

Consider, for example, a service robot preparing a mug of coffee (see Figure 1.1). To perform this task, perception capabilities have to provide vital information for answering questions such as *a*) where to grasp the coffee pad?, *b*) is the mug's handle within reach?, *c*) when to take out the mug? and so forth. Developing a single set of perception capabilities answering all these perception-related questions simultaneously and efficiently would result in unmanageable complexity. Hence, remarkable methods and algorithms for solving some perceptual issues in isolation have already been developed.

In order to provide those solutions, robot perception systems developers – or just domain experts – perform a creative, experimental process which yields one or more perception architectures. Such an architecture implements not only a perception capability, but also

implicitly encodes a set of design decisions made by the expert at design time. Examples are the robot platform and its sensing equipment, assumptions about the environment in which the robot will operate, the tasks the robot should perform and structural, functional, non-functional and behavioral aspects of perception capabilities, architectures and their deployment on robot platforms.

However, all too often design decisions remain implicit, which limits not only the developers ability to extend, modify and reuse perception capabilities and architectures, but also the robots ability to satisfy changing requirements. In real-world scenarios those changing requirements are a given. Here, robots are confronted at run time with varying context conditions such as environmental changes (e.g. different lighting), varying resources such as memory and energy and exceptional situations such as sensor failures.

Ignoring those varying context conditions will usually degrade the performance of perception capabilities and thus the robot's ability to successfully execute tasks. Therefore, the robot itself should be able to autonomously adapt its perception capabilities, architectures and deployments to a wide range of situations.

Implementing such an adaptive approach to robot perception remains challenging. Firstly, means to explicitly represent design decisions need to be developed. To do so, one needs to identify and formalize those concepts and their relations among them which are elementary for stating design decisions. Secondly, adaptation methods need to be developed which derive adaptation actions based on stated design decisions and observed context conditions. Those methods need to efficiently and effectively master the search space of potential solutions such that changing requirements are met and the impact of adaptation upon the system is minimized. Thirdly, both aspects need to be integrated in a fully functioning robot system such that it remains open to extensions and modifications of its constituent parts.

### 1.3. Thesis Objectives

The goal of this thesis is to establish a method and corresponding means to enable domain experts to systematically construct robot perception systems so that robots can satisfy changing requirements by themselves.

In this context, the concrete research objectives of this thesis are:

1. To formally define the constituents of robot perception systems, namely **perception capabilities**, **perception architectures** and their **deployments** on robot platforms.
2. To develop means to enable domain experts to **declaratively specify** robot perception systems.

3. To study how to **grant robots access** to those specifications so that information required for the task at hand can be **automatically derived**.
4. To study how to **autonomously adapt** robot perception systems in response to changes in the environment, in the tasks or in the available resources.

## 1.4. Solution Approach

There are two main ideas to achieve the thesis objectives. Firstly, means are required to enable domain experts to represent their knowledge and design decisions in an explicit, yet machine-readable manner. Secondly, robots are endowed with those representations in order to autonomously reflect on and modify their perception systems in response to varying requirements induced by their tasks, environment and resources.

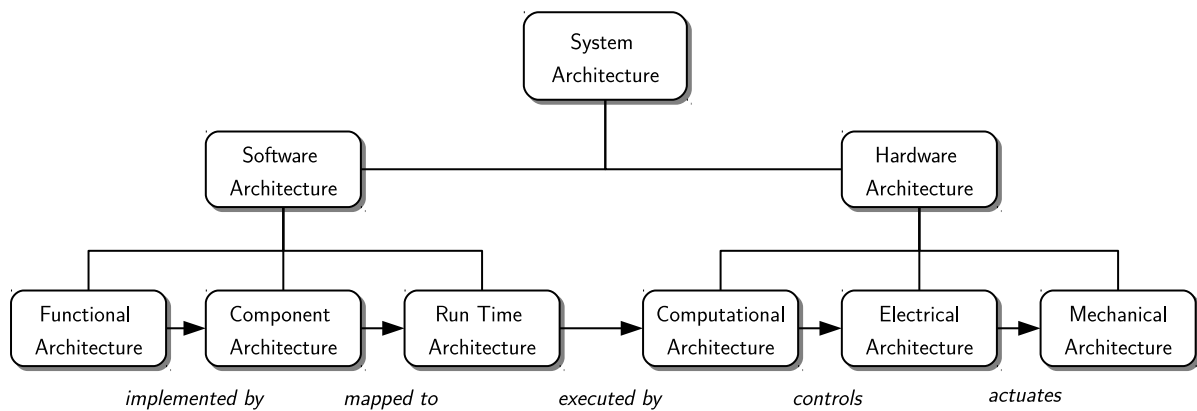
To this end, architectural views (see Figure 1.2) are introduced not only to support domain experts in analyzing and representing robot perception systems from different perspectives, but also to utilize a Model-Driven Engineering (MDE) approach where models express specific properties of a view.

In order to create those models, this thesis develops a set of Domain-specific Languages (DSLs). Those DSLs are tailored to express the concepts of a particular view, thus enabling domain experts to make their design decisions explicit. The DSL concepts as well as their rules for composing them are rigorously formalized. Such a formalization paves the way not only to construct robot perception systems in an incremental and correct manner, but also to assess – if necessary – all the possible robot perception systems that can be gained from composing different models.

In order to endow robots with those models as part of their knowledge base, this thesis utilizes labeled property graphs. The graph is used as a compact representation to compose models and to derive implicitly defined information which is revealed by the labels, properties and edges of the graph.

In order to enable robots to autonomously adapt their perception systems in the presence of varying requirements, this research introduces a reference architecture for deploying and adapting robot perception systems. The reference architecture serves as a blueprint for implementing adaptive robot perception systems and includes several reusable and application-specific components.

The effectiveness of the above mentioned means has been evaluated in the context of three applications.



**Figure 1.2.** The architectural views employed for the design and development of robot perception systems. The figure is based on the architectural views introduced in the BRICS project [11].

## 1.5. Contributions

This thesis integrates methods from the fields of Software Engineering, Artificial Intelligence and Robotics in order to develop means to enhance the specification, deployment and adaptation of robot perception systems and to advance the state-of-the-art in the corresponding scientific fields.

In summary, the main contributions of this thesis are as follows:

- The assessment of MDE approaches in robotics based on well-defined criteria providing an in-depth overview of the state-of-the-art (Chapter 2).
- The design and implementation of two specification languages, namely the Robot Perception Specification Language (RPSL) and the Deployment Specification Language (DepSL). Both languages enable domain experts to declaratively specify robot perception systems with concepts which have been rigorously formalized, thus paving the way for correct-by-construction robot perception systems (Chapter 3).
- The definition and implementation of a framework which enables domain experts to efficiently and effectively perform Design Space Exploration (DSE) of given RPSL and DepSL specifications. Having such a framework supports domain experts to compare and evaluate competing design alternatives for the same problem and is a powerful ingredient to master the complexity of robot perception systems (Chapter 4).
- The definition and implementation of labeled property graphs as simple, yet powerful means to store, compose and query models specified by possibly heterogeneous languages originating from different functional domains and development phases (Chapter 5).

- The specification, application-specific implementation and experimental evaluation of a reference architecture for adaptive robot perception systems integrating the above mentioned means (Chapters 6 and 7).

## 1.6. Outline

This thesis should be read in a linear fashion.

In order to provide some guidance to the reader the following paragraphs offer a brief summary of each chapter.

**Chapter 2** reviews the basic principles and terminology of Model-driven Engineering and associated technology. It presents the core constituents of a robotic-specific process model, namely the BRICS RAP (BRICS Robot Application Development Process) [11]. Both the BRICS RAP and core robotic functionalities described in [12] serve as evaluation dimensions to review and discuss the adoption of MDE in robotics.

**Chapter 3** proposes two domain-specific languages, namely the RPSL and DepSL. Both languages enable domain experts to declaratively specify robot perception systems. The core concepts and abstractions of RPSL and DepSL are based on a domain analysis which is also discussed. In addition, the rigorous formalization of the language concepts and constraints by employing the formal specification language Alloy is presented.

**Chapter 4** introduces a design space exploration approach. Having RPSL and DepSL at their disposal means domain experts can define the design space of robot perception systems as a combination of functional and architectural variability expressing the set of all possible implementations. In order to explore such a design space a framework which allows the automatisisation of the exploration task is proposed.

**Chapter 5** employs labeled property graphs as simple, yet powerful means to persistently store and compose domain models originating from different functional domains and software development phases. A corresponding implementation of labeled property graphs using latest graph database technology is presented as well. Using such technology allows both humans and robots to raise semantic queries in order to derive implicitly defined information.

**Chapter 6** proposes a reference architecture integrating all the means required for deploying robot perception systems. Such a reference architecture enables robots' not only to plan, but also to autonomously execute deployments even in the presence of changing resources conditions.

**Chapter 7** presents three different applications of robot perception systems. Those applications need to deal with varying requirements induced by changing task, platform



and environmental context. Each application employs not only the reference architecture proposed in Chapter 6, but also integrates the knowledge required to carry out the adaptation.

**Chapter 8** concludes this thesis with main contributions, limitations of the approach and directions of future work.

Chapters 2, 3, 4, 5, 6 and 7 contain the core contributions. Therefore, a detailed discussion of the contributions in context with the state-of-the-art is made within each aforementioned chapters.

## 1.7. Publications

Parts of this thesis have been published in journal and conference proceedings. The publications are provided in chronological order. For each publication the main corresponding chapter is given.

- Davide Brugali and Nico Hochgeschwender. **Managing the Functional Variability of Robotic Perception Systems**. In *Proceedings of the IEEE International Conference on Robotic Computing*. 2017. Taichung, Taiwan. Chapter 3.
- Nico Hochgeschwender, Sven Schneider, Holger Voos, Herman Bruyninckx and Gerhard Kraetzschmar. **Graph-based Software Knowledge: Storage and Semantic Querying of Domain Models for Run Time Adaptation**. In *Proceedings of the IEEE International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAN)*. 2016. San Francisco, USA. Chapter 5.
- Loïc Gammaitoni and Nico Hochgeschwender. **RPSL meets Lightning: A Model-based Approach to Design Space Exploration of Robot Perception Systems**. In *Proceedings of the IEEE International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAN)*. 2016. San Francisco, USA. Chapter 4.
- Arne Nordmann, Nico Hochgeschwender, Dennis Wigand and Sebastian Wrede. **A Survey on Domain-Specific Modeling and Languages in Robotics**. In *Journal of Software Engineering for Robotics (JOSER)*. Vol. 7. Nr. 1. 2016. Chapter 2.
- Jose M. S. Loza, Sven Schneider, Nico Hochgeschwender, Gerhard Kraetzschmar and Paul Plöger. **Context-Based Adaptation of In-Hand Slip Detection for Service Robots**. In *Proceedings of the 9th IFAC Symposium on Intelligent Autonomous Vehicles (IAV)*. Leipzig, Germany. 2016. Chapter 3.
- Nico Hochgeschwender, Miguel A. Olivares-Mendez, Holger Voos and Gerhard K. Kraetzschmar. **Context-based Selection and Execution of Robot Perception Graphs**. In *Pro-*

*ceedings of the 20th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Luxembourg, Luxembourg. 2015. Chapter 7.

- Luca Gherardi and Nico Hochgeschwender. **RRA: Models and Tools for Robotics Run-time Adaptation**. In *Proceedings of the 28th IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Hamburg, Germany. 2015. Chapter 6.
- Sebastian Blumenthal, Nico Hochgeschwender, Erwin Prassler, Holger Voos and Herman Bruyninckx. **An Approach for a Distributed World Model with QoS-based Perception Algorithm Adaptation**. In *Proceedings of the 28th IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Hamburg, Germany. 2015. Chapter 7.
- Luca Gherardi and Nico Hochgeschwender. **Poster: Model-based Run-time Variability Resolution for Robotic Applications**. In *Poster Proceedings of the 37th International Conference on Software Engineering (ICSE)*. Florence, Italy. 2015. Chapter 7.
- Arne Nordmann, Nico Hochgeschwender and Sebastian Wrede. **A Survey on Domain-specific Languages in Robotics**. In *Proceeding of the 4th International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAN)*. Springer Lecture Notes in Computer Science (LNCS). Vol. 8810. Bergamo, Italy. 2014. Chapter 2.
- Nico Hochgeschwender, Sven Schneider, Holger Voos and Gerhard K. Kraetzschmar. **Declarative Specification of Robot Perception Architectures**. In *Proceeding of the 4th International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAN)*. Springer Lecture Notes in Computer Science (LNCS). Vol. 8810. Bergamo, Italy. 2014. Chapter 3.
- Sven Schneider, Nico Hochgeschwender and Gerhard K. Kraetzschmar. **Structured Design and Development of Domain-specific Languages in Robotics**. In *Proceeding of the 4th International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAN)*. Springer Lecture Notes in Computer Science (LNCS). Vol. 8810. Bergamo, Italy. 2014. Chapter 3.
- Markus Klotzbuecher, Nico Hochgeschwender, Luca Gherardi, Herman Bruyninckx, Gerhard Kraetzschmar, Davide Brugali, Azamat Shakhimardanov, Jan Paulus, Michael Reckhaus, Hugo Garcia, Davide Faconti and Peter Soetens. **The BRICS Component Model: A Model-Based Development Paradigm For Complex Robotics Software Systems**. In *Proceedings of the 28th ACM Symposium on Applied Computing (SAC). Track on Software Architecture: Theory, Technology, and Applications (SA-TTA)*. Coimbra, Portugal. 2013. Chapter 3.
- Nico Hochgeschwender, Luca Gherardi, Azamat Shakhimardanov, Gerhard K. Kraetzschmar, Davide Brugali and Herman Bruyninckx. **A Model-based Approach to Software Deployment in Robotics**. In *Proceedings of the 26th IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Tokyo, Japan. 2013. Chapter 6.

- Azamat Shakhimardanov, Nico Hochgeschwender and Gerhard K. Kraetzschmar. **Component Models in Robotics Software**. In *Proceedings of the 10th Performance Metrics for Intelligent Systems Workshop (PERMIS)*. Baltimore, USA. 2010. Chapter 3.

The following publications are not covered in this thesis and have been written during my time as research assistant.

- Sebastian Zug, Tim Niemueller, Nico Hochgeschwender, Kai Seidensticker, Martin Seidel, Tim Friedrich, Tobias Neumann, Ulrich Karras, Gerhard Kraetzschmar and Alexander Ferrein. **An Integration Challenge to Bridge the Gap among Industry-inspired RoboCup Leagues**. In *Proceedings of the 20th annual RoboCup International Symposium*. Leipzig, Germany. 2016
- Francesco Amigoni, Emanuele Bastianelli, Jakob Berghofer, Andrea Bonarini, Giulio Fontana, Nico Hochgeschwender, Luca Iocchi, Gerhard Kraetzschmar, Pedro Lima, Matteo Matteucci, Pedro Miraldo, Daniele Nardi and Viola Schiaffonati. **Competitions for Benchmarking: Task and Functionality Scoring Complete Performance Assessment**. In *IEEE Robotics & Automation Magazine*. Vol. 22. Issue 3. 2015.
- Sven Schneider, Frederik Hegger, Nico Hochgeschwender, Rhama Dwiputra, Alexander Moriarty, Jakob Berghofer and Gerhard Kraetzschmar. **Design and Development of a Benchmarking Testbed for the Factory of the Future**. In *Proceedings of the 20th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Luxembourg, Luxembourg. 2015.
- Gerhard K. Kraetzschmar, Nico Hochgeschwender, Sven Schneider, Walter Nowak, Rainer Bischoff, Rhama Dwiputra, Jakob Berghofer and Frederik Hegger. **RoboCup@Work: Competing for the Factory of the Future**. In *Proceedings of the 18th annual RoboCup International Symposium*. Joao Pessoa, Brazil. 2014.
- Sven Schneider, Nico Hochgeschwender and Gerhard K. Kraetzschmar **Declarative Specification of Task-based Grasping with Constraint Validation**. In *Proceedings of the 27th IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Chicago, USA. 2014.

## 1.8. Collaborations

Parts of this thesis have been developed in collaboration with others.

**Chapter 2** is partially based on a joint effort with Arne Nordmann (Robert Bosch GmbH), Dennis Wiegand and Sebastian Wrede (both University of Bielefeld). The main goal of this effort is to structure, consolidate and harmonize MDE approaches in robotics. This effort resulted not only in two publications (see [13] and [14]), but also in an online accessible

annotated bibliography<sup>1</sup> of MDE approaches in the domain of robotics and automation. The goal of the bibliography is to foster exchange between developers of MDE approaches and potential users interested in applying MDE in robotics.

**Chapter 3** adapts a DSL development process model initially proposed by Sven Schneider and me (see Schneider *et al.* [15]). The process model is based on the master thesis of Sven Schneider which I was supervising. In order to motivate some concepts proposed in Chapter 3 a perception system developed by Jose M. S. Loza (Bonn-Rhein-Sieg University) has been employed. I discussed this system in detail and proposed some extensions which have been reported in [16].

**Chapter 4** is based on a collaboration between Loïc Gammaitoni (University of Luxembourg) and myself. The main goal of the collaboration was the conceptual and technical integration of RPSL and Lightning – a Language Workbench – for the sake of model-based design space exploration of robot perception systems.

**Chapter 7** presents three applications of some methods and tools proposed in this thesis. In order to realize and discuss those applications I collaborated with several colleagues. The first application deals with the task-based adaptation of robot perception systems and is inspired by a collaboration with Luca Gherardi (ETH Zurich). This collaboration lead to an initial implementation of some concepts and ideas described in this chapter (see also Gherardi and Hochgeschwender [17]). The second application deals with the platform-based adaptation of robot perception systems and reports the integration of RPSL with approaches for distributed world modeling as proposed by Sebastian Blumenthal (KU Leuven). The third application deals with the environment-based adaptation of robot perception system and is based on a use case specified by Miguel Angel Olivares-Mendez (University of Luxembourg) and myself.

---

<sup>1</sup><http://corlab.github.io/dslzoo/>

## Chapter 2.

# Model-driven Engineering in Robotics

*“Essentially, all models are wrong, but some are useful.”*

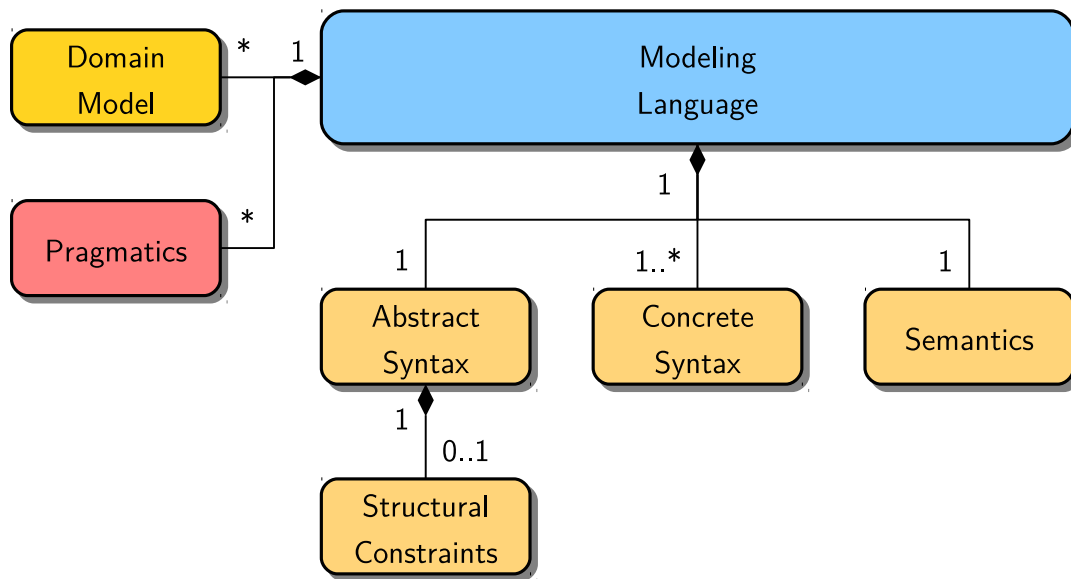
— George Box, 1919–2013

### 2.1. Introduction

Modeling denotes the process of defining, creating and modifying models. Thereby a model can be defined as an abstraction of a real system or phenomenon [18] [19]. Depending on the modeling domain and goal, models may vary in their *“level of formality, explicitness, richness in detail, and relevance”* [19]. Modeling and models are crucial constituents of every practice in science, technology, engineering and mathematics where models are used to understand, verify, simulate and visualize systems or phenomena. Without modeling and models it would be unimaginable to build bridges, to launch satellites into space and to control robots.

In order to create models, modeling languages are required to make knowledge about a system or phenomenon explicit. Depending on the domain, modeling languages might range from geometry, topology, statistics and fluid mechanics to engineering languages describing digital [20] and analog [21] circuits.

In the context of software engineering, modeling and models are fundamental ingredients of the MDE [22] [23] methodology. In MDE modeling languages – also known as Domain-specific Languages (DSLs) – are used by domain experts to specify concerns, aspects and views of software-intensive systems. Having purposeful models fosters not only the mastering of the intrinsic complexity of real-world systems – as those considered in this thesis – but also to structure the overall development process. In such a process, models are not only employed to facilitate automation (e.g. through code generation), but also used to verify and to analyze, for example, non-functional properties [23] of software systems.



**Figure 2.1.** The core ingredients of a modeling language visualized in a UML class diagram style. The figure is based on [23].

The remainder of this chapter is structured as follows. In Section 2.2 and Section 2.3 the core MDE concepts are briefly reviewed and corresponding implementation approaches are discussed. In Section 2.4 the adoption of MDE in the robotics domain is assessed. To this end, two evaluation dimensions are defined. Firstly, the process dimension defines which phases of a typical robotics system development process is facilitated and enhanced by the DSL. Secondly, the functional dimension is defined by the functional aspects that are covered by a DSL. For example, which kinds of robot system aspects such as a perception or control algorithm can be modeled using a DSL from the robotics domain. Those dimensions are then used to assess and discuss the state-of-the-art of MDE in robotics. Section 2.5 summarizes the core findings of the state-of-the-art assessment.

## 2.2. Model-driven Engineering

MDE denotes a software methodology where domain models, or just models, are considered to be the central artefacts of a software engineering process. In MDE a model is defined as *"an abstraction of a system, which may already exist or is intended to exist in the future"* [23].

In order to specify models, a modeling language is required which enables domain experts to specify domain knowledge with concepts and notations closer to the respective problem domain. This raises the level of abstraction and results not only in models that are easier to understand, but also in models which can be automatically translated (e.g. for the sake of generating code), verified and possibly executed.

The constituent parts of modeling languages and their underlying principles have been described with somehow varying terminology by approaches such as Model-based Engineering (MBE) [24], Model-driven Development (MDD) [25], Model-based Software Development (MDSO) [26] and the Model-driven Architecture (MDA) [27] standardized by the Object Management Group (OMG) [28].

In [23] Silva consolidates and summarizes those approaches by harmonizing the core ingredients of MDE as shown in Figure 2.1 and which are briefly explained in the following paragraphs.

**Abstract Syntax.** The definition of a modeling language is captured in a metamodel, also known as abstract syntax. The metamodel represents all the concepts and relations among them which are necessary to describe a certain aspect, concern or problem of a system. It is important to note that the core concepts of a modeling language represent the stable parts of a domain whereas domain models themselves can differ a lot. Depending on which approach is employed to implement modeling languages the abstract syntax is represented by different means. In the context of MDA, for example, metamodels are specified within the Meta Object Facility (MOF) [28] language ECore whereas other approaches use context free grammars like Backus-Naur Form (BNF).

**Domain Model.** Having a modeling language allows domain experts to create many different models which all conform to the same metamodel. Such an approach also allows developers to create models by reusing and for reuse.

**Concrete Syntax.** Once a metamodel is defined, a modeling language might have several notations also known as concrete syntax, such as textual or graphical representations used by the user or the computer to read and write models.

**Structural Constraints.** Solely defining an abstract syntax is usually not sufficient to implement a modeling language as the abstract syntax does not *"prevent users from creating models that violate the rules of liaison and the orchestration of its elements"* [23]. In order to prevent users from creating domain models which are prone to errors, the structural constraints of a modeling language needs to be defined. Those constraints declare invariants which should hold for certain model elements, their properties as well as their relations among each other. In order to specify the constraints there are different approaches on different levels of formality, ranging from natural language specifications to constraint and specification languages such as Object Constraint Language (OCL) [29], Z Specification Language (Z) [30] and Alloy [31].

**Semantics.** Another ingredient of modeling languages is their semantics, which can be differentiated between executable and non-executable semantics. The former reveals meaning to the order of execution-related models (e.g. state machines) whereas the latter deals with *"...concepts not directly related to software execution..."* [23] such as those found in

requirements and use case models. In order to describe the executable semantics of modeling languages it is fairly common to employ formalisms from the field of programming language theory such as operational and denotational semantics [32].

**Pragmatics.** The pragmatics of a modeling language denotes how, when and by whom a modeling language should be used [33] and is often described through case studies, guidelines and best practices. It is worth noting that such a description always depends on the context, for example, the background and expertise of language users, their modeling goals and their constraints imposed by legacy systems, tools and development process models.

In the context of MDE, another important ingredient is the concept of transformation, which enables developers to generate software artifacts, for example, source code, models, XML files or arbitrary text, from models in an automatic or semi-automatic manner. To this end, dedicated transformation languages [34] are available to support the development of model-to-model or model-to-text transformations.

Having introduced the core ingredients of modeling languages it is worth pointing out that in general two types of modeling languages are distinguished, namely General Purpose Modeling Language (GPML) and Domain-specific Modeling Language (DSML), or just DSL. The former are characterized by a larger number of generic constructs as found, for example, in the Unified Modeling Language (UML) and the System Modeling Language (SysML) whereas the latter typically comprises a smaller set of concepts and possibly graphical notations that are close to the respective application domain.

Although GPMLs are adopted by practitioners their limitation to capture domain-specific aspects is evidenced by the fact that some GPMLs provide means to add domain-specific concepts and abstractions to them. For example, the UML provides a profiling mechanism to customize UML modeling elements, for example class diagrams, through domain-specific concepts as exemplified in the Modeling and Analysis of Real-time and Embedded Systems (MARTE) [35] approach.

### 2.3. Domain-specific Languages

According to van Deursen *et al.* [36], a DSL is a "*programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular domain*". In the context of MDE, domain-specific languages have recently raised attention although they have been already used and described in the past as little languages [37]. In general DSLs can be classified in two types, namely *internal* and *external DSLs* [38].



The internal DSLs are implemented on top of general-purpose programming languages such as Ruby, Scala, Python or F#. They extend the syntax and possibly semantics [39] of their host language with notations and abstractions suitable for the particular domain. From a DSL developer perspective it is worth noting that internal DSLs reuse the hosts language infrastructure such as compilers, interpreters and integrated development environments. This enables developers that are familiar with the host language to realize internal DSLs rather quickly. Depending on the host language, different implementation patterns are available to realize internal DSLs [40]. Those patterns are related to the following DSL design principles, namely *a)* abstraction to make domain concepts explicit, *b)* absorption to integrate domain knowledge in the DSL and *c)* compression to yield a concise domain notation [40].

The external DSLs are designed and developed as standalone languages. In order to implement external DSLs several functionalities are required in order *a)* to specify the abstract and concrete syntax, *b)* to perform lexical analysis, *c)* to perform syntactic analysis, and *d)* other elementary functionalities for language development.

With the advent of language workbenches [38] developing external DSLs has been tremendously enhanced as those functionalities are integrated in ready-to-use development environments. Language workbenches such as MPS [41], Spoofox [42], MetaEdit+ [43] and others [44] enable domain experts to define, develop and compose languages. Some of those workbenches leverage advancements in mainstream IDEs such as Eclipse, for example, XText [45], whereas others are built from scratch, for example MetaEdit+ [43]. Further, some workbenches solely support the realization of textual DSLs such as XText and Spoofox [42] whereas others such as MPS [41] support the development of DSLs composed of a mixture of notations, namely textual, graphical and tabular. Although language workbenches are getting more and more used to develop real-world DSLs (see [44]) reuse of language ingredients, for example the abstract syntax, is limited due to workbench-specific representations and mechanisms.

## 2.4. Domain-specific Languages in Robotics

The design and development of robotic systems is a highly interdisciplinary and complex exercise, which requires domain experts to integrate and consolidate knowledge and domain concepts from heterogenous fields of research and engineering. Domain-specific languages and modeling approaches are therefore well-suited to lower not only the skills required to handle the complexity of robotic systems, but also to structure the overall robotic development process.

In this section, the state-of-the-art of MDE approaches in robotics is assessed both from the perspective of DSL users and developers. To this end, two evaluation dimensions are introduced. The first deals with the functional aspects that are covered by a DSL whereas the

second dimension deals with the question of which development phase is enhanced by the DSL.

### 2.4.1. Development Process Dimension

One major goal of MDE approaches is to facilitate and enhance the engineering process. However, in robotics such an engineering process does not yet exist. One reason might be that it is not very common in robotics to investigate the complete life cycle of robot applications due to the fact that there are so few applications that have been deployed in real environments over long periods of time. Within the EU-funded project Best Practice in Robotics (BRICS) a Robot Application Development Process (RAP) has been developed. The BRICS RAP [11], or just RAP, is a holistic process model for developing robot applications both in academic and industrial settings. The process model combines ideas from traditional software engineering [46], agile software development [47], model-based engineering [23] and system engineering [48] and foresees – in its latest revision – eight different phases, each of which requires several steps to complete the task. The RAP also foresees feedback and interaction among development phases and is like other process models (e.g. V-Model XT [49]), tailorable to the specific needs of all involved stakeholders (e.g. developers, customers and system integrators).

In the context of this thesis RAP is used to assess the state-of-the-art of MDE in robotics for two reasons. Firstly, RAP is one of the very few reported process models targeting robotic applications and is therefore applicable for the assessment. Secondly, RAP aims to cover the complete life cycle of robotic applications, which enables DSL users and developers to investigate whether DSLs are used to a particular extent in certain process phases. In this regard, it is worth noting that RAP has been successfully used to describe the usage of DSLs in the context of developing robot motion control software [50].

The following paragraphs briefly describe each process phase of the RAP.

**Scenario Building (SB)** deals with defining and specifying environment features, constraints and characteristics. Furthermore, the robot's task is defined. This includes the specification of customer acceptance tests to be performed in the specified and possibly generalized environment.

**Functional Design (FD)** deals with deriving general hardware requirements and top-level functionalities based on the the scenario definition. Furthermore, top-level functionalities are decomposed and dependencies among them are identified. Also an initial functional design stating which functionalities interact with each other is developed.

**Platform Building (PB)** deals with determining the robot's hardware. This includes the selection and potential configuration of robots' sensors and actuators so that they meet the requirements defined in the functional design phase.

**Capability Building (CB)** deals with constructing basic and composite components up to the application level and specifying their constraints for deployment. This also includes the specification and possibly generation of additional knowledge required for component execution such as knowledge bases and training data.

**System Deployment (SD)** deals with packaging top level component(s) into a complete application system which defines a mapping of components and composites to computational units. Furthermore, features and procedures for system launch management are developed.

**System Benchmarking (SBM)** deals with performing test procedures targeting different quality attributes such as stress testing, safety and security testing, reliability and durability testing and performance testing.

**Product Deployment (PD)** deals with tailoring an application to a specific robot system. This includes also the installation of maintenance instrumentation and a final target platform system testing.

**Product Maintenance (PM)** deals with operating and maintaining the robot application. This includes also the analysis of log files and the tuning of system parameters at run time.

Obviously the above phases include several steps and activities to achieve the task at hand. Those steps are described in more detail in the corresponding RAP documentation [11].

### 2.4.2. Functional Dimension

In order to answer the question of which DSLs can be and have been used to express functional concerns of robotic systems a functional assessment dimension will be introduced. The assessment dimension defines a classification scheme of the robotic domain into subdomains based on the Springer Handbook of Robotics [12]. More precisely, part A (*Robotics Foundations*) of the handbook is employed as a well-accepted, neutral reference which covers the fundamental principles and methods needed to create a robotic system.

The following paragraphs briefly describe each subdomain. The description includes an overview of DSLs, which can be located in that subdomain and their potential use in RAP and further details about their realization in terms of abstract syntax, concrete syntax and others. A summary of the assessment is shown in Tables 2.1 and 2.2.

**Kinematics** refers to the motion of bodies in robotic mechanisms without taking forces and torques into account. Hence, it includes general representations of the position and orientation of a body, the relation among the joints as well as conventions for representing the geometry of rigid bodies connected by joints. In [51] Frigerio *et al.* the authors present

Functional Dimension	DSL	Process Phase	MDE		
			Representation	Structural Constraints	Concrete Syntax
Kinematics	Frigerio <i>et al.</i> [51]	CB	ECore	–	Text.
	Kanayama <i>et al.</i> [52]	PB, CB	C++	–	Text.
	Jara <i>et al.</i> [53]	PB, CB	Java	–	Text. & Graph.
	De Laet <i>et al.</i> [54]	PB, CB	C++	✓	Text.
	Shakhimardanov [55]	PB, CB	C++	✓	Text.
Dynamics	Frigerio <i>et al.</i> [56]	CB	ECore	–	Text.
	Aertbeliën <i>et al.</i> [57]	CB	Lua	–	Text.
	Jara <i>et al.</i> [53]	PB, CB	Java	–	Text. & Graph.
Mechanisms & Actuation	Kitagishi <i>et al.</i> [58]	PB, CB	XSD	–	Text.
	Hornby <i>et al.</i> [59]	CB	L-System	–	Text.
	Schneider <i>et al.</i> [60]	PB, CB	ECore	✓	Text.
Sensing & Estimation	Henderson <i>et al.</i> [61]	FD, CB	EBNF	✓	Text.
	Ramaswamy <i>et al.</i> [62]	PB, CB	ECore	✓	Text. & Graph.
	Gordillo <i>et al.</i> [63]	CB	Lisp	–	Text.
Motion Planning	Feniello <i>et al.</i> [64]	FD, CB	EBNF, F#	✓	Text.
	Mackenzie <i>et al.</i> [65]	FD, CB	–	–	Text. & Graph.
	Dantam <i>et al.</i> [66]	CB	CFG	–	Text.
Motion Control	Buch <i>et al.</i> [67]	CB	BNF	–	Text.
	Thomas <i>et al.</i> [68]	CB	UML/P	✓	Graph.
Force Control	Klotzbücher <i>et al.</i> [69]	CB	Lua/ECore	✓	Text.
Reasoning Methods	Steck <i>et al.</i> [70]	CB	Lisp	✓	Text.
	Joyeux <i>et al.</i> [71]	CB	Ruby	✓	Text.

**Table 2.1.** A summary of DSLs belonging to the Kinematics, Dynamics, Mechanism & Actuation, Sensing & Estimation, Motion Planning, Motion Control, Force Control and Reasoning Methods subdomains.

a textual DSL for encoding such kinematic descriptions in order to generate C++ code implementing kinematics and dynamics algorithms. The generated code includes various matrices such as Jacobians and inertia matrices which can be used, for example, in the context of model-based controllers [51]. Similarly, serial-link structures and corresponding Denavit-Hartenberg parameters can be encoded with the Java-based DSL presented in [53]. In order to encode the kinematics of mobile robots such as differential and omnidirectional mobile bases a C++-based DSL is proposed in [52]. Each DSL mainly deals with the platform building phase and partially with the capability building phase (e.g. [51]). Although, research on robot kinematics can be considered mature, each DSL introduces slightly different representations of bodies, frames and relations. In [54] and [55] this

problem is addressed by introducing semantically enriched metamodels for topological primitives, geometric models and coordinate representations.

**Dynamics** covers the relationships between actuation and contact forces that act on robot mechanisms described by rigid bodies which are connected by joints. Furthermore, it pertains to the acceleration and motion trajectories resulting from these relationships. In order to encode these relationships and constraints (e.g. acceleration constraints) Aertbeliën *et al.* [57] developed a Lua-based DSL. The DSL allows domain experts to specify a set of prioritized, weighted constraints on the position and velocity level which are then translated into a numerical optimization problem and solved by an optimization library. The transformation step from the symbolic representation of an optimization problem to the representation of a solver (e.g. optimization library) can be considered as a model-to-model transformation. The work of Aertbeliën *et al.* [57] and other research in the dynamics domain (e.g. [56] and [53]) mainly deals with the capability building phase.

**Mechanisms and Actuation** focuses on the mechanical structure of a robot that creates its movable skeleton. All elements that cause a robotic mechanism to move – so called actuators – are addressed along with the mathematical model that is used to characterize the robot's performance. In [60] Schneider *et al.* the authors introduce a Grasp Domain Definition Language (GDDL) which enables domain experts to explicitly describe grasping problems. Part of GDDL is a DSL which allows developers to specify grasping devices such as dexterous robotic hands in terms of its physical properties (e.g. three-dimensional mesh) and semantic properties such as whether or not certain grasp types can be executed with a hand. Those properties and constraints are formalized in OCL and validated at design time and run time. From an implementation point of view, GDDL is realized as an external, textual DSL implemented with the Eclipse-based XText [45] framework. As the domain of mechanisms and actuation also implies other DSL approaches (e.g. [58] and [59]) are mainly applied during the platform building phase.

**Sensing and Estimation** ranges from robot state estimation for feedback control to task-oriented interpretation of sensor data of any kind. Apart from estimation techniques, this category also covers different kinds of information representations. According to [72], the work of Henderson and Shilcrat [61] is considered one of the first DSLs in robotics and dates back to 1984. Their DSL is part of the logical sensor systems framework and enables domain experts to specify the structure of sensor and sensor data processing components and their composition. The specifications are stored as symbolic expression and translated to the Function Equation Language [73], yielding a function graph which can be evaluated and possibly executed. A more recent work in the sensing and estimation domain is presented by Ramaswamy *et al.* in [62]. Here, an Eclipse-based graphical DSL is presented which allows domain experts to specify the component architecture of sensing and estimation systems by composing different component types such as

splitting, merging and synchronizing components. The work of Henderson *et al.* [61] and Ramaswamy *et al.* [62] focuses on structural aspects of sensing and estimation systems whereas Gordillo *et al.* [63] propose a declarative, visual language to specify complete vision verification tasks, for example to identify assembly parts, in the context of the execution of assembly plans. DSL approaches in the sensing and estimation domain are mainly employed at the functional design and capability building phase.

**Motion Planning** covers collision-free trajectory planning for mobile platforms as well as robot actuators. In [65], collision-free trajectories for a fleet of mobile robots are planned based on scenario specifications. In order to encode those specifications Mackenzie *et al.* [65] propose a DSL to specify the configuration of societies of robot systems by describing high-level goals and constraints, for example, the goal for a group of robots to maintain a certain formation. Similarly, in Finucane *et al.* [74], Linear Temporal Logic is employed to formalize task specifications which are then automatically translated in verifiable robot (motion) controllers. In [64] the authors describe a DSL to reason about object repositioning tasks in the context of a learning from demonstration framework. The main objective of the aforementioned approaches is a resulting capability (e.g. a motion planner), thus they mainly deal with the capability building phase.

**Motion Control** addresses the dynamic model of robotic manipulators, which also includes different control approaches like independent joint and torque control. Another objective in the motion control domain is the orchestration of different controllers depending on the task and environment context. In [68], a UML/P-based [75] DSL called LightRocks is proposed, which enables domain experts to compose and orchestrate motion skills represented as finite state machines. The DSL is implemented with the MontiCore language workbench [76] and experiments have been executed in the context of a KUKA LWR screwing a wooden screw into a cube. Another notable DSL, which allows domain experts to encode assembly operations in the context of small size productions, is presented in [67].

**Force Control** deals with robust and dynamic behavior of robotic systems in compliant interaction between a robot and its environment. Similar to the Motion Control domain, it includes different control aspects, for example, stiffness and impedance control. In order to support developers in the specification of force control tasks, Klotzbücher *et al.* [69] proposed an internal, Lua-based DSL. The DSL enables domain experts to specify hybrid force position/velocity control operations based on the Task-Frame Formalism [77]. The specifications are robot-independent and by employing platform-specific models, those specifications can be executed on different robot platforms such as the PR2 and KUKA LWR. Although the DSL is an internal DSL, an abstract syntax in the ECore format is provided. Interestingly, this ECore metamodel was refined in the GDDL DSL [60] (Mechanism and Actuation domain) by extending it by a units metamodel. The work of

Klotzbücher *et al.* [69] is also a good example how existing robotics knowledge is reused in the form of a DSL as the article is from 2011, whereas the underlying theory dates back to the 1980s and 1990s.

**Reasoning Methods** focus on symbol-based reasoning and knowledge representation. They cover logic as well as probability-based approaches. Furthermore, this category also addresses learning, such as inductive logic programming, neural networks, and reinforcement learning. In order to perform reasoning, learning, and planning domain experts need to represent the relevant knowledge required for the task at hand. In Artificial Intelligence, several domain specific languages have been introduced, such as PDDL [78] and ADL [79] to name just two. Some of these languages have been used in robotics to represent knowledge in the context of robot plan optimization [80], to embed geometric reasoning in action descriptions [81] and to develop an integrated robot manipulation application where task planning capabilities are required [82]. On the other hand, several DSLs in robotics have been developed which can be located in the reasoning domain. For example, Joyeux *et al.* [71] proposed a Ruby-based DSL and framework to manage and execute robot task plans and Steck *et al.* [70] proposed a Lisp-based DSL to implement situation-driven task execution. Both the work of Joyeux *et al.* and Steck *et al.* can be classified in the capability building phase and the product maintenance phase as domain models are executed and eventually modified at run time.

Although, for each of the aforementioned subdomains DSLs have been developed the majority of DSLs in robotics belong to the Architectures & Programming subdomain (see Table 2.2), which will be described in the next paragraph.

**Architectures and Programming** refers to the way a robotic system is designed at the software level. It can be divided into architectural structure and architectural style. The structure is represented by how the system is split up into sub-systems or units and how they interact with each other. In order to represent structural information about robot software architectures several authors proposed DSLs to model basic and composite robotic software components, for example, Schlegel *et al.* [83], Ortiz *et al.* [84] and Mallet *et al.* [85] to name a few. Those components usually encapsulate one or more functionalities created within the aforementioned domains. In [86], the interaction among those components is verified in order to ensure deadlock freedom of a complete robot software architecture. The aforementioned DSLs mainly deal with component architectures, whereas the work of Gherardi *et al.* [87] introduced an Eclipse-based toolchain to model both functional and component architectures of robotic systems. Here, the functional architecture is represented with feature models [88]. The architectural style addresses the underlying computational concepts. In [89] the authors introduce a graphical language and corresponding formalism to represent and analyze different architectural styles in robotics. Furthermore, the Architectures and Programming domain also deals with how

the architecture is implemented in terms of architectural units and their mapping to processes, threads and objects. Some DSLs addressing such a mapping process are proposed by Morelli *et al.* [90], Gobillot *et al.* [91] and Datta *et al.* [92]. Those DSLs are typically employed in the system deployment phase. Another important implementation concern is how to organize data and control-flow among architectural units as well as how to handle reactive and temporal events which is addressed in [93] [94] and [95]. The aforementioned DSLs are mainly used during the design phase of robot software architectures, whereas the work of [70] [96] also deals with the usage of domain models at run time in order to cope with varying requirements.

Functional Dimension	DSL	Process Phase	MDE		
			Representation	Structural Constraints	Concrete Syntax
Architectures & Programming	Ramaswamy <i>et al.</i> [97]	CB, PB	ECore	✓	Text. & Graph.
	Tousignant <i>et al.</i> [93]	CB	Silver (see [98])	–	Text.
	Schlegel <i>et al.</i> [83]	CB, SD	UML Profile	✓	Text. & Graph.
	Dhouib <i>et al.</i> [99]	CB, SD	UML Profile	✓	Text. & Graph.
	Gobillot <i>et al.</i> [91]	CB, SD	ECore	✓	Text.
	Morelli <i>et al.</i> [90]	SD	UML Profile	–	Graph.
	Datta <i>et al.</i> [92]	CB, SD	XML	–	Graph.
	Gherardi <i>et al.</i> [87]	ECore	FD, CB	✓	Graph.
	Alonso <i>et al.</i> [100]	CB, SD	ECore	✓	Text. & Graph.
	Wei <i>et al.</i> [101]	CB	EBNF	–	Text.
	Nordmann <i>et al.</i> [102]	CB, SD	MPS	✓	Text.
	Dittes <i>et al.</i> [89]	FD, CB	XML	✓	Graph.
	Fleurey <i>et al.</i> [96]	CB, SD	ECore	✓	Graph.
	Steck <i>et al.</i> [70]	PM	Lisp	–	Text.
	Rusakov <i>et al.</i> [95]	CB	Eiffel	✓	Text.
	Ortiz <i>et al.</i> [84]	CB, SD	UML Profile	✓	Text. & Graph.
	Mallet <i>et al.</i> [85]	CB, SD	EBNF	–	Text.
	Biggs <i>et al.</i> [94]	CB	Python/EBNF	✓	Text.
	Cassou <i>et al.</i> [103]	CB	DiaSpec	✓	Text.
	Abdellatif <i>et al.</i> [86]	CB, SBM	GenoM/BIB	✓	Text.

**Table 2.2.** A summary of DSLs belonging to the Architectures & Programming subdomain.



## 2.5. Summary

This chapter described the core ingredients of MDE and assessed the state-of-the-art of domain-specific modeling approaches in robotics.

In summary, DSLs are getting more and more prevalent in robotics. Although some early DSLs can be found in the 1980s [61] and early 1990s [63] [65], interest in DSLs for robotics increased during the last few years. For each subdomain, DSLs have been developed and some approaches can be assigned to two process phases. In particular, DSLs in the Architectures & Programming subdomain provide not only the means to model components, but also enable domain experts to express how those components are packaged and possibly mapped and deployed to computational units (e.g. [99], [91]). Nevertheless, very few DSLs deal with the system deployment, product deployment and product maintenance phases. In fact, the majority of DSLs support human experts in performing development activities, for example, during the capability building phase. This is not surprising as developing core robotic functionalities (e.g. sensing, control and planning) is performed within that development phase. Some functional domains are well-covered by DSLs, for example the Kinematics domain (see Table 2.1), whereas for others, for example the Sensing and Force Control domains it is still early days.

While the main target of DSLs seems to be the automation of software development (e.g. code generation [99] [51] [100]), some approaches are also used for analysis and verification (e.g. [86] [89]). From a developer perspective different implementation approaches are employed to realize DSLs, for example, UML profiles [83], external DSLs using Eclipse-based toolchains [97], language workbenches [102] and internal DSLs [54] [69]. However, to a large extent the DSLs lack a treatment of their structural constraints. Furthermore, the surveyed DSLs report little about the DSL development process itself. That is, how do DSL developers identify and consolidate abstractions which on the one hand suit the domain best and on the other hand are the building blocks of DSLs. Finally, it is worth noting that many robotic DSLs are used and evaluated in the context of real (e.g. [102] [94]) and simulated (e.g. [103]) applications even on varying hardware [69].



## Chapter 3.

# Specifying Robot Perception Systems

*“The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification.”*

— Frederick P. Brooks, 1931–\*

### 3.1. Introduction

In this chapter, two domain-specific languages are proposed, namely the Robot Perception Specification Language (RPSL) and the Deployment Specification Language (DepSL). The design and development of both languages is motivated by the pressing need of domain experts to specify their design decisions of robot perception systems in an explicit manner. To this end, both RPSL and DepSL provide domain-specific notations and abstractions supporting domain experts to state their functional, architectural and deployment design decisions of robot perception systems in the form of domain models. The provided notations and abstractions are based on architectural views representing partial aspects of a robot perception system showing specific properties. Further, both the abstract syntax and the structural constraints of RPSL and DepSL are formalized in a rigorous way which enables domain experts to verify their design decisions already in an early development stage. This contributes to a creative, yet structured design, development and deployment of robot perception systems where correct-by-construction domain models are used as first-class citizen for the sake of analysis, inspection and eventually execution.

The remainder of this chapter is structured as follows. In Section 3.2 a DSL development process is proposed. The process assembles those stakeholders and their activities and artifacts which are involved in the design and development of DSLs. The design and development of RPSL and DepSL described in this chapter were carried out using this process. Furthermore, the DSL development process defines the structure of the remaining sections. In Section 3.3 one

of many domain examples used to ground the DSL design and development is described. In Section 3.4 the domain example is analyzed and subsequently in Section 3.5 common concepts appearing in those domain examples are identified and described. In Section 3.6 those concepts are formalized in an abstract syntax (metamodel) and corresponding constraints. In Section 3.7 the implementation details of both RPSL and DepSL are discussed. Finally, related work is discussed in Section 3.8 and core results and insights are summarized in Section 3.9.

## 3.2. DSL Development Process

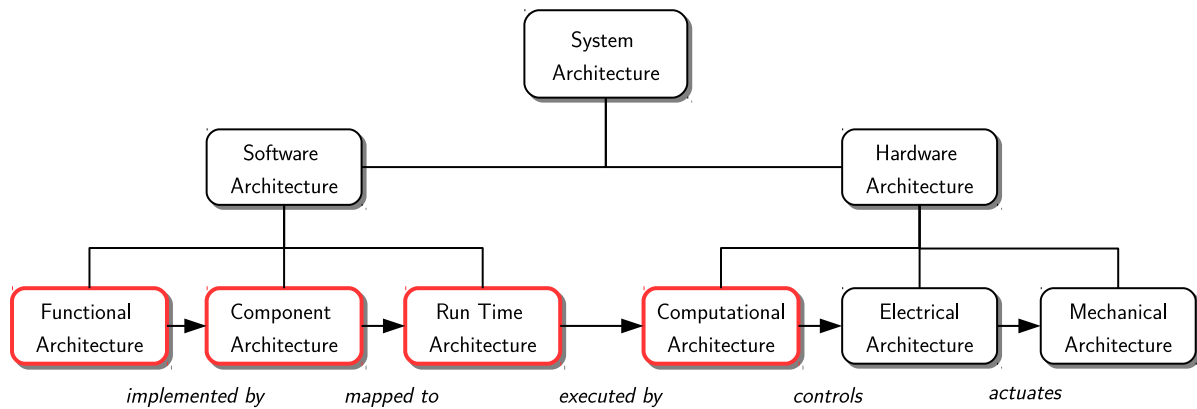
According to Mernik *et al.* [104] a DSL development process can be roughly decomposed in four major phases. Every DSL project starts with a decision to develop a DSL. Such a decision is usually justified by the needs of potential DSL users. For example, the need to automate software development activities, to support domain-specific analysis or verification [104] tasks or the need to express and specify design decisions.

The subsequent domain analysis phase investigates the vocabulary, terminology and concepts of a domain and is important in order to define the scope of DSL development. The input and output of the domain analysis varies significantly on the applied analysis approach, the maturity of the assessed domain and on the format as well as on the availability of domain knowledge.

Domain analysis approaches can be roughly classified in informal and formal approaches. The former do not follow a strict methodology and gather domain knowledge from various sources, for example, source code, technical documentation and the consultation of domain experts. In addition, the output of those approaches ranges from informal notes and requirement documents to use case diagrams. Applying somehow more formal approaches such as FODA (Feature-oriented Domain Analysis) [88] and ODE (Ontology-based Domain Engineering) [105] yields in domain descriptions consisting of terminologies and explanations of domain concepts represented in a format proposed by the analysis methodology, for example, an ontology in the case of ODE.

It is important to highlight that each domain analysis should be based on agreed upon guiding principles in order to ensure the systematic analysis of a domain. Those principles govern not only the analysis activities, but also explicitly define the scope of the analysis yielding in turn in a focused DSL development.

Based on the output of a domain analysis, a DSL still needs to be created which remains challenging as neither automated tool support nor clear guidelines exist [104]. In the subsequent DSL design phase, abstractions are identified which are the building blocks of DSLs realized in the implementation phase.



**Figure 3.1.** The different architectural views decomposed into two main categories, namely software and hardware architecture. The software architecture is further decomposed into the functional architecture implemented by the component architecture which in turn is mapped to the run time architecture executed by the computational architecture originating from the hardware architecture. This chapter deals mainly with the views highlighted in red.

In contrast to the vast majority of DSL approaches in robotics – as described in Chapter 2 – this work does not neglect to describe the DSL development process itself. To this end, a DSL development process is proposed and employed to design and develop DSLs for the robot perception systems domain. The process employs the major phases of Mernik *et al.* [104] and applies architectural views (see Section 3.2.1) as guiding principles in the domain analysis phase.

### 3.2.1. Architectural Views

Architectural views are means to analyze, design and develop the architecture of software-intensive systems – like those considered in this thesis – from different perspectives. According to the Software Engineering Body of Knowledge [106] published by the IEEE Computer Society a view “represents a partial aspect of a software architecture that shows specific properties of a software system”. For example, in [107] Kruchten proposed the “4+1” view model of software architecture where each view deals with a set of related architectural aspects. Kruchten’s physical view, for example, deals with questions such as how to map software to the hardware while taking non-functional requirements of the application into account.

In the context of this thesis, the architectural views proposed in the EU-funded BRICS project [11] are employed as guiding principles to facilitate the analysis of the assessed domain.

The BRICS project identified six architectural views (see Figure 3.1). The first three views are related to hardware and can be treated as the hardware architecture:

- The **computational architecture** view deals with information of the computational devices. This includes which types of processing units, for example CPUs and GPUs, are available, how much working memory is available, which operating system they run and how the computational devices are networked together.
- The **electrical architecture (EA)** view deals with all electrical issues of the robot system. This includes all electromechanical (e.g. actuators), electrical (e.g. batteries, switches, and so forth), electronic (e.g. sensors, circuit boards, and so forth) and computational (e.g. microcontrollers, PCs, and so forth) elements and all the wiring (e.g. power supply, buses, and so forth) between them.
- The **mechanical architecture** view deals with all the mechanical aspects of a robot system. This includes CAD models of each component, how and where they are connected to each other during assembly, and additional information about the mechanical components or the overall system that may be of relevance for software development. For example, the color and shape of system parts is of interest to detect when robot parts get into the field of view of the robot's own perception system.

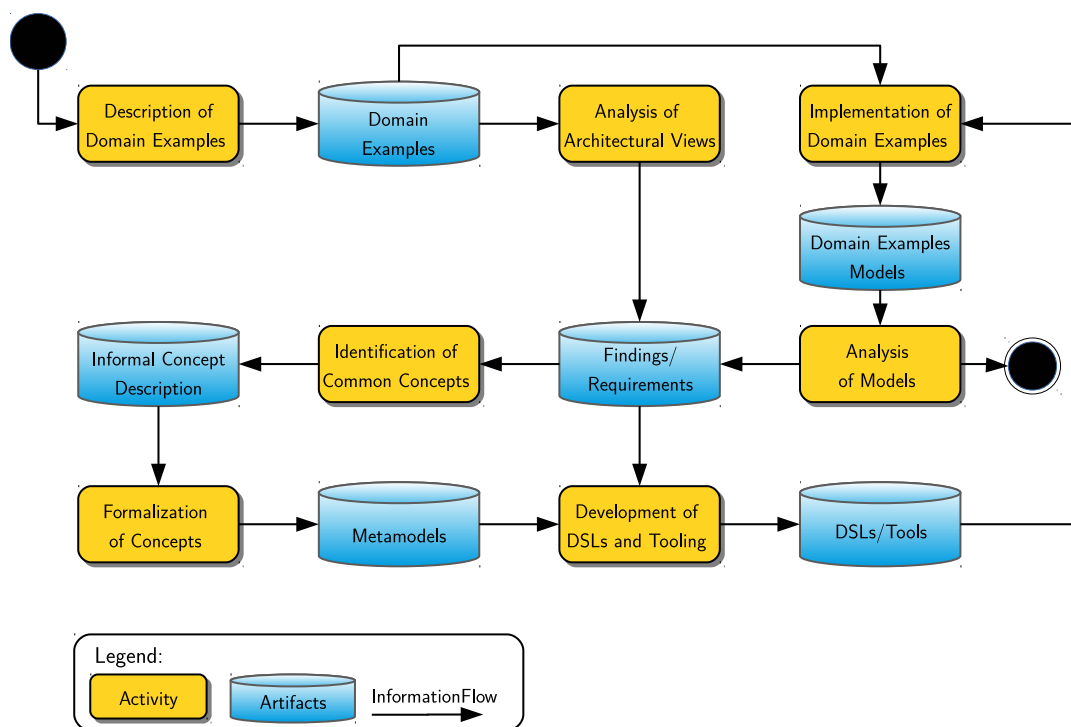
With respect to the BRICS RAP process model (see Chapter 2), information about the hardware architecture is determined in the platform building phase. The remaining three views are related to the software and can be jointly considered as the software architecture (see Figure 3.1):

- The **functional architecture** view deals with general functionality rather than software and implementation issues. The view deals with major functional components and how they interact in order to solve certain tasks. Although a function can be provided solely by a hardware module, for example, a sensing, they often have a software equivalent, for example, software drivers. Thus, the functional architecture belongs to the software architecture view.
- The **component architecture** view deals with all concerns of the actual software implementation of the functional architecture, mainly software components and their interaction and interfaces plus the relevant data structures.
- The **run time architecture** view maps the component architecture onto a particular computational architecture, mainly by mapping components onto processes and threads and by mapping processes and threads onto the available computational devices.

With respect to the BRICS RAP process model (see Chapter 2), information about the functional architecture is determined in the functional design phase, information about the component architecture is determined in the capability building phase and information about the run time architecture is determined in the capability building and system deployment phase.

### 3.2.2. Process Overview

An overview of the DSL development process proposed in this thesis is depicted in Figure 3.2. In fact, the DSL development process is based on a reverse-engineered process model presented in [15]. Here, insights gained from the development of the Grasp Domain Definition Language (GDDL) [60] – a textual DSL for the specification of grasping knowledge – where reverse-engineered into a process model. Although the GDDL language is targeted towards another domain – than the one considered in this work – the DSL development process is generic, yet tailorable to meet the requirements of other DSL developments. To this end, architectural views have been integrated as guiding principles in the domain analysis phase to accommodate the needs of the robot perception systems domain.



**Figure 3.2.** Diagram of the DSL development process composed of activities, artifacts and information flow. The activities are executed by different stakeholders involved in the DSL development.

### 3.2.3. Process Concepts

The process involves three types of stakeholders, which have different experiences and background knowledge in, for example, developing robotic applications. Each stakeholder contributes to the development process during different phases.

**Domain Experts** want to solve a problem in their domain and see the potential that a DSL in one way or another improves the solution or simplifies the task. Thus, they are also the

main users of the developed DSL. While the domain experts have a sound understanding of their domain, they are not necessarily software developers.

**Architecture Experts** have very good knowledge about one or more architectural views (see Section 3.2.1) which are related to the domain experts' problem. Thus, in certain phases, they contribute knowledge about those views to the DSL development. Very often those experts are also knowledgeable about other parts of a robotic system such as navigation, manipulation or planning and they keep the robots' task and environment in mind.

**DSL Developers** are the software developers in the process and usually do not have a background in the previous domains. Therefore, they cooperate with the aforementioned experts, to get an insight into their domains, but also to extract the relevant domain knowledge. The developers then formalize this knowledge into metamodels. Based on these metamodels, they design and implement DSLs and corresponding tooling.

It is worth noting that in small DSL projects often these roles are taken on by the same person. However, in more complex project settings, different and potentially multiple persons represent the stakeholders. The interaction between these stakeholders in different project phases forms the DSL development process, which consists of the concepts described in the following paragraphs.

**Activities** represent work that is performed by one or more of the stakeholders to solve a task. Usually, activities require input from previous process phases and produce output for other activities.

**Artifacts** are the output of an activity. Depending on the activity, artifacts include developed software or textual and graphical documents which describe, for example, use cases, requirements or reviews. Each artifact is stored in a repository.

**Models** are a special type of an artifact in the DSL development process, which represents formalized domain knowledge.

#### 3.2.4. Process Flow Phases

In the following paragraphs the process activities depicted in Figure 3.2 are explained.

**Description of Domain Examples.** The domain expert describes several domain examples or use cases which demonstrate *a)* the diversity and variability and *b)* the commonalities in the domain of his/her problem. The resulting domain examples are stored in a repository for further analysis in the next activity.

**Analysis of Architectural Views.** In the next phase, the domain expert in cooperation with the architecture expert, investigates each domain example from the perspective of an architectural view (see Section 3.2.1). One outcome of this investigation is a set of findings



representing an insight. For example, which type of laser scanner or bus system is used in the domain example. Another outcome is a set of requirements related to the DSL itself and its corresponding tooling. For example, whether a textual DSL is preferred over a graphical DSL or which keywords of the language shall be highlighted in the editor.

**Identification of Common Concepts.** This activity is handled only by the architecture expert who is knowledgeable about different architectural views. The expert investigates the findings which she/he gets from the repository and derives and eventually generalizes concepts. A concept represents a commonality or category within an architectural view. For example, the concept of a sensor is a generalization of the laser scanner findings described in the above mentioned paragraph. After the architecture expert has identified the concepts, she/he documents them, so that a common terminology emerges. The resulting informal descriptions are stored in a repository. For each concept, they contain elements such as, a lexicographic explanation or an exemplifying diagram, a graphical or textual ontology which represents the relation to other concepts, a list of known limitations or examples of this concept.

**Formalization of Concepts.** In this phase the DSL developer receives the informal descriptions and transforms them to formal metamodels (see Chapter 2). While the DSL developer performs the main work in this phase, she/he also receives support from the domain and architecture expert, for example, to clarify ambiguities in the informal description. During the formalization, the DSL developer selects one of the following approaches:

- **Definition.** New metamodels are defined, if no proper metamodel exists. This will usually be the case for the core metamodels of a domain.
- **Refinement.** Existing metamodels are refined, in case there are metamodels from related or previous projects which already cover the architectural view partially.
- **Reuse.** If a metamodel for an architectural view already exists, for example, as part of a standard, or from another project, it is reused unchanged.

**Development of DSLs and Tooling.** Only the DSL developer is responsible for the tooling development. The input to this activity are, on the one hand, the metamodels from the metamodel repository and, on the other hand, the requirements from the findings/requirements repository. The metamodels form the basis of the textual DSL's grammar or the items visualized in a graphical editor. Based on the requirements, the type, layout, structure and constraints of the editor are specified. The developed tools, such as the editors for specifying domain models and also – if required – the application-specific generators for generating, for example, source code are stored in the tool repository.

**Implementation of Domain Examples.** In this phase, the domain expert and the architecture expert implement their domain examples with the developed tools. The results are domain models which are then either input to *a*) runtime components directly as configu-

ration or *b*) code generators (which are also part of the tools repository) for generating artifacts, such as code, deployment or configuration files.

**Analysis of Models.** In the final phase, the domain and architecture experts investigate the modeled domain examples and the created tools. If the models describe the domain examples sufficiently well and the experts are satisfied with the tools, the process terminates. Otherwise, deficits, missing functionality or newly found concepts and requirements are identified and serve as input for subsequent iterations.

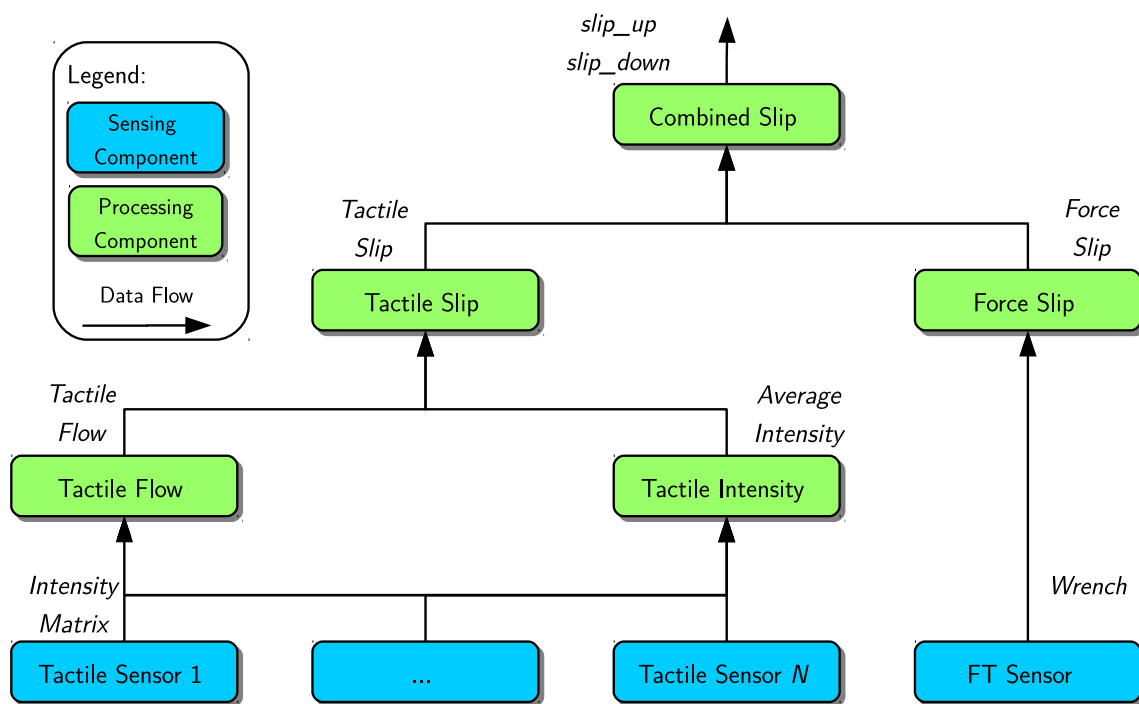
In the following sections the proposed DSL development process is executed. Section 3.3 describes a representative domain example which is used as a running example throughout this chapter. Section 3.4 analyzes the domain example from the perspective of the BRICS architectural views (see Section 3.2.1) in order to retrieve architectural findings. Those findings serve as input to derive common concepts (see Section 3.5) which in turn are formalized (see Section 3.6) yielding in metamodels. The implementation of DSLs conforming to those metamodels is described and discussed in Section 3.7.

### 3.3. Description of Domain Examples

The domain example is taken from the field of service robotics. Those robots are getting more and more deployed both in domestic and industrial environments in order to perform a wide range of tasks. In tasks such as object picking, placing and human-robot handovers in-hand slippage is inherent, i.e. a grasped object moving within the robot's grasp. Therefore, robots' ability to detect slippage is crucial for executing manipulation tasks successfully.

In order to detect in-hand slippage on the Care-O-bot 3 service robot (see Figure 1.1), Sanchez proposed in [108] three different types of slip detectors based on tactile (exteroceptive) and force (proprioceptive) measurements and as a fusion of these. The force slip detector assumes a slip occurs whenever a force is exerted in the right direction (e.g. downwards with respect to the grasp frame). The tactile slip detector employs the algorithm introduced in [109] to estimate the tangential force on the sensor caused by a sliding pressure (e.g. a grasped object slipping). The combined slip detector fuses both slip signals from the tactile slip detector and the force slip detector in a rule-based manner where experimentally obtained threshold values for the tactile and force slip detector are compared with each other.

The domain example described above serves as a running example throughout this chapter. It is worth noting that further domain examples have been assessed in this work, for example, a people detection approach [110] based on 3d data provided by a RGB-D camera, an object categorization approach [111] and basic object detection, recognition and tracking approaches [112] [113] for domestic and industrial service robots. Those examples were used



**Figure 3.3.** Schematic visualization of the in-hand slip detection system proposed by the domain expert Sanchez [108].

to identify architectural findings and to generalize common concepts in the domain of robot perception systems.

### 3.4. Analysis of Architectural Views

A schematic diagram of the three slip detectors is shown in Figure 3.3. Analysing the diagram, consulting the domain expert and assessing the corresponding implementation leads to the findings described in the following paragraphs.

From a computational architecture point of view:

- The sensing modalities available on the Care-O-bot 3 are a set of tactile sensors and one force/torque sensor (**F1**<sup>1</sup>).

From a functional architecture point of view:

- Each slip detector implements the same functionality to detect whether an in-hand slip occurs or not. The notable difference is how each slip detector represents a slip. On the one hand the combined slip detector represents a slip on a symbolic level and on the other hand both the tactile and force slip detector represent a slip on a numeric level (**F2**).

<sup>1</sup>From now on **Fx** will be used to enumerate an architectural finding.

- The combined slip detector depends on the output provided by both the force and tactile slip detector (F3).
- As experimentally validated in [16] the performance of each slip detector is context-dependent. The performance depends on the action the robot performs. The tactile slip detector, for example, recognizes a slip whenever an object is grasped whereas the force slip detector detects actual slips very accurately. Nevertheless, the force slip detectors performance is poor when no slippage occurs and the robot moves its base (F4).

From a component architecture point of view:

- One can distinguish each slip detector in its integral parts, namely sensing and processing components connected by directed edges denoting flow of data (F5).
- The sensing components only provide data whereas processing components compute and produce data (F6).
- The data produced and consumed by the components spans multiple level of abstractions. The data ranges from raw sensor data (e.g. wrench  $x \in \mathbb{R}^6$ ) and intermediate results (e.g. average intensity  $x \in \mathbb{R}^n$ ) to symbolic data (e.g. slip\_up) (F7).

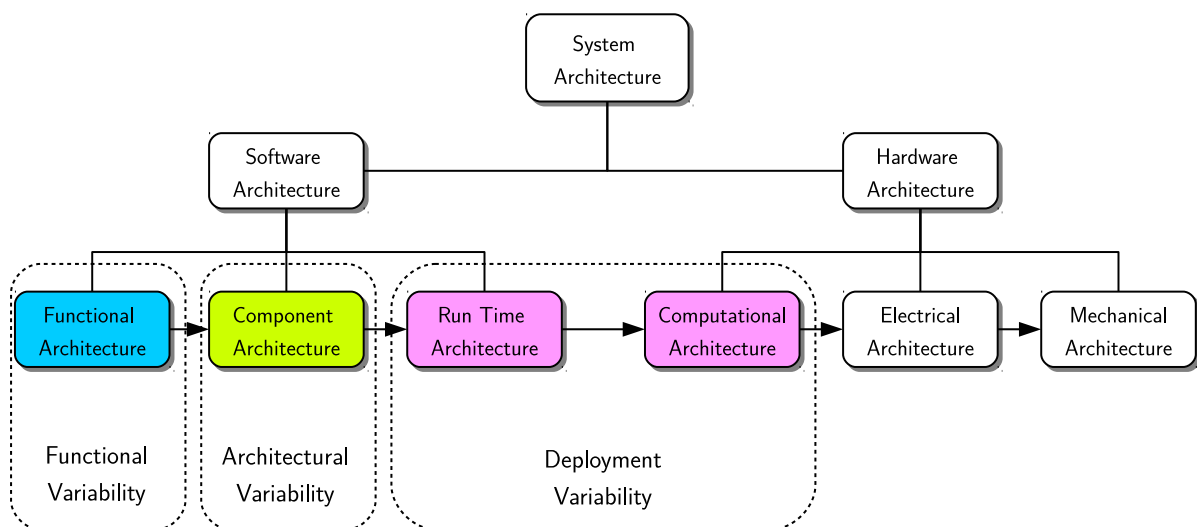
From a runtime architecture point of view:

- In order to minimize latency and to ensure a fast response of the slip detectors, both tactile and force slip detectors should be deployed on the same computer where the sensors are connected to. For example, on the Care-O-bot 3 three computational platforms are available (F8).

It is important to emphasize that each finding (F1–F8) correlates to one or more design decisions made by domain experts. Very often those design decisions are implicitly represented, hidden and scattered in technical documentation, source code, configuration files or even worse remain in the head of the expert. Which in turn is a major cause for the inflexibility of today's robot perception systems as if any of the implicit assumptions is changing, the task to adapt the system remains challenging and is prone to errors.

### 3.5. Identification of Common Concepts

Before common concepts of robot perception systems are identified it is worth noting that those systems are characterized by a huge amount of variability. The variability is caused by the different and evolving environmental, task and platform requirements making the design and development of robot perception systems such a challenging exercise. Resolving those requirements is usually done by making concrete design decisions yielding a selection of one or more variants originating from functional, architectural and deployment variability dimensions (see Figure 3.5). For example, in order to satisfy the requirement of avoiding



**Figure 3.4.** The relation between variability dimensions and architectural views.

slippage of grasped objects one needs to select between three different functional variants, namely the force, tactile and combined slip detector (see Section 3.3).

Although different variability dimensions should be completely orthogonal, i.e. they can vary independently, selecting one variant over another requires a deep understanding of the properties of the variant as they might influence each other. For example, finding **F3** (see Section 3.4) implies that selecting the combined slip detector requires also to select the force and tactile slip detector as the combined one depends on them.

Thus, a MDE approach – as the one suggested in this thesis – should not only foster modeling of variability dimensions in an orthogonal manner, but should also enable domain experts to make affects among and withing variability dimensions explicit.

This requires to identify which architectural views logically belong to which variability dimension. To this end, the functional, architectural and deployment variability dimensions are arranged with the architectural views (see Figure 3.5) introduced in Section 3.2.1. Here, the functional variability is covered by the functional architecture, the architectural variability is covered by the component architecture and the deployment variability is covered both by the run time and the computational architecture.

In the following sections the concepts appearing in those dimensions are introduced and discussed.

### 3.5.1. Functional Variability

The core concept to describe functional architectures of robot perception systems are perception features introduced in the following paragraphs.

**Perception Features** encode higher-level knowledge about functionalities capable to achieve some perception-related task (e.g. detecting a slipping object). Similar to generic features proposed in the Feature-oriented Domain Analysis [88] approach, perception features, or just features, are independent of implementation attributes. From a modeling perspective, perception features are arranged in a tree structure where an edge between parent and child feature represents a containment relation. Two kinds of containments are available, namely aggregation where a parent feature is made of child features (AND-relation) and specialization where child features are possible variants (OR-relation) of the parent feature (see finding **F2** in Section 3.4). It is important to note that the term feature originates from the software engineering domain whereas in robotics one would use the term capability [114] to describe the same thing.

In order to encode dependencies and incompatibilities among perception features the concept of integrity constraints is introduced (see finding **F3** in Section 3.4). Two kinds of integrity constraints are available, namely requires for modeling dependencies and excludes for modeling incompatibilities among perception features.

Obviously, a tree of perception features expresses – at a large scale – the complete set of available perception features. Selecting exactly those features required to meet the requirements of a certain application still remains the task of the domain expert. In Chapter 4 an approach to support domain experts in finding and selecting variants is presented. The set of selected features is valid if and only if it does not violate the imposed constraints.

In the context of the domain example one could express the tactile and force slip detectors as features which are specializations of a rather generic slip detection feature (see Section 3.3). In addition, one could express a dependency from the combined slip detection feature to the tactile and force slip detection features.

### 3.5.2. Architectural Variability

The core concepts involved in modeling the architectural variability dimension are components and perception graphs which are described in the following paragraphs.

**Components** are the basic architectural building blocks of robot perception systems. Here, a component encapsulates a functionality and restricts the access to that functionality via explicitly defined interfaces [115]. Component-based development fosters not only a structured design, but is also nowadays the predominant software implementation approach in robotics [116]. Many diverse component-oriented robotic software frameworks such as ROS [117], Orocos [118], OpenRTM [119], GenoM [120] [85] and so forth exist. Those frameworks differ mainly in how component concepts, for example ports,

interfaces and so forth, are mapped to concrete programming language primitives and implementation-specific attributes [121].

Many domain experts use component-oriented robotic software frameworks in their daily work and are capable to express their system designs in a component-oriented manner. The architectural sketch, for example, shown in Figure 3.3 includes the concept of a component as a system constituent and expresses which type these components may be, which components are connected to each other and which types these connections may be.

In the context of this work, two types of components are distinguished, namely sensor and processing components. Sensor components are used to model exteroceptive and proprioceptive sensors such as cameras, laser scanners, force sensors and so forth whereas processing components are solely used to model computational components realizing perception-related functions such as filters, feature descriptors and so forth (see also finding **F5** and **F6** in Section 3.4). As sketched in Figure 3.3 components interact with other components. The concepts making this interaction possible include ports, interfaces, data types and connections. A port is the software equivalent to the concept of a connector in hardware and are components' communication end-points for its connections to other components. Ports are typed (see Section 3.5.5) and play an important role in component-based design as they enable developers to provide several functionally different interfaces and to constrain their use to well-defined entities that will be connected to a port.

In this work a port is by definition a data-flow port having a name, type and an interface for reading and writing data. Via this interface, the port can only communicate information with data semantics to and from other components' ports; the interaction is supposed to not directly influence control flow on both the sender and the receiver side. This implies that ports are not employed for configuration concerns. Hence, the domain expert needs to specify (see Section 3.5.5) all components together with particular configuration values. This approach fosters the modeling of feasible configuration values for components (e.g. camera resolution, filter thresholds and so forth), as the domain expert is led to provide them. This also provides the possibility – if required at a later development stage – to reduce the design space (e.g. through grouping components).

**Perception Graphs** are reusable architectural units realizing specific perception features. For example, the sensor component *FT Sensor* and processing component *Force Slip* shown in Figure 3.3 form together a perception graph realizing the force slip detection feature. Both components are linked by connections providing the actual wiring between ports of different components. That is, while a port is a component-level mechanism to make a particular component interface available to the outside, connections perform the linking between ports. In this work those connections are always directed and acyclic which yields a directed, acyclic graph of connected components. The concept of a perception

graph enables domain experts to model – from a structural point of view – the realization of diverse perception features ranging from simple filtering pipelines to more elaborated perception graphs with multiple input, output and processing branches. It must be stressed that perception graphs do not remain isolated, but can be composed to construct more advanced systems which are rich of functionality (see finding **F3** in Section 3.4). Further, perception graphs can be – like components – attached with configuration values such as contextual information about the appropriateness of a slip detector for a certain context (see finding **F4** in Section 3.4) or other functional and non-functional properties.

### 3.5.3. Deployment Variability

The core concepts involved in modeling the deployment variability dimension are platforms and deployments which are described in the following paragraphs.

**Platforms** encode not only the configuration of peripheral devices such as sensors, actuators and their properties (see finding **F1** in Section 3.4), but also choices related to the computational architecture. Firstly, the computational architecture is composed of virtual and real processors. Examples for the former are virtual machines and examples for the latter are CPUs and GPUs. Those processors are capable of scheduling and executing processes and threads. Secondly, platforms are composed of different memory types, for example working memory like RAM, used to store data and code. Thirdly, in order to interconnect platforms, processors, memory and devices different bus types such as CAN, Ethernet and so forth are available. Platforms and their constituents can be enriched with functional and non-functional properties such as latency, frequency and so forth (see finding **F8** in Section 3.4).

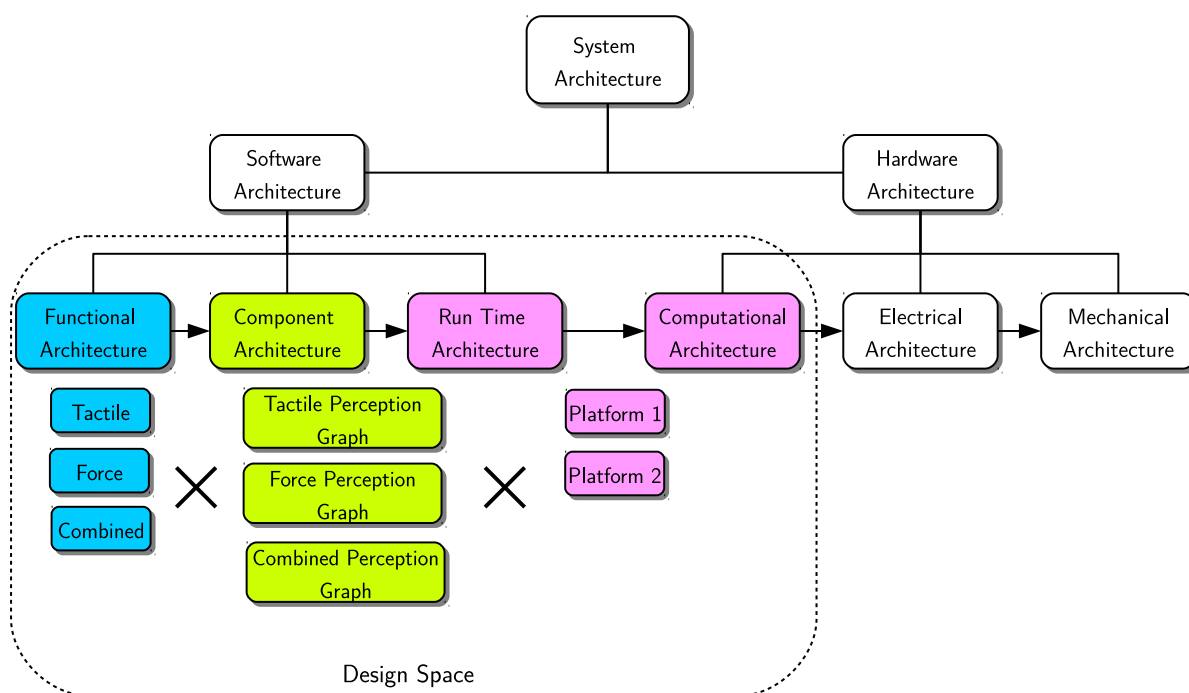
**Deployments** encode how the component architecture – in the context of this thesis expressed as perception graphs – are mapped to the run time architecture which in turn is executed on the aforementioned platforms. In order to facilitate the association of component architectures, run time architectures and platforms one requires additional concepts. Firstly, the run time architecture needs to be defined in terms of executable and schedulable processes and threads. Secondly, different constraint types are required to specify deployment requirements such as the availability of peripheral devices like sensors (see finding **F1** in Section 3.4) or constraints on properties of platform elements (see finding **F8** in Section 3.4). Lastly, the peculiarities of deployment mechanisms and frameworks needs to be described. Those mechanisms which can be, for example, found in robotics software frameworks take the aforementioned information as an input and initiate the deployment according to it, i.e. components are started, configuration parameters are set and so forth.



### 3.5.4. Design Space

Developing robot perception systems requires domain experts to combine functional, architectural and deployment variability. Such a combination is required as one needs not only to select the perception features and their perception graphs required for the task at hand, but also as one needs to decide how and on which platform those graphs shall be deployed on.

Generally speaking, combining variability dimensions forms a design space which defines at a meta level what are all of its possible implementations. Those possibilities are called design alternatives and differ on many different aspects, one being preferred to the other depending on how, where, when or what the application should do. It is important to emphasize that even small domain models expressing functional, architectural and deployment variability include already many design alternatives. Such a situation is exemplified in Figure 3.5 where the design space of the slip detection domain example is visualized as the cartesian product of the elements in the variability dimensions.



**Figure 3.5.** The design space as a combination of functional, architectural and deployment variability. In this example the functional variability is composed by three features, the architectural variability is composed by three perception graphs and the deployment variability is composed by two platforms.

In total the slip detection domain example contains 18 design alternatives. More precisely, from a functional architecture point of view four different feature selections are possible, namely (*Force*), (*Tactile*), (*Combined, Force, Tactile*) and (*Force, Tactile*). Here, each single feature (e.g. *Force*) is realized by one perception graph and each graph can be deployed on two platforms

(see finding **F8** in Section 3.4). Therefore, selecting solely feature (*Force*) yields in 2 design alternatives whereas selecting features (*Combined, Force, Tactile*) yields in 8 design alternatives. Adding, for example, another platform or perception graph contributes significantly to the overall number of design alternatives.

Exploring such a huge design space in order to review design alternatives is a complex and challenging exercise. Hence, in Chapter 4 an approach for design space exploration is proposed.

### 3.5.5. Conceptual Spaces

Assessing the architectural views one can identify the need to represent not only structural elements such as components, perception graphs and so forth, but also more abstract, conceptual information. For example, from an architectural perspective, the concepts *Wrench, Slip\_Up* and so forth are used to express the data produced and consumed by the components of the slip detection example (see finding **F2** in Section 3.4). In addition, concepts are not only used to express functional and non-functional properties of platforms, components and perception graphs (see finding **F8** in Section 3.4), but also to express contextual information about, for example, the suitability of certain slip detectors for a certain task (see finding **F4** in Section 3.4). Very often domain experts are capable to assign concrete values to those concepts, for example, the size and number of memory for platforms and so forth.

It is important to note that the concepts which are involved in specifying robot perception systems are on different levels of abstractions. For example, the concepts produced by the components of the slip detection example (see finding **F2** in Section 3.4) range from raw sensor data and sub-symbolic information to symbolic information. Hence, a knowledge representation framework which enables domain experts to represent input and output of components, functional and non-functional properties of components, perception graphs and so forth as well as contextual information on various levels of abstractions is required.

In this work the Conceptual Space knowledge representation approach by Gärdenfors [122] is utilized. Here, a conceptual space is a metric space composed of dimensions. In that space concepts are defined as convex regions. A more detailed treatment of conceptual spaces is given in Section 3.6.5.

## 3.6. Formalization of Concepts

This section formalizes the concepts identified in the previous section in the form of metamodels and corresponding constraints.

### 3.6.1. Approach

As discussed in Chapter 2 different approaches exist to formalize domain concepts in the form of one or more metamodels and corresponding constraints. In the context of this work the modeling language Alloy is used to formalize the domain concepts identified in Section 3.5.

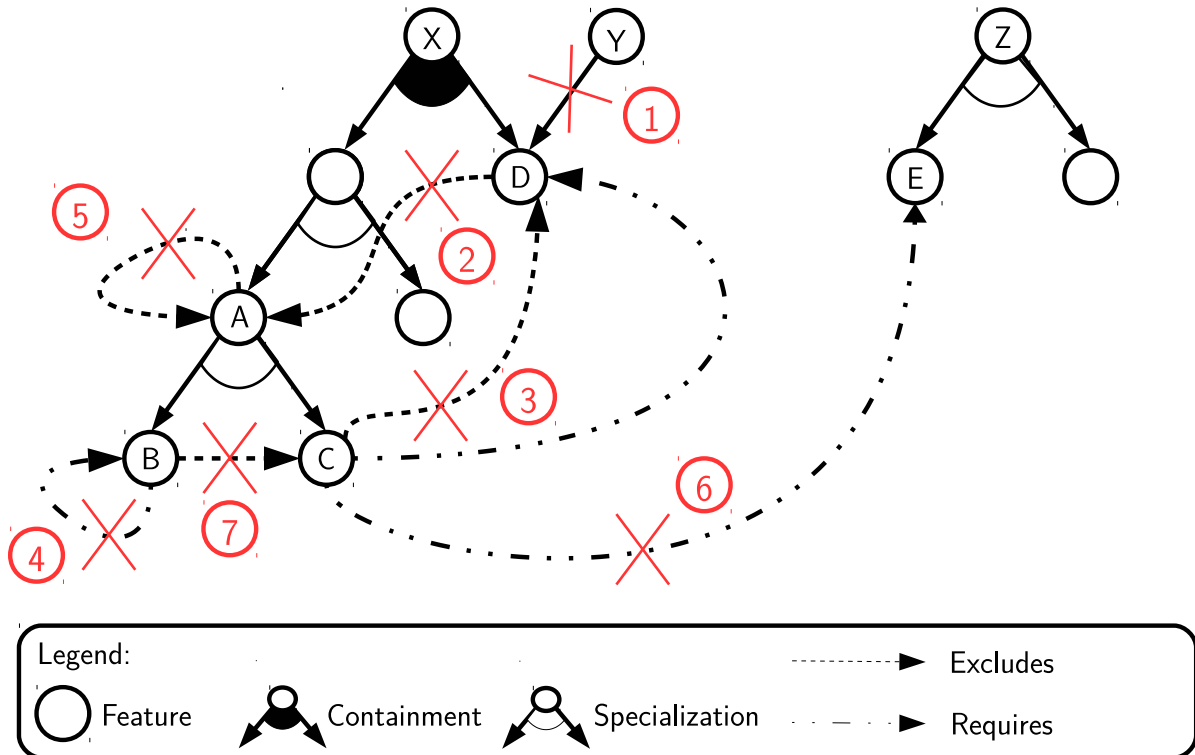
Alloy [31] is a textual, precise, declarative formal modeling language based on relational calculus and transitive closure. Originally, Alloy was developed to “*capture the essence of software abstractions simply and succinctly, with an analysis that is fully automatic, and can expose the subtlest of flaws*” [31], and was successfully applied in a variety of domains, from general software engineering [123] [124] to cryptographic protocol verification [125].

Alloy allows the definition – in an Alloy model – of *a*) concepts called signatures, *b*) relations between concepts called fields, and *c*) constraints on those relations called facts. In essence, a signature introduces a set of elements called atoms. Furthermore, Alloy comes with its dedicated tool, the Alloy Analyzer, which relies on SAT (Boolean Satisfiability) solvers to find in a finite domain, given an Alloy model, the set of all instances conforming to its constraints. It can also find counter-examples, given an Alloy expression to be asserted. The Alloy Analyzer mechanism fosters to develop Alloy models in an incremental, step-wise manner as every modeling step can be analyzed and possibly reverted. Therefore, the technique propagated by Alloy is often called an agile, formal modeling approach.

In the following sections some Alloy models are introduced. Those models formalize the previously identified concepts, namely perception features, components, perception graphs (see Section 3.5.2) and platforms (see Section 3.5.3).

### 3.6.2. Functional Variability

**Perception Features** In order to express the functional variability of robot perception systems the Alloy model shown in Figure A.1 expresses the concept of a feature model. Features are defined with the signature `Feature` that can be mutually exclusive or require one another. In essence, a feature model is a tree of features expressed with the `FeatureTree` signature having a root feature without any incoming edges. The relationship between parent and children in a feature tree are of two kind, namely specialization or containment. Their semantics differ when it comes to feature selection, namely any children feature can be selected in the case of a containment relationship, while exactly one should be selected in the case of a specialization relationship. In the context of Feature Oriented Domain Analysis (FODA) [88], feature selections define a possible configuration of a software that belongs to the application expressed by the feature model. As discussed in Section 3.5.1 those selections can be further refined by integrity constraints. For example, a feature *A* excluding *B* implies that *B* can not be selected if *A* is selected.



**Figure 3.6.** Visualization of the constraints corresponding to the Feature signature. Here, the numbers in the red circle correspond to the constraints enumerated in Section 3.6.2. The figure depicts two distinct feature models with root features named *X* and *Z*. Obviously the relation from feature *Y* to *D* is not permitted as a feature model solely has one root feature. Feature *D* attempts to exclude feature *A* which is not permitted as *C* requires *D* and *A* is a parent of *C*. As *C* already requires *D* it is not allowed at the same time to exclude *D*. Furthermore, features can not exclude or require itself (see features *A* and *B*). As the specialization is a logical OR relation it is meaningless for *B* to exclude *C*. It is important to note that *C* is not permitted to exclude or require feature *E* from another feature model. Having this constraint prevents the composition of feature models. Although feature model composition approaches are becoming popular [126] [127] the core idea of the approach presented in this thesis is to have feature models expressing solely the functional variability and not additional aspects as possibly introduced by composing further feature models.

In addition, some obvious constraints for the Feature signature are defined in the following enumeration and visualized in Figure 3.6.

1. There is only one root feature in the parents of a given feature.
2. Given a feature  $f$ , features required by  $f$  can not exclude  $f$  and  $f$ 's parent.
3. A feature  $f$  can not exclude and require the same feature.
4. A feature  $f$  can not exclude itself or its children/parent.
5. A feature  $f$  can not require itself or its children/parent.

6. Features of one feature tree can not exclude or require features of another feature tree.
7. A feature  $f$  can require or exclude another one only if both have ancestors which are (or are themselves) different alternatives of a same containment.

### 3.6.3. Architectural Variability

**Components.** The Alloy model shown in Figure 3.7 declares two main signatures, namely `SensorComponent` and `ProcessingComponent`. Signatures in Alloy introduce a set of elements called atoms. In Alloy speak both signatures are extensions of the abstract signature `Component` and form a disjoint union (partition) of the `Component` set. Similarly, the `Input` and `Output` signatures are extensions of the abstract signature `Port`.

```

1  abstract sig Port {
2    portType: one Concept,
3    prototype: set Prototype
4  }
5
6  sig Input extends Port {}{
7    this in Component.input
8  }
9
10 sig Output extends Port {}{
11   this in Component.output
12 }
13
14 abstract sig Component {
15   input: disj set Input,
16   output: disj set Output,
17   prototype: set Prototype
18 }{
19   this in PerceptionGraph.components
20 }
21
22 sig SensorComponent extends Component {}{
23   #(input) = 0 and #(output) > 0
24 }
25
26 sig ProcessingComponent extends Component {}{
27   #(input) > 0 and #(output) > 0
28 }

```

**Figure 3.7.** The Alloy model declaring components as the core architectural building blocks. Note, that both `Component` and `Port` are associated with any number of `Prototype` used to express component respectively port properties.

The former expresses the concept of an input port and the latter expresses an output port (see Section 3.5.2). Both `Input` and `Output` have a field `portType` which represents a relation with domain `Input` respectively `Output` and a range given by the signature `Concept`. It is important to note that each element from `Input` and `Output` is associated with exactly one element from `Concept` (see Section 3.6.5). Having defined this multiplicity it is ensured that a port is always

declared with exactly one type. The Component signature defines also two fields, namely input and output declaring distinct input respectively output ports.

In addition, the Alloy model defines some simple, yet important constraints. In order to ensure that ports are always part of a component two signature facts are given both for Input and Output (see line 7 respectively 11 in Figure 3.7). Similarly, elements from both the SensorComponent and ProcessingComponent sets are part of a PerceptionGraph (see line 19 in Figure 3.7). Further, cardinality constraints (see line 23 and 27 in Figure 3.7) are given, namely *a*) sensor components have zero input and at least one output port; and *b*) processing components have at least one input and one output port.

**Perception Graphs.** The Alloy model shown in Figure 3.8 formalizes a PerceptionGraph signature having three fields, namely components, connections and compGraph. The components field is simply any subset of Component. The connections field defines the possible connections among components expressed as an arrow product of Output and Input defined in the component Alloy model (see Figure 3.7).

```

1 abstract sig PerceptionGraph{
2   prototype: set Prototype,
3   components: set Component,
4   connections: Output -> Input,
5   compGraph: Component -> Component
6 }{
7   all port: connections[Output] + connections.Input | port in components.(input+output)
8   all out : connections.Input | out.type = connections[out].type
9   all disj c1,c2 : components| c2 in compGraph[c1] <=> c2 in connections[c1.output].~
   input
10  no c:components| c in c.^compGraph
11  compGraph[Component]+ compGraph.Component in components
12 }

```

**Figure 3.8.** The Alloy model declaring perception graph as a composition of components. Note, that a PerceptionGraphs are associated with any number of Prototype used to express port properties.

Similarly, the possible computation graphs – connected sensing and processing components – is defined as an arrow product of Component. However, having solely the PerceptionGraph signature is not sufficient for expressing well-formed perception graphs as identified in Section 3.5.2. To this end, some constraints (signature facts) for the perception graph model are, in the same order:

- All the input and output ports of connections belong to components which are part of the perception graph.
- All the connections between input and output ports are only possible if the port type is the same.

- The components in `compGraph` reflect the input and output relations of connections.
- All the perception graphs are acyclic.
- Components in `compGraph` are actual components of the perception graph.

#### 3.6.4. Deployment Variability

**Platforms.** The Alloy model shown in Figure 3.9 declares the computational architecture. The main signature is the `Platform` and associated platform constituents, namely `Processor`, `Memory` and `NetworkInterface`.

Here, the `Cpu`, `Gpu` and `VirtualProcessor` signatures are extensions of the abstract signature `Processor` and form a partition of the `Processor` set. Semantically `Cpu`, `Gpu` and `VirtualProcessor` express elements where processes and threads can be executed. According to the AADL standard [128] it is important to note that a `VirtualProcessor` is simply a logical resource. Hence, the Alloy model ensures that a `VirtualProcessor` itself is bound to a physical CPU. Further, it is ensured that a platform always contains at least one `Cpu` and one `Ram` element. Here, the `Ram` signature is an extension of the abstract `Memory` signature expressing random access memory. In order to model other means to store data and code, for example, hard disks and so forth, one would simply add another `Memory` extension. The `Platform` signature defines also a `networkInterface` field declaring distinct `NetworkInterfaces`. Semantically, such a software or hardware interface is used to connect the `Platform` to a computer network, other computers or peripherals. To this end, the `Bus` signature expresses a means to interconnect platforms.

#### 3.6.5. Conceptual Spaces

The Alloy model shown in Figure 3.10 declares conceptual spaces according to the description of Adams and Raubal [129]. However, for the sake of simplicity the Alloy model does not include a `Domain` signature for expressing sub-spaces of a conceptual space as originally proposed by Adams and Raubal. A conceptual space is built up from a set of `QualityDimensions` and a `Concept` is a convex region in that space. Points within that region are called `Prototypes` and represent entities/objects of the real world. In [122] Gärdenfors describes a simple, yet illustrative example of a one-dimensional conceptual space of time where a point *now* divides the space in two concepts, namely *past* and *future*.

It is important to note that prototypes encode typical values for a concept. For example, within a three-dimensional conceptual color space RGB made of *red*, *green* and *blue* dimensions a prototype with the values (255, 255, 0) would represent a typical value for the concept *yellow*. As quality dimensions represent “...means for measuring and ordering different quality values of objects in the space” [129] they need to be equipped with a scale of measurement. In the context of this work three `MeasurementLevels` – as proposed by Adams and Raubal – are employed,

```

1  abstract sig Processor {
2      prototype: set Prototype
3  }{
4      this in Platform.processor
5  }
6
7  sig Cpu, Gpu extends Processor {}
8
9  sig VirtualProcessor extends Processor {
10     boundTo: disj one Cpu
11 }
12
13 sig Device {
14     prototype: set Prototype,
15     connectedTo: some Platform
16 }
17
18 abstract sig Memory {
19     prototype: set Prototype
20 }{
21     this in Platform.memory
22 }
23
24 sig Ram extends Memory {}
25
26 sig Bus {
27     prototype: set Prototype,
28     connectedTo: disj set Bus
29 }{
30     this not in connectedTo
31 }
32
33 sig NetworkInterface {
34     prototype: set Prototype,
35     connectedTo: lone Bus
36 }{
37     this in Platform.networkInterface
38 }
39
40 sig Platform {
41     prototype: set Prototype,
42     processor: disj set Processor,
43     memory: disj set Memory,
44     networkInterface: disj set NetworkInterface
45 }{
46     one p: processor | p in Cpu
47     one m: memory | m in Ram
48 }

```

**Figure 3.9.** The figure shows the Alloy model declaring the elements required to model the computational architecture.

namely `RATIO`, `INTERVAL` and `ORDINAL`. Each dimension permits to apply a set of logical and mathematical operators suitable to model different data. For example, each dimension of the previous mentioned RGB conceptual color space would be expressed as a non-circular `QualityDimension` with an `INTERVAL` having an integer Range from 0 to 255.



```
1  sig Concept {
2    dimension: some QualityDimension
3  }
4
5  sig QualityDimension {
6    measurementLevel: one MeasurementLevel,
7    circular: Bool,
8    range: lone Range
9  }{
10   measurementLevel in INTERVAL implies (#range >= 1)
11 }
12
13 sig MeasurementLevel {}
14 one sig RATIO, INTERVAL, ORDINAL extends MeasurementLevel {}
15
16 sig Range {
17   min: one Int,
18   max: one Int
19 }
20
21 sig Value {}
22
23 sig Prototype {
24   concept: one Concept,
25   values: set Value
26 }
```

Figure 3.10. The Alloy model formalizing conceptual spaces according to Adams and Raubal [129].

### 3.6.6. Resolutions

Up to now the feature, perception graph and platform models represent three different variability dimensions of robot perception systems. It is important to emphasize that those models are orthogonal to each other which means that they can be modified independently without affecting each other. Nevertheless, at the latest when a certain perception feature is selected at run time a mapping from features to perception graphs and platforms need to be performed. In order to enable such a mapping and to maintain the orthogonality of those models two resolution models are introduced in order to serve as a weaving mechanism.

The first resolution model encodes the resolution of a feature by one or more perception graphs. In contrast to Gherardi and Brugali [87] the resolution is not distinct as more than one perception graph might resolve a feature. Such an undetermined resolution is necessary in situations where several perception graphs are realizing the same feature, but with different behavior. Examples are varying timing and precision properties or differences in the expected output of a graph. In order to expose those differences and to enable – at a later stage – the selection process the resolution model allows to expose one or more output ports and possibly properties of perception graphs. In order to ensure the well-formedness of resolutions two constraints are defined. Firstly, exposed ports must belong to components composed in the resolved perception graph. Secondly, exposed prototypes must belong either to components composed in the resolved perception graph or to the perception graph itself.

The second resolution model encodes the mapping of a perception graph to a platform. The mapping is distinct and expresses a default deployment of a perception graph to a particular computational platform. Such a default deployment is desirable in situations where domain experts already know a priori that certain deployment constraints, for example, availability of sensors and so forth are fulfilled by certain platforms. Here it should be mentioned that providing a resolution from perception graph to platform is not mandatory as the deployment approach presented in Chapter 6 is capable to perform the mapping in an automatic manner.

## 3.7. Development of DSLs and Tooling

This section details the implementation of RPSL and DepSL.

### 3.7.1. Approach

Basically two general approaches are common to implement domain-specific languages, namely internal and external DSLs (see Chapter 2). In this research both RPSL and DepSL are realized as internal DSLs. Due to the fact that internal DSLs reuse language infrastructures of their host languages, for example, interpreters, compilers, libraries and so forth, DSL developers can focus on language design and not on DSL implementation peculiarities such as lexical and semantic analysis.

In this work, the RPSL and DepSL domain abstractions (see Sections 3.5 and 3.6) are embedded within the type system of the host language. Hence, both DSLs can be considered as embedded DSLs [130] and not as generative DSLs as their domain abstractions are not transformed, for example, to code. As many internal DSLs are using interpreted languages such as Ruby, Python, Lua and so forth, internal DSLs usually require solely an interpreter for the host language. Both RPSL and DepSL employ Ruby as a host language.

Ruby is a multi-paradigm, dynamic, interpreted programming language designed and developed by Yukihiro Matsumoto [131]. An important feature of Ruby is its type system also known as duck typing. Here, objects are not identified and distinguished according to a strict, static hierarchy of classes, but rather by checking the existence of certain properties and methods. To this end, reflection capabilities are built into Ruby to derive information about a program. For example, Ruby provides methods to infer properties like names and parameters of classes, modules and so forth. Having access to such information makes it possible to apply metaprogramming techniques in a systematic manner. Those techniques allow, for example, during run time the deletion of class methods or even the modification of their visibility. Both reflection and metaprogramming are crucial techniques used in the context of Ruby-based DSL development [132] as they enable developers to modify the appearance and to some extent the behavior of the host language. The suitability of those techniques for the sake

of Ruby-based DSL development is shown in major Ruby-based frameworks and tools, for example, Sinatra [133], a framework for developing web applications or Rake [134], a build utility for Ruby. Those frameworks and tools expose a large part of their functionality through DSLs.

### 3.7.2. Implementation

As envisaged in the DSL process (see Figure 3.2) both metamodels (see Section 3.6) and requirements/findings (see Section 3.4) serve as input to the DSL development activity. Based on those artifacts the identified and formalized domain concepts (see Section 3.5) are grounded in usable – from a DSL user perspective – abstractions.

The domain concepts belonging to the functional and architectural dimension are embedded in RPSL and the domain concepts found in the deployment dimension are embedded in DepSL. According to Günther [132] grounding and embedding activities can be distinguished in three types, namely *a*) language modeling which deals with implementing the domain concepts using the primitives and concepts available in the host language, *b*) language integration which deals with making the DSL usable with other DSLs, frameworks and tools; and *c*) language purification which deals with making the DSL as expressive as possible by, for example, eliminating non-domain relevant tokens and providing syntactic sugar.

From a language modeling point of view the core concepts such as components, perception graphs, platforms and so forth are embedded as Ruby classes which is a standard approach [132] to express domain concepts in Ruby. Those classes can be considered as the backbone of both RPSL and DepSL as they are not directly exposed to the DSL user. In fact, mainly domain operations, for example, adding a port to a component, defining the name of a perception graph and so forth are exposed. Those domain operations are available within reusable Ruby modules. To this end, two modules are defined, namely one for RPSL and another for DepSL.

In Figure 3.11 an excerpt of a domain model for the force-based slip detector (see Section 3.3) is shown. Using RPSL to specify the force-based slip detector simply requires to load the RPSL Ruby module (see line 1 in Figure 3.11). Subsequently, all the elements of the force-based slip detector, for example, sensing and processing components, perception graphs and so forth can be specified.

To support this the concept of Ruby block scoping is intensively used both in RPSL and DepSL as a DSL purification technique. A block scope provides a clear context for evaluating statements and – if required – to stack hierarchical information. By using the `do . . . end` notation (see, for example, line 3 and 6 in Figure 3.11) one can define a block scope. Scoping improves the DSL expressiveness as the clear context provided by the block decreases the required text to write in order to use certain operations. For example, having the block scope one can simply write

```

1  require 'rpsl'
2
3  rpsl.sensor_component do
4    name "force_sensor"
5    add_port :out, "out_port", "wrench"
6  end
7
8  rpsl.processing_component do
9    name "slip_detection"
10   add_port :in, "in_port", "wrench"
11   add_port :in, "out_port", "force_slip"
12 end
13
14 rpsl.perception_graph do
15   name "force_slip_detector"
16   connect :src => "force_sensorA", "out_port",
17           :sink => "slip_detection", "in_port"
18   attach_prototype "is_base_moving"
19 end
20
21 rpsl.concept do
22   name "task_context"
23   use_dimensions "twist_linear_x"
24   use_dimensions "twist_linear_y"
25   # ...
26 end
27
28 rpsl.interval_dimension do
29   name "twist_linear_x"
30   interval :is_circular => false, :start => 0, :end => :INFINITY,
31           :type => :RPSL_FLOAT64
32 end
33
34 rpsl.prototype do
35   name "is_base_moving"
36   concept "task_context"
37   add_prototype_element :NON_ZERO, "twist_linear_x"
38   # ...
39 end

```

**Figure 3.11.** An excerpt of the domain model of the force-based slip detector (see Section 3.3) represented in RPSL. Two atomic components are modeled, namely a `force_sensor` providing wrench data (`out_port`) and a `slip_detection` component demanding wrench data (`in_port`) and providing a slip signal. Both components are connected in the `force_slip_detector` yielding a structurally complete specification of the force-based slip detector. The attached prototype `is_base_moving` belonging to the `task_context` concept encodes the suitability of the `force_slip_detector` for scenarios when the base is moving. The `task_context` concept is composed of dimensions expressing the odometry of the robot, namely linear and angular velocity. To precisely express the situation when the base is moving a prototype `is_base_moving` is defined which assigns non-zero values for the corresponding linear and angular dimensions. Note, the model is erroneous in line 8 where a processing component is defined without having an output port and in line 16 where the non-existing component `force_sensorA` is connected with the `slip_detection` component. Interpreting the domain model with RPSL yields the error message shown in Figure 3.12. Further, the RPSL above is an excerpt and lacks the specification of how the functional and deployment variability is resolved. An example domain model of perception features is shown in Chapter 4 and an DepSL example is shown in Chapter 6.

`add_port` instead of `sensor_component.add_port` (see line 5 in Figure 3.11) for adding a port to a sensor component.

In the context of RPSL and DepSL additional purification techniques are applied, namely parentheses cleaning and keyword arguments [132]. The former eliminates parentheses around method calls and improves readability. The latter uses a literal hash to name parameters in order to avoid ambiguities. For example, the literal hashes `:src` and `:sink` (see Figure 3.11 line 16–17) are used to distinguish the parameters required to declare a connection among components.

In order to ensure, for example, the well-formedness of the RPSL domain model shown in Figure 3.11 one needs to check all the constraints defined in Section 3.6. This checking is done once the domain model is interpreted. To this end, dedicated methods are programmatically checking whether constraints are violated or not. Those methods range from reflection techniques used to check the existence of properties to more elaborated approaches implementing, for example, a topological sort to ensure the DAG property of perception graphs. In case constraints are violated the DSL user gets informed with an elaborated error message (see Figure 3.12).

```
1 $> ERROR: processing component slip_detection [LINE: 8] does not have an output port.  
2 ERROR: perception graph force_slip_detector [LINE: 14] is not valid as component  
force_sensorA does not exists.
```

**Figure 3.12.** The textual error message printed on a console when interpreting the domain model shown in Figure 3.11 with RPSL.

## 3.8. Related Work and Discussion

In this section related work will be discussed with respect to the activities defined in the DSL development process (see Section 3.2). Before each individual DSL process activity is discussed, the following paragraph assesses related work for the development process itself.

**DSL Development Process.** In robotics – as noted in Chapter 2 – very little is known about how DSL developers identify and consolidate abstractions which on the one hand suit the domain best and on the other hand are the building blocks of DSLs. Although some authors report how they ground their DSLs, for example, based on an ontology [99], an architectural pattern [103] or a domain analysis [102], little is known about the involved stakeholders and their activities. Having such reports would help the robotics community to design and develop DSLs and could be the starting point for structuring an overall DSL development process in robotics.

In this work stakeholders were defined before assessing domain examples. This supported not only a structured DSL development, but also improved the awareness of certain aspects. For example, while assessing the slip detection domain example, Sanchez initially was only the domain expert whereas the author of this thesis was the architecture expert and DSL developer. After some iterations it turned out that Sanchez – based on his experience in deploying the slip detector – actually contributed several findings (see Section 3.4) by learning the architectural views through the interaction with the architecture expert.

Interestingly, the tailored DSL development process employed in this work contains related building blocks and terminology as those found in well-known, generic software development methodologies such as Unified Process [135] and more agile methods [47] such as Scrum [136]. Those methodologies contain several building blocks, for example, activities, iterations, stakeholders and so forth which are generic enough to describe who, when and why some activity is performed. In addition, activities such as the creation of use cases is similar to techniques employed in the Unified Process. However, one major difference remains: the major artifacts created in the DSL development process are metamodels and not executable source code. Therefore, activities like unit testing – as envisaged in agile process models – are not that straightforward to implement in the DSL development process as required techniques, for example, model-based testing are not (yet) mature enough.

In the software engineering domain, Kolovos *et al.* [137] have identified three stakeholders in their DSL development process, namely *a)* the system/software engineer, who aligns with the DSL developer in the proposed process (see Section 3.2) and develops the tooling, *b)* the developers, who uses the tools to develop domain models; and *c)* the customer, who evaluates the developed models. While only the system/software engineer has an equivalent in the proposed process, Kolovos *et al.* also outline an end-user programming approach, which combines the developer and the customer. This latter approach is also part of the proposed development process.

**Description of Domain Examples.** Very often grounding DSL design and development on domain examples is the standard, yet most appropriate approach for assessing a domain if other means, for example, domain ontologies are not available. It is worth noting that the selection of domain examples is challenging. On the one hand those examples should be broad enough to identify commonalities of a domain and on the other hand specific enough to identify variabilities of a domain [138]. In order to ensure that domain examples are both broad and specific domain experts have been included for selecting and describing domain examples. Including domain experts in this activity yielded a pleasant side effect, namely domain experts experienced the pressing need to have means to specify their design decisions of robot perception systems.

**Analysis of Architectural Views.** Atkinson and Tunjic described in [139] the importance and need to employ view-based approaches for designing and developing software-intensive systems. Views are in particular useful in model-based development environments where one or more DSLs are utilized to represent knowledge about partial aspects of a system. To conclude, the views used in this thesis showed to be meaningful and sufficient as demonstrated in the structured identification of architectural findings (see Section 3.4).

**Identification of Common Concepts.** In the following paragraphs modeling approaches from the general-purpose and robotics domain are assessed from the architectural views perspective. To this end, the concepts available in those approaches are related to the concepts which are available in RPSL and DepSL. A summary of this assessment is shown in Table 3.8 and discussed in the following paragraphs.

- AADL is a textual architecture description language with visual portrayals of certain modeling elements [128]. Initially, AADL was developed for modeling avionics architectures, but is nowadays used for modeling embedded and cyber-physical systems of various sort [140]. From the architectural views perspective, AADL lacks concepts to model high-level software features as those proposed in this thesis. Central to AADL are different component categories used to model application software not only in terms of their component architecture, but also in terms of processes, threads and execution platforms. As discussed in Section 3.6 the concepts available in DepSL are inspired by AADL.
- SysML is a visual modeling language [141] realized as an UML profile (see Chapter 2). On the one hand the profile is a subset of UML2 and on the other hand introduces several modifications and extensions of standard UML in the form of new diagram types. From the architectural views perspective SysML lacks concepts to model high-level software features as the one proposed in this thesis. Nevertheless, SysML provides the UML package diagram which enables domain experts to organize models into packages and to define dependencies between them. Although the package diagram could be somehow misused to model the functional variability of robot perception systems it is not expressive as the feature modeling approach employed in this thesis as, for example, the specialization concept is missing. In order to model component-based systems, SysML introduces two additional diagram types, namely the block diagram and the internal block diagram. The former enables domain experts to model components and their composition whereas the latter represents interconnections and interfaces between blocks. Those blocks are not only used to model software components, but also to model other system entities, for example, hardware, data and even persons interacting with the system. Hence, first-class citizens for modeling, for example, platforms (see Section 3.5), devices and so forth are not available.

Domain	Approach	Architectural Views			
		Functional	Component	Runtime	Computational
General Purpose	AADL	–	●	●	●
	SysML	○	●	○	○
Robotics	Hyperflex	●	●	–	–
	SmartSoft	–	●	●	○
	V3CMM	–	●	–	○
	RobotML	–	●	○	●
	SafeRobots	○	●	–	–
	LSS	–	●	–	○
	RPSL	●	●	–	–
	DepSL	–	–	●	●

**Table 3.1.** The table summarizes related work from the architectural views perspective. Here, a – means that no concepts are available to express a view, a ○ means concepts are partially available and a ● means concepts are available.

- HyperFlex is a software toolchain supporting developers in designing, reusing and composing robotic software systems [87]. From an architectural views perspective, HyperFlex is mainly concerned with modeling the functional and architectural variability of robotic software systems. To this end, feature models and framework-specific component models are integrated in the toolchain. However, HyperFlex lacks concepts to model the run time and computational architecture.
- SmartSoft is a model-based robotics software development approach. The approach provides an integrated development environment called SmartMDSD for robotics software development [83]. SmartMDSD is based both on Eclipse modeling tools and on UML profiles for specifying their underlying metamodel. The toolchain integrates textual and graphical DSLs in order to enable domain experts and developers to model not only the component architecture of robotic systems, but also to coordinate tasks and to orchestrate components [70]. Nevertheless, concepts for modeling the functional architecture are missing. Further, SmartMDSD is based on the SmartSoft robotics software frameworks and framework-specific concepts such as communication patterns and so forth are exposed to the toolchain user. Therefore, a basic understanding of SmartSoft concepts is crucial in order to use the toolchain.
- V<sup>3</sup>CMM is a modeling language – based on Eclipse/ECore – for component-based design and development of robotic systems [100]. The V<sup>3</sup>CMM metamodel and tooling comprises three complementary views, namely a structural view, a coordination view and an algorithmic view. The structural view mainly deals with the specification and composition of components and relates to the component architecture view defined in this thesis. The coordination view deals with the event-driven behavior of the components by employing UML state machines whereas the algorithmic view is



based on UML activity diagrams to express different component implementations. For example, simple method calls or requests to execute operations provided by other components. However, concepts to model the functional architecture are missing and details about the run time architecture are hidden in code generation templates.

- RobotML is a modeling language based on UML profiles and targeted to ease the design, simulation and deployment of robotic applications [99]. To this end, RobotML is structured around four main packages, namely *a*) the architecture package providing concepts to model software and hardware components, *b*) the behavior package providing state machine concepts to model the behavior of software components, *c*) the communication package providing concepts relevant for data and control flow among components, and *d*) the deployment package providing concepts to define an assignment of a robotic system to a target platform, for example, a middleware or simulator. Although RobotML provides an impressive tooling – for example for the sake of code generation – concepts for modeling the functional architecture are missing. In addition, details about the run time architecture and deployment peculiarities are hidden in code generation facilities.
- SafeRobots is an Eclipse-based toolchain integrating graphical and textual DSLs for developing robotic systems [97]. From an architectural views perspective – as discussed in Chapter 2 – SafeRobots is mainly concerned with modeling the component architecture not only from a structural, but also from a non-functional point of view by attaching non-functional properties to components and sub-systems.
- LSSL is the logical sensor specification language [61] which enables domain experts to model the structure of sensor and sensor data processing components (see Chapter 2). LSSL is focused on modeling the computational architecture – solely in terms of sensing devices – and the component architecture. Concepts for modeling the functional and run time architecture are not available.

**Formalization of Concepts.** As stated in Chapter 2 the majority of developers in robotics apply somewhat established approaches and corresponding tooling, for example ECore/OCL, for formalizing their DSL metamodels and constraints. On the contrary, employing Alloy for this purpose – as done for RPSL and DepSL – is somehow unusual. Using Alloy for DSL development yields an iterative process where signatures (concepts) are specified and instances of those signatures and their relations are generated. Analysing and possibly finding problems in those instances fosters the definition of structural constraints which can be added to the concerned signatures. Subsequently, those constraints can be directly evaluated by generating instances again. However, a major disadvantage of such an approach is that language formalization and implementation are decoupled. More precisely, there is no automatic way to ensure that all the constraints for RPSL and DepSL are actually realized in the corresponding Ruby implementation. To this end, the DSL

developer is in charge of implementing functions to ensure the well-formedness of RPSL and DepSL domain models.

**Development of DSLs and Tooling.** From a DSL development and tooling perspective the most obvious question to raise is why RPSL and DepSL were developed as internal DSLs and not, for example, as modeling languages based on UML/SysML profiles. Clearly, profile approaches are appealing as the development is well-supported in terms of tools and libraries. Those tools and libraries facilitate the development of model-to-text transformations, graphical editors and so forth. However, profile approaches are considered to be heavyweight as they inherit UML/SysML concepts. In fact, as UML/SysML are general-purpose modeling approaches they contain numerous, domain-independent concepts. Furthermore – as Alonso *et al.* [100] already pointed out – domain models built from those profiles carry many tags and stereotypes which makes it challenging to inspect, analyze and debug them.

**Implementation and Analysis of Domain Examples/Models.** All the domain examples described in the first DSL development process activity were modeled with RPSL and DepSL. In addition, the robot perception applications described in Chapters 5 and 7 employed RPSL and DepSL to express design decisions. In those applications the concise and small set of concepts available in RPSL and DepSL turned out to be sufficient, yet expressive enough to let domain experts express their design decisions. Furthermore, one observed that domain experts reused domain models, for example, sensor and platform models, for specifying new robot perception systems. Clearly, such a modeling by reuse activity is enabled by the availability of DSLs as they enable domain experts to create models in a reusable manner. Having RPSL and DepSL at their disposal domain experts easily identified design flaws of their perception systems, for example, redundant high-level perception features, missing dependencies among features, data type errors, underspecified platform models and so forth. It is worth noting that RPSL has been also used not only to specify robot perception systems, but also to analyze them. For example, Ingibergsson *et al.* [142] employed RPSL to model a perception system for the sake of analyzing functional safety requirements of an agricultural robot. To this end, Ingibergsson *et al.* [142] extended RPSL with means to annotate domain models with safety and validity requirements.

Last but not least it is worth to discuss RPSL and DepSL in connection with a criticism often associated with DSLs, namely the language cacophony problem. According to Fowler [38] language cacophony describes the concern that languages in general are difficult to learn. Although DSLs should not be confused with general-purpose programming languages – which are much harder to learn – both RPSL and DepSL propose two – for many domain experts – unknown concepts and abstractions, namely perception features and conceptual spaces. Although a DSL should be “*natural/suitable for the stakeholder who specify a particular concern*” [143], those concepts were selected not because they appear explicitly in the domain of

robot perception systems, but because they can be used to solve some representation problems (see Sections 3.5.1 and 3.5.5). Thus, domain experts need to study those concepts in order to use RPSL and DepSL. It can be said that domain experts like doctoral students who have employed RPSL and DepSL grasped those concepts very easily after studying some examples. Nevertheless a study which investigates the advantages and disadvantages of RPSL and DepSL from different user perspectives remains to be done in future work.

### 3.9. Summary

The quote by Brooks sets the tone of this chapter, namely how to enable domain experts to specify their robot perception systems in the presence of functional, architectural and deployment variability. To this end, this chapter introduced a structured, domain-specific approach for identifying, consolidating and describing the concepts which are involved in specifying robot perception systems. The small, yet concise set of formalized concepts are the building blocks of two internal DSLs, namely RPSL and DepSL both enabling domain experts not only to express, but also to communicate their design decisions to other stakeholders. Having RPSL and DepSL at their disposal domain experts are encouraged to employ a lightweight, agile development approach of robot perception systems where specifications are not dumb text documents, but interpretable, verifiable RPSL and DepSL models.



## Chapter 4.

# Exploring the Design Space of Robot Perception Systems

*“Something hidden. Go and find it. Go and look behind the ranges – something lost behind the ranges. Lost and waiting for you. Go!”*

— “The Explorer” from Rudyard Kipling, 1865–1936

### 4.1. Introduction

Chapter 3 showed how the functional, architectural and deployment variability dimensions of robot perception systems can be specified individually by utilizing RPSL and DepSL. Combining those dimensions leads to a design space of robot perception system which defines at a meta level what are all of its possible implementations. Those possibilities are called design alternatives and differ on many different aspects, one being preferred to the other depending on how, where, when or what the application should do.

Exploring the design space is the process of reviewing those design alternatives – prior to their implementation – with intention to verify that the set of all design alternatives to be implemented covers all the possible scenarios in which the application is to be executed. This exercise is known as Design Space Exploration (DSE) [144].

In this chapter two challenges related to DSE of robot perception systems are addressed, namely, *a)* the formal definitions of design spaces, a non-trivial task due to the many dimensions to be taken into consideration, and *b)* the automatisisation of DSE, that is, enabling a domain expert to review design alternatives corresponding to a given design space effortlessly.

Those challenges are tackled by applying two technologies, namely RPSL (see Chapter 3) for the specification of both functional and architectural variability and Lightning [145], a

language workbench. This workbench is used not only to obtain design alternatives from RPSL specifications, but also to visualize them.

The remainder of this chapter structured as follows. In Section 4.2 typical DSE activities performed by domain experts are identified. In Section 4.3 those activities are composed into a DSE approach. Furthermore, it is described how each activity is realized and supported by software tools in the DSE approach. In Section 4.4 the proposed DSE approach is exemplified with the help of a real-world case study. Related work is discussed in Section 4.5 and some insights are summarized in Section 4.6.

## 4.2. Motivation

Starting point of the DSE approach proposed in this chapter is a robot perception domain expert who is capable of expressing a design space as a combination of variability dimensions. To this end, both functional and architectural variability – expressed in RPSL – of robot perception systems are considered as design spaces (see Chapter 3). The domain models representing those variabilities are *a*) well-formed as they conform to the specified metamodels (see Section 3.6), and *b*) checked whether or not they comply with the constraints (see Section 3.7).

However, solely relying on well-formed domain models is merely adequate to carry out design space exploration for two main reasons. Firstly, there are simply too many design alternatives even for small domain models (see Section 3.5.4) to inspect manually. Secondly, obtaining those design alternatives manually is already an exercise considered to be prone to errors as all the constraints of RPSL need to be checked also for each design alternative.

Therefore, an approach to DSE of robot perception design spaces should enable domain experts to carry out the activities described in the following paragraphs. To support domain experts in executing DSE it is desirable that those activities are performed by software tools in a (semi)-automatic manner:

**Obtaining Design Alternatives.** A DSE is about reviewing valid design alternatives for a given design space, thus a DSE approach should provide mechanisms to automatically obtain from a domain model the set of all possible – and valid with respect to structural constraints – design alternatives.

**Inspecting Design Alternatives.** In order to let domain experts inspect design alternatives they need to be depicted in an adequate manner. As discussed in [146] it is desirable to depict design alternatives in a syntax the domain expert is familiar with. In the context of this thesis, design alternatives obtained while performing DSE are represented intuitively in order to minimize the cognitive effort the domain expert need to provide to inspect them. This has effect to increase both the speed and the quality of the design space exploration.

**Filtering Design Alternatives.** Very often domain experts are not interested in the complete set of design alternatives, but might want to review those design alternatives having certain properties solely. For example, an execution time below a given threshold, usage of certain processing components and sensors and so forth. Hence, a DSE approach should thus let domain experts filter out valid design alternatives that do not meet their expectations regarding the previous properties.

### 4.3. Approach

An overview of the DSE approach proposed in this thesis is depicted in Figure 4.1. Here, the DSE activities described in Section 4.2 are arranged in a process model. The first activity performed by a domain expert is the specification of domain models using RPSL. Those domain models are stored in a repository. Based on those models design alternatives are obtained. This activity is automated by the language workbench Lightning described in Section 4.3.2. In order to enable domain experts to inspect those design alternatives they are automatically translated in a graphical representation.

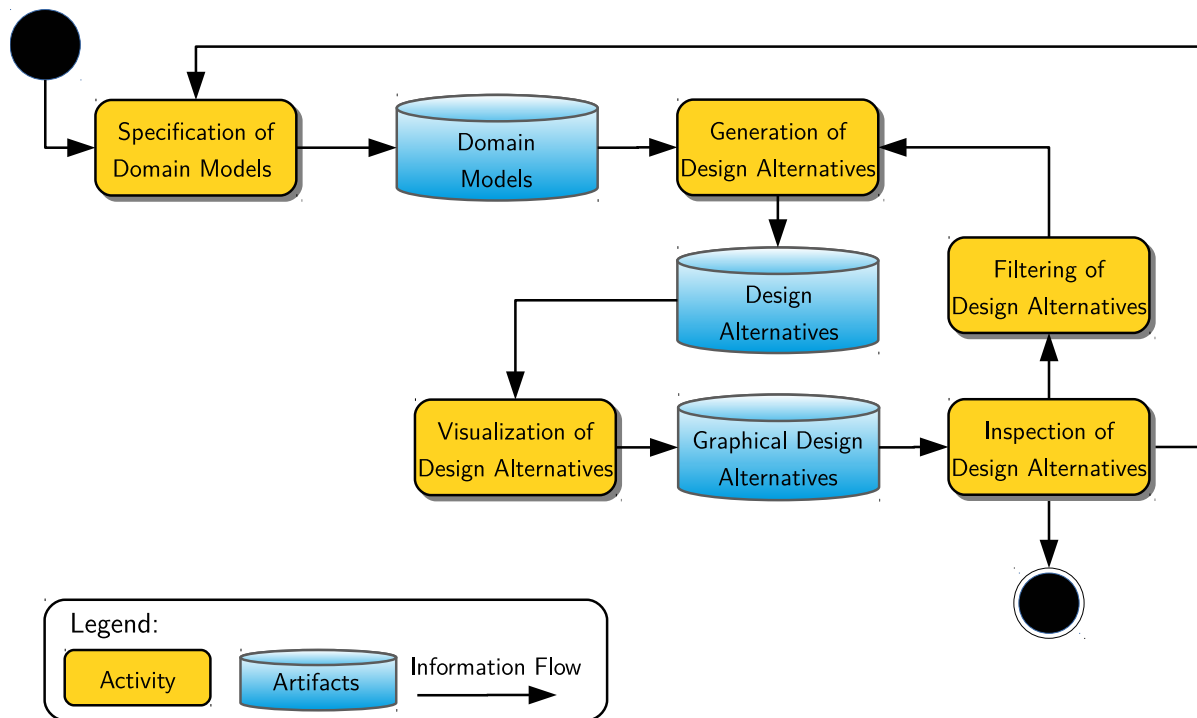
Depending on the goals of the DSE the domain expert decides whether or not to terminate the exploration. In essence, two options are available, namely *a)* the domain expert recognizes that there are too many design alternatives so she/he needs to filter and guide the exploration, or *b)* the RPSL domain models need to be refined in case unfeasible design alternatives have been identified. Those refinement activities, for example, adding perception graphs, removing features and so forth, are well-supported by RPSL.

#### 4.3.1. Specification of Domain Models

In order to specify domain models for the sake of design space exploration one simply needs to employ RPSL to express functional and architectural design decisions.

#### 4.3.2. Generation of Design Alternatives

Before design alternatives can be obtained they need to be rigorously defined. Given a domain model – expressed in RPSL – a conforming design alternative consists of a selection of features and of a selection of exactly one perception graph per selected feature. Those selections are called configurations. The design alternative also contains what is called a super graph, the super graph being a well-formed composition of all the selected perception graphs. Having the concept of a super graph enables domain experts to identify perception graph compositions which were not foreseen and arise simply by selecting certain features. Like the formalization



**Figure 4.1.** The proposed design space exploration process composing activities and repositories of artifacts.

of RPSL metamodels (see Section 3.6), configurations and super graphs are defined in Alloy (see Figure 4.2). This allows us to employ Alloy to obtain all possible design alternatives.

Obtaining design alternatives is achieved by leveraging the *Alloy Analyzer*. The analyzer relies on boolean satisfiability (SAT) solvers in order to find in a finite domain the set of all instances conforming to its constraints.

The structural constraints expressed in Alloy for the design alternative model (see Figure 4.2) are, in the same order:

- With respect to configurations:
  - For each selected feature there should be exactly one perception graph mapped to this feature in the resolution model and selected in the configuration.
  - The number of selected graphs should correspond to the number of selected features.
  - The set of selected feature should not be composed of features which are excluding each other or their parents.
  - The set of selected feature should contain at least one leaf feature implementing each required feature.
  - All selected features should be leaf features.



```

1 module AlternativeModel
2   open ResolutionModel
3
4   one sig Configuration{
5     selectedFeatures: set Feature,
6     selectedGraph: set PerceptionGraph,
7   }{
8     all f:selectedFeatures | one p:PerceptionGraph | f->p in feature2Graph.mapping and p
9       in selectedGraph
10    #selectedGraph = #selectedFeatures
11    no disj x,y:selectedFeatures.~*(spec+contain) | x.excluded=y
12    selectedFeatures.required in selectedFeatures.~*(spec+contain)
13    selectedFeatures.(contain+spec)=none
14  }
15  one sig SuperGraph extends PerceptionGraph{
16  }{
17    components= compGraph[Component] + compGraph.Component
18    components=Configuration.selectedGraph.@components
19    this.contains[Configuration.selectedGraph]
20  }

```

**Figure 4.2.** An Alloy model defining a design alternative. In order to access the concepts of features and perception graphs (see Section 3.6) the Alloy model opens the resolution model.

- With respect to the super graph resulting from the given configuration:
  - The components of the super graph should have at least one input/output connection.
  - The super graph contains all selected graphs.
  - The components present in the super graph are those composing the selected graphs.

In order to exemplify the concepts introduced above the slip detection domain example from Section 3.3 is employed. Assuming the functional variability of the slip detection example is expressed by three features, namely *Force*, *Slip* and *Combined* and all of them have a containment relation with a root feature. In addition, those features are resolved by exactly one perception graph. A configuration which selects these three features would then yield in a super graph respectively design alternative expressed in Figure 3.3 as the constraints of the super graph are respected. For example, at least one input port of the *Combined Slip* component and so forth are connected.

### 4.3.3. Visualization and Inspection of Design Alternatives

The design alternatives obtained by the Alloy analyzer (see Section 4.3.2) need to be inspected by the domain expert. To this end, design alternatives should be visualized in an – for domain experts – intuitive manner. By using Alloy a graphical representation of instances (design alternatives) is already available out of the box. Here, the Alloy analyzer generates a graphical visualization of instances following the structure of their origin, that is elements typed by given signatures are represented as boxes and relations between those elements are depicted

```

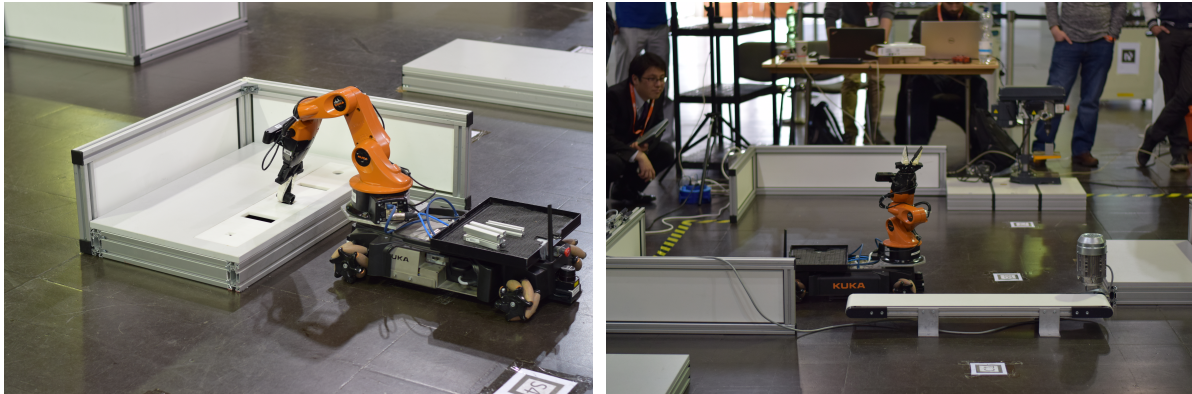
1  module VisualisationTransformation
2  open AlternativeModel
3  open VisualLanguageModel
4
5  one sig CREATE{
6    mainFrame: Component -> INVISIBLE_CONTAINER,
7    inputFrame: ProcessingComponent -> INVISIBLE_CONTAINER,
8    component: Component -> RECTANGLE,
9    inputPort: Input -> RECTANGLE,
10   outputFrame: Component -> INVISIBLE_CONTAINER,
11   outputPort: Output -> RECTANGLE,
12   arc: Output -> Input -> CONNECTOR,
13 }
14 pred guard_component(c:Component) {
15   c in SuperGraph.components
16 }
17 pred value_component(c:Component, r:RECTANGLE) {
18   r.color=(c.weight=1 implies WHITE else (c.weight=2 implies YELLOW else ORANGE))
19 }
20 pred guard_arc(o:Output, i:Input) {
21   o->i in SuperGraph.connections
22 }
23 pred value_arc(o:Output, i:Input, c:CONNECTOR) {
24   c.source=CREATE.outputPort[o]
25   c.target=CREATE.inputPort[i]
26   c.color=RED
27 }

```

**Figure 4.3.** This F-Alloy excerpt contains all the mappings used to define how the super graph of a design alternative is to be rendered and a selection of two pre and post conditions (called guard and value predicates) applying to two of those mappings, namely, component and arc. The component mapping defines, according to its associated guard predicate, that each component composing the SuperGraph resulting from the current configuration is to be rendered as a rectangle. The value predicate then assigns a color to that rectangle given the weight associated to the component it represents (see Section 4.3.4). The arc mapping and its associated guard and value predicate together define, that for each pair of output and input port connected in the resulting SuperGraph, a red connection has to be created between their visual representations (defined by the inputPort and outputPort mapping, respectively).

by an arrow. However, for the sake of design space exploration such a standard depiction of design alternatives is not feasible as the visualization would not convey any domain-specific information. For example, both features and perception graphs would be represented in the same manner – as they are expressed as Alloy signatures – although both concepts differ significantly.

In order to enhance the readability of design alternatives this work employs Lightning, an Alloy-based language workbench. In essence, Lightning is an Eclipse-based plug-in allowing the definition and instantiation – using Alloy – of domain-specific languages. Here, the domain specific visualization of design alternatives is given as a model transformation from the alternative model (see Figure 4.2) to a visual language model. This transformation is expressed in F-Alloy [147], a sub-language of Alloy and integrated in Lightning, allowing the specification



**Figure 4.4.** A youBot mobile manipulation robot deployed in a factory-like environment composed of different service areas. Here, the picture shows an area to insert objects and an area composed of a conveyor belt.

of efficiently computable model transformations. An excerpt of such transformation is given in Figure 4.3.

Having visualized design alternatives, domain experts are required to inspect them. Depending on the overall exploration objectives domain experts will either conclude their exploration if they are satisfied or they will continue to explore the design space by adding filters (see Section 4.3.4).

#### 4.3.4. Filtering of Design Alternatives

From an architectural perspective design alternatives are basically compositions of perception graphs which in turn are built up from sensing and processing components. Very often domain experts are capable to express functional and non-functional properties of those components. For example, different implementations of the SURF feature descriptor [148] available in different processing components, a domain expert is often competent to assign weights to these components. Depending on concrete application requirements a weight might have different meaning such as precision, processing time, memory overhead and so forth. Expressing those weights is important when it comes to DSE and corresponding filtering where certain perception graph compositions are possibly unexpected for domain experts. For example, two graphs are composed in a design alternative and both graphs have a computationally expensive component yielding in an unfeasible design alternative due to timing issues. Having a weighting mechanism a domain expert can filter out those design alternatives not meeting her/his expectations. In order to support such filtering RPSL is extended with the notion of weights assigned to each component which enables domain experts to discriminate perception graphs by their characteristics.

```

1 rpsl.feature_model do
2   name "PickandPlace"
3   add_feature "Application", :is_root
4   add_feature "ServiceArea", :is_mandatory, :child_of => "Application"
5   add_feature "ObDetection", :child_of => "Application"
6   add_feature "ObRecognition", :requires => "ObDetection", :child_of => "Application"
7   add_feature "ContRecognition", :child_of => "Application"
8   add_feature "CavRecognition", :child_of => "Application"
9 end

```

**Figure 4.5.** The functional variability expressed in RPSL. Note, the `ServiceArea` detection feature is mandatory and the `ObRecognition` feature requires the `ObDetection` feature.

## 4.4. Case Study

In this section the proposed DSE approach is exemplified with the help of a case study. The study has been developed in the context of two recent robot competitions, namely RoCKIn [149] and RoboCup@Work [6]. In those competitions mobile robots such as a youBot (see Figure 4.4) are deployed in factory-like environments. Here, the environment is composed of service areas and each service area represents a region of the factory having a specific purpose for a particular task. For example, areas to load objects, to insert objects into object-specific cavities and to place objects into containers. Depending on a goal specification given by some factory worker the task of the robot is to pick objects such as screws, nuts and profiles from containers and to place and eventually insert them at corresponding service areas.

### 4.4.1. Specification of Domain Models

From a robot perception perspective a huge functional variability is required to achieve the task at hand, namely features to detect, recognize and possibly track service areas and objects are required. The variability – expressed in RPSL – is shown in Figure 4.5 and depicted graphically in the upper-part of Figure 4.7.

The feature model shown (see Figure 4.5) includes five leaf features representing high-level perception functions. More precisely, the features described in the following paragraphs are crucial for a pick and place task of industrial objects, for example, screws, nuts and so forth, in the factory-like environment:

- The `ServiceArea` detection feature allows to delimit the service area by detecting the dominant plane in the surrounding of the robot. This information is required by all other features as objects, container and cavities are all lying on this dominant plane.
- The `ObDetection` feature provides a bounding box for each object present in the service area.

```
1 rpsl.sensor_component do
2   name "Kinect"
3   add_port :out, "outCloud", "xyzRGB"
4 end
5
6 rpsl.processing_component do
7   name "PlaneDetect"
8   weight 2
9   add_port :in, "inCloud", "xyzRGB"
10  add_port :out, "outPlane", "Plane"
11 end
12
13 rpsl.perception_graph do
14   name "ServiceArea1"
15   connect :src => "Kinect", "outCloud",
16           :sink => "PlaneDetect", "inCloud"
17 end
```

**Figure 4.6.** This perception graph specification proposes to perform the service area detection by using a Kinect RGB-D camera and a plane detection algorithm such as RANSAC [150]. The Kinect is declared as a sensor component whose output port is called outcloud (typed xyzRGB) and the plane detection algorithm is declared as a processing component whose input port is called inCloud (typed xyzRGB) and output port is called outPlane (typed Plane).

- The ObRecognition feature provides, when possible, a pose and a label for each detected object. The ObDetection feature is thus required by this feature.
- The ContRecognition feature provides, when possible, a pose and a bounding box for each container present in the service area.
- The CavRecognition feature provides, when possible, a pose for each cavity present in the service area.

Each perception feature is resolved by one or more perception graphs. For example, the RPSL specification of a perception graph associated to the ServiceArea detection feature is given in Figure 4.6. It is important to note that this perception graph composes the super graph depicted in Figure 4.7 as it is a possible implementation of the selected feature ServiceArea.

#### 4.4.2. Generation of Design Alternatives

The first step in order to obtain design alternatives is the translation of RPSL models – like the ones given in Figure 4.6 – into Alloy models. Those models are constrained so that the only instance obtainable by Alloy analysis corresponds to the given RPSL specification. This translation is done automatically as it is very straightforward, namely each element of the specification is declared as a singleton extending the appropriate type and fields of those elements having their value bound by constraints. Subsequently, design alternatives are obtained by performing an Alloy analysis on the alternative model. The Alloy analysis produces conforming instances by translating constraints of the analyzed model into a boolean formula

which is then solved by off-the-shelf SAT-solvers, for example, miniSAT [151], SAT4J [152] and so forth.

#### 4.4.3. Visualization and Inspection of Design Alternatives

As discussed in Section 4.3.2 instances obtained by the Alloy analysis are given as input to the F-Alloy interpreter embedded in the Lightning tool along with the model transformation given in Figure 4.3. This interpreter builds from the transformation specification and its input the corresponding visual language instance that can then be rendered to the user. Such a visual feedback, obtained from the analysis of the alternative model (see Figure 4.2), is depicted in Figure 4.7.

The figure shows a domain specific visualization of an alternative model instance. The tree in the upper part of the visualization represents the feature tree of the case study, in which selected features are highlighted in green. For the sake of readability the alternative model to visual language transformation was modified to mask requirement (feature dependencies) arrows. This change can be undone at anytime by the user. The lower part of the visualization depicts the super graph resulting from the composition of perception graphs mapped in the resolution model to the highlighted selected features. It is important to note that this super graph was not specified in RPSL and is resulting from the Alloy analysis of those well constrained models.

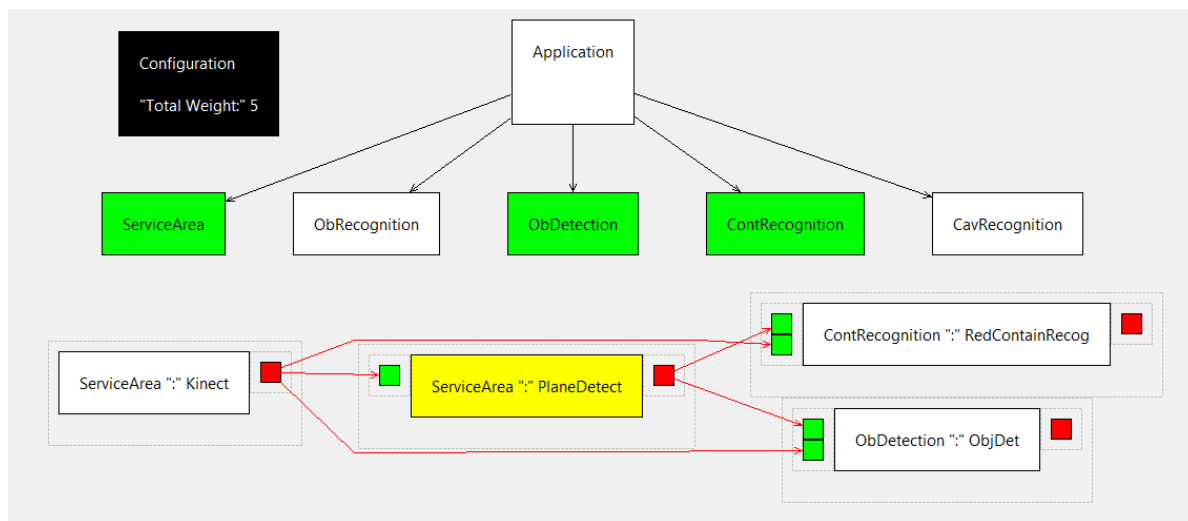
The red and green squares surrounding each component are their output and input ports, respectively. The PlaneDetect component appears in yellow as it is assigned a weight of 2 in the RPSL specification. The black box in the top left corner lists additional properties of the selected configuration. In this example the total weight of the super graph implementing the selected features is shown.

#### 4.4.4. Filtering of Design Alternatives

Domain experts can further guide the exploration by defining additional constraints in the alternative model or by changing the weights assigned to each component. Adding constraints has as effect to reduce the number of instances conforming to the alternative model, thus narrowing the set of design alternatives to be considered. This mechanism becomes useful when the domain expert is interested in design alternatives showcasing specific properties.

In the following paragraphs two examples of constraints used to guide the design space exploration of the case study are discussed.

**Specific Feature/Component Selection.** Assuming a domain expert is interested in reviewing all design alternatives implementing the `ObjectRecognition` feature and whose su-



**Figure 4.7.** A screenshot of a visualization of a design alternative as depicted in the Lightning language workbench.

pergraph contains a sensor providing xyzRGB data in order to ensure that `ObjRecognition` can be carried for this kind of input data. The Alloy constraint used to express this is shown in Figure 4.8.

**Optimal Solution Selection (with respect to the attributed weights).** Assuming a domain expert is interested in reviewing design alternatives with exactly three features implemented and having a minimal weight. The Alloy constraint used to express this is shown in Figure 4.8 with  $n$  incrementally increasing until a design alternative is found.

## 4.5. Related Work and Discussion

The proposed DSE approach builds up on methods and tools from several domains, namely Electronic Design Automation (EDA), Robotics and Software Engineering.

In the electronic design automation domain the DSE problem is often framed as a constrained optimization problem where one or more objectives such as timing, energy and costs of material need to be optimized for a particular task. For example, the work of Hourani *et al.* [153] synthesizes designs of digital signal processors performing well in terms of area, throughput and power dissipation. The overall design space of their application domain is reduced by user-defined performance constraints. Here and in other work, for example Oh *et al.* [154], design space exploration is performed with the aim to optimize one or more objectives.

The approach presented in this chapter is focused on first obtaining all possible design alternatives differing mainly on the structural appearance (e.g. whether certain perception



```
1 ObjRecognition in Configuration.selectedFeatures and some
2 (SensorComponent & xyzRGB.~type.~output ) & SuperGraph.components
3
4 #Configuration.selectedFeatures=3 and SuperGraph.getWeight [] < n
```

**Figure 4.8.** Alloy constraints used to filter design alternatives.

graphs are connected or not) and without considering further properties. As demonstrated in Section 4.4 this appearance carries enough domain knowledge such that a domain expert can seamlessly perform an exploration even without prior knowledge of intermediary languages and concepts.

As already shown in Chapter 2 MDE approaches are getting popular in robotics to structure and manage specific aspects of a design space. Those approaches propose domain-specific languages allowing the intuitive representation of functional [87], architectural [83] [99] [84] and platform [83] [99] variabilities. Although the development of robotic systems greatly benefits from those approaches (e.g. through model validation, model reuse and code generation), providing model-based solutions to design space exploration remains challenging. In particular, modeling design spaces as a whole –i.e., taking into account all variabilities – and using those models to perform a systematic and possibly (semi)-automatic exploration has not yet been achieved. Thus, the proposed DSE approach can be considered as the first one in the domain of Robotics Software Engineering enabling domain experts to explore a design space in a systematic and (semi)-automatic manner.

The design space to be explored in presented approach to DSE is based on the combination of functional and architectural variability. It is important to emphasize that the representation of those dimensions is inspired by metamodels described [87]. Similar to their work, resolution models are employed to compose functional with architectural variability, but in a domain-specific manner –i.e., an architecture is represented as a perception graph. In addition, the following small, yet important change is made: a perception feature resolution yields one or more perception graphs in order to express different architectures, with different characteristics for the same capability. This enables domain experts to compose different perception graphs in the exploration phase.

In the domain of Software Engineering Saxena *et al.* [146] showed that DSE can benefit from approaches based on Model-Driven Engineering advances. More precisely, authors have shown that a framework allowing the definition of a specification language and the exploration of design spaces defined in that language can be implemented. On the contrary, the work presented in this thesis provides an alternative solution to the problem of providing a DSE framework by reusing already existing tools and technologies rather than implementing a framework from scratch. The advantages of such an approach is that – the RPSL to Alloy



transformation being provided – an RPSL expert can directly perform design space exploration without learning any new intermediate language. However, guiding the design space exploration through the addition of Alloy constraints requires some basic Alloy knowledge which can be seen as a limitation to the proposed approach. Nevertheless, a general trend is to define graphical representations for constraint languages (see e.g. [155] and [156]) to make them more user-friendly, suggesting the possibility that such a language could also be defined for Alloy. The presented approach also differs from [146] with the domain specific visualization of design alternatives provided by the framework. The visualization definition can be used out of the box by neophytes but can also easily be modified by Alloy experts. It is still to be determined whether or not the visualization provided by the proposed DSE approach is suitable for other case studies.

A limitation of the current approach is the restriction of the design space to two types of variabilities, namely functional and architectural variability. For more advanced application scenarios a domain expert would also be interested in taking into consideration other design space dimensions as the deployment variability (see Section 3.5.3). In a future work the definition of multi-dimensional design spaces needs to be investigated. This could be achieved by using jointly several DSLs or by providing a general language to express them. It would then be interesting to see whether or not the approach proposed in this chapter can still be applied to explore such multi-dimensional design spaces.

As the generation of design alternatives is based on analyzing Alloy models it is worth noting that the analysis itself is a generally undecidable problem. Therefore, the Alloy Analyzer employs a finite scope approach using SAT-solvers where the number of objects corresponding to each signature is fixed. As demonstrated in the case study, the small scope approach is also feasible for reviewing design alternatives in an incremental manner. Here, the exploration scope is possibly enlarged by the domain expert depending on his/her exploration objectives (e.g. assessing all design alternatives, specific alternatives and so forth).

## 4.6. Summary

This chapter introduced a structured, systematic approach to DSE for robot perception systems. The approach defines a design space as a combination of functional and architectural variability expressed in RPSL. As both RPSL and the DSE approach are based on Alloy it is possible to exploit Alloy analysis mechanism to obtain design alternatives in an automatic manner. By utilizing latest MDE technologies, namely F-Alloy and the Lightning language workbench the obtained design alternatives are visualized in a domain-specific way. The DSE approach contributes significantly to an overall and structured development of robot perception systems where specifications are written (see Chapter 3), explored and possibly modified.



## Chapter 5.

# Implementing Semantic Queries about Domain Models

*“The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.”*

— Edsger W. Dijkstra, 1939–2002

### 5.1. Introduction

Chapters 2, 3 and 4 showed how MDE-based approaches can remedy the problem of implicit knowledge representations in robotics. However, these approaches are usually regarded as a tool for human robot designers – the robots themselves are denied access to this knowledge.

In MDE such knowledge fulfills varying requirements such as documentation of approaches, visualization of concepts or the generation of code. The latter point is most frequently associated with MDE and is a major focus in today’s robotics software engineering (see Section 2.4). It involves modeling a complete system – or a part of that system – at design time. Then, the software is generated and the modeling effort is forgotten.

This chapter argues that the next step is to provide robots with these explicit knowledge representations and let them reason about their software at run time. It is one of the key ingredients for autonomous and intelligent robots which are able to adapt to their tasks and dynamic environments [114].

Thus, the core problem being investigated in this chapter is: *How to grant robots access to the software-related models at run time?* This involves *a)* persistently storing the different notations and formats of DSLs, *b)* composing the various domain models and *c)* querying over multiple domains at design time and run time.

To a certain extent, the AI community has already tackled this requirement, as evidenced by knowledge-enabled approaches like KnowRob [157], RoboBrain [158] or the OpenRobot Ontology (ORO) [159]. At the core of these approaches, graph-based knowledge representations such as ontologies provide common representations and query interfaces to the robot's run-time environment.

This chapter introduces the following contributions for software-related knowledge. Firstly, the concept of labeled property graphs as simple, yet powerful means to persistently store and compose domain models originating from different functional domains and software development phases is being proposed. Secondly, it demonstrates how the labeled property graph enables semantic queries in order to derive implicitly defined information based on stored and composed domain models. Those queries can be instantiated both by *a*) domain experts, for example to analyze systems at design time and *b*) robots, for example, to adapt systems at run time.

The remainder of this chapter is structured as follows. In Section 5.2 a case study is introduced exemplifying the challenges which are inherent in utilizing model-based approaches in robotics. In Section 5.3 the labeled property graph is introduced as a means to store and compose domain models of various sorts. Section 5.4 explains the feasibility of the aforementioned graph structure as a suitable representation in order to raise queries and to derive explicitly and implicitly defined information. Based on the previously mentioned graph and query concepts the case study is implemented by transforming RPSL and DepSL in a graph structure. Related work is discussed in Section 5.6 and Section 5.7 summarizes the contributions presented in this chapter.

## 5.2. Motivation

This section exemplifies the structured, model-based development of a real-world robotic application based on the RAP (see Section 2.4.1). Although the RAP foresees eight different phases, only four phases will be used in the following section to motivate the scenario.

### 5.2.1. Application Scenario

The application constitutes a quadcopter instructed to fly in a GPS-denied indoor environment with time-varying lighting conditions (see Figure 5.1). The environment is equipped with fiducial markers [160] used by the quadcopter to localize itself.

As discussed in [161], the recognition performance in the presence of time-varying lighting conditions significantly depends on adapting the modifiable parameters of the marker recognition algorithm at run time. Therefore, the quadcopter needs to continuously monitor the



**Figure 5.1.** The GPS-denied indoor environment under varying lighting conditions seen from the perspective of the quadcopter.

lighting condition and eventually adapt its software architecture to continue properly estimate its own pose. However, as the computational hardware of the quadcopter is limited it is not possible to execute all functionalities (e.g. marker detection, flight control etc.) for the task at hand on the same platform. Therefore, a remote computer with time-varying memory resources is available where functionalities can be swapped out.

### 5.2.2. Model-based Development

In order to apply the RAP or any other process model in combination with a model-based development approach, textual and graphical DSLs are applied in certain development phases to create domain models. As discussed in Chapters 2 and 3 those models make domain knowledge explicit, which on the one hand, is relevant for a functional or architectural concern of the application under study and, on the other hand, important to represent knowledge during a particular development phase.

Domain models are either created by humans supported through development tools, for example, integrated development environments or by run time environments in an (semi-)automated manner. In both cases, domain models come in various forms such as source code, configuration files, drawings or technical documentation to name a few, all of which are usually represented in heterogeneous formats (see Chapter 2). For example, the domain models created by RPSL and DepSL are Ruby code (see Chapter 3).

Therefore, it remains challenging to compose those domain models technically and conceptually in order to infer answers about the system as a whole. The situation is exemplified in Figure 5.2. Here, in the *platform building* phase, the quadcopter's computational hardware is modeled with DepSL (see Chapter 3), which leads to a textual model which then makes connections and properties such as the number and size of physical memory explicit. Moreover, in the *capability building* phase some perceptual components are modeled with RPSL. A typical question a developer could ask at design time would be: *Which components are executable on*

*the robot?* Answering this question requires the storage of heterogenous domain models in a somehow unified manner as well as their composition, at design time, in a meaningful way so that questions can be answered.

It is important to note, that some domain models can only be partially instantiated or not instantiated at all at design time, as binding information is not (yet) available. For example, the concrete memory usage of an application is not known before deployment time and depends on the execution context. Therefore, several authors in robotics [70] [162] and software engineering [163] argue that domain models need to be created, modified and eventually executed at run time.

Further, domain models do not necessarily remain isolated. In fact, as shown in Figure 5.2 domain models do have implicit links refining some information. For example, the link from the capability building phase to the functional design phase refines the information of how a certain feature is resolved in terms of software components. However, all too often those links are not made explicit, which prevents the systematic composition of domain models at design time and run time.

In summary, applying a model-based development approach throughout a complete development process is rarely done and it remains challenging *a)* to persistently store and compose heterogenous domain models in a unified, systematic manner, *b)* to query composed domain models originating from different functional domains and development phases, and *c)* to systematically modify and employ domain models both at design time and run time.

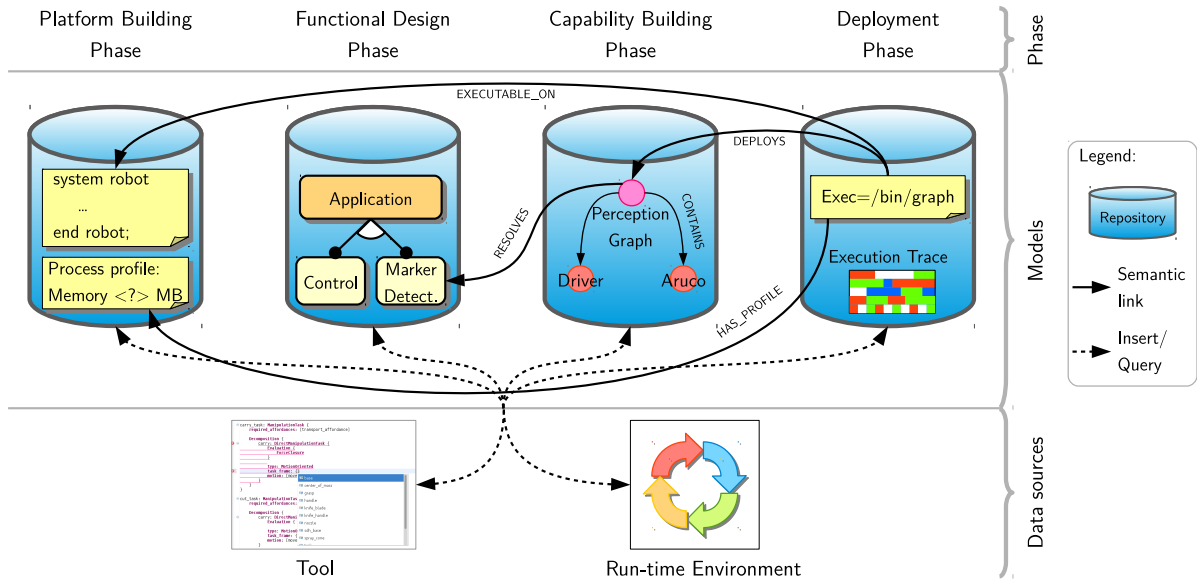
### 5.3. Graph-based Storage and Composition of Domain Models

In order to store and compose heterogenous domain models some sort of common representation or *lingua franca* that describes those models needs to be defined. From a DSL developer's perspective this appears to be a somehow paradox situation as DSL developers usually aim to define very specific abstractions and representations. Nevertheless, such a *lingua franca* is crucial in order to persistently store and eventually compose domain models. Therefore, in this section labeled property graphs are proposed as such a *lingua franca*.

#### 5.3.1. Labeled Property Graph

Generally speaking, graphs are not only well-studied and naturally preserve structure, but can also be easily implemented. A labeled property graph  $\mathcal{G}$  is formally defined as a quadruple

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{P}, \mathcal{L}) \quad (5.1)$$



**Figure 5.2.** A schematic representation of a labeled property graph containing heterogeneous domain models, originating from different functional domains and development phases. Some domain models are semantically connected through links (e.g. RESOLVES which denotes that the perception graphs implements a feature). Human developers or run-time environments either insert new elements into the graph or update existing ones.

where  $\mathcal{V}$  are the nodes and  $\mathcal{E}$  are the edges  $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$  of the graph. Additionally, the graph contains properties represented as key-value pairs ( $\mathcal{P}$ ) and labels ( $\mathcal{L}$ ). Arbitrarily many properties  $p \in \mathcal{P}$  can be attached to either nodes or edges. Similarly, arbitrary many labels can be attached to nodes ( $l_v \in \mathcal{L}$ ) and edges ( $l_e \in \mathcal{L}$ ).

**Example.** In order to demonstrate the formal description some examples based on Figure 5.2 are provided. It is important to note that the aforementioned generic graph structure is not sufficient to enable semantic queries (see Section 5.4) or to enrich the graph with meaning. Therefore, further constraints on the labeled property graph have to be imposed. There must be one or more  $l_e \in \mathcal{L}$  attached to any edge in order to give meaning to relations among nodes. Here, the meaning is expressed by domain-specific labels which are either pre-specified ( $\mathcal{X}$ ) or coming from the domain expert ( $\mathcal{D}$ ), please note that  $(\mathcal{X} \cup \mathcal{D}) = \mathcal{L}$ .

For example, in the capability building phase software components are represented as nodes in the graph. As they represent components, they are further labeled as Component. Similarly, atomic features, represented in the functional design phase, are labeled as Feature. In addition, to link the Aruco component node with the marker detection feature node two edges are introduced. The first one from Perception Graph to Aruco is labeled CONTAINS and encodes that a particular component is part of a larger architecture. In this example, the architecture is represented as an RPSL perception graph (see Chapter 3).

The second one from the perception graph to the Marker detection feature with the label RESOLVES encodes that a particular perception graph implements a higher-level description of a functionality. In addition, properties of the nodes can be attached in the form of key-value pairs such as the names of nodes, e.g.  $\langle Name, Aruco \rangle$ . The labeled-property graph model also supports the late binding of domain information in the form of uninitialized properties and nodes. For example, in the deployment phase a description of the location and name of some executable is provided (represented as a node in the graph). Depending on the deployment infrastructure or robot software framework used in the application, this node is labeled as `DeploymentFile`, `RosLaunch` or `SystemDServiceFile` with a link to some process profile node which encodes execution properties of the deployed process. As the properties of the latter node are not known before deployment, the values of the key-value pairs remain blank, e.g.  $\langle MemoryUsage, - \rangle$  and  $\langle StartTime, - \rangle$ .

### 5.3.2. From Domain Models to Graph Models

Until now, the labeled property graph described in the previous section is a general concept and not integrated, for example, into the developer's workflow. In order to achieve a programmatic integration, domain models need to be (semi-)automatically translated to elements of the labeled property graph (see Equation 5.1).

To achieve this, one needs to assess the domain models in terms of their structural entities and relations which yields a *graph model*. The assessment involves the question which entities of the domain models should be represented as nodes, labels, properties and which should be represented as edges (relationships). For software-related domain models in robotics, such as component models, coordination models (e.g. state-charts) or deployment descriptions the translation step is obvious. The core entities of interest should be represented as nodes, whereas edges are used to represent connections between entities. For example, states in a state-chart are represented as nodes and transitions between states are represented as edges. Following such an approach also paves the way for (semi-)automatic translations of domain models to labeled property graph representations.

It is worth noting that such an assessment needs to be performed for each DSL where a graph model should be created. For example, in Section 5.5 a graph model for RPSL and DepSL is defined.

Depending on the application scenario a developer also needs to decide which and how much information is translated from a domain model to a graph model. Fortunately, the labeled property graph model does not impose any constraints here. Developers can either decide to completely translate domain models to graph concepts or to partially translate them, where a node simply contains a property which points to the location of the more detailed domain model, for example, on the disk.



## 5.4. Semantic Querying of Domain Models

The question posed in Section 5.2, namely which components are executable on the robot's platform, is a *semantic query*. As in the context of semantic web technologies (see Bailey *et al.* [164]) a semantic query denotes the process of retrieving implicitly defined information based on the structural information expressed by the underlying representation. Similarly, as for related approaches (e.g. OWL ontologies, topic maps, and so forth [164]), the underlying representation in this thesis is a graph.

In order to answer the previous question, one needs to check which components have a deployment description which in turn points to the robot's platform (see Figure 5.2). Realizing this checking can be achieved by standard graph traversals where information is collected and checked while the graph is being traversed. Although these traversals are application-independent they can be interpreted in a domain-specific way. That is, what type of information is collected and checked depends on the instantiation of the graph (e.g. types of nodes, properties, and so forth) which in turn depends on the meta-model of the translated domain model. This is what makes a graph such a powerful representation to implement semantic queries.

In general, semantic queries can be raised both by humans (domain experts) and robots. Domain experts usually raise queries at design time in order to retrieve meta-information about domain models (e.g. the coupling and cohesion ratio among model elements, model duplication checks, and so forth). Robots on the other hand typically raise queries to retrieve information what is relevant to the current context condition. For example, which components are required to achieve a certain task. The queries raised at run time typically also incorporate information which has been bound at run time (see Section 5.3).

## 5.5. Case Study

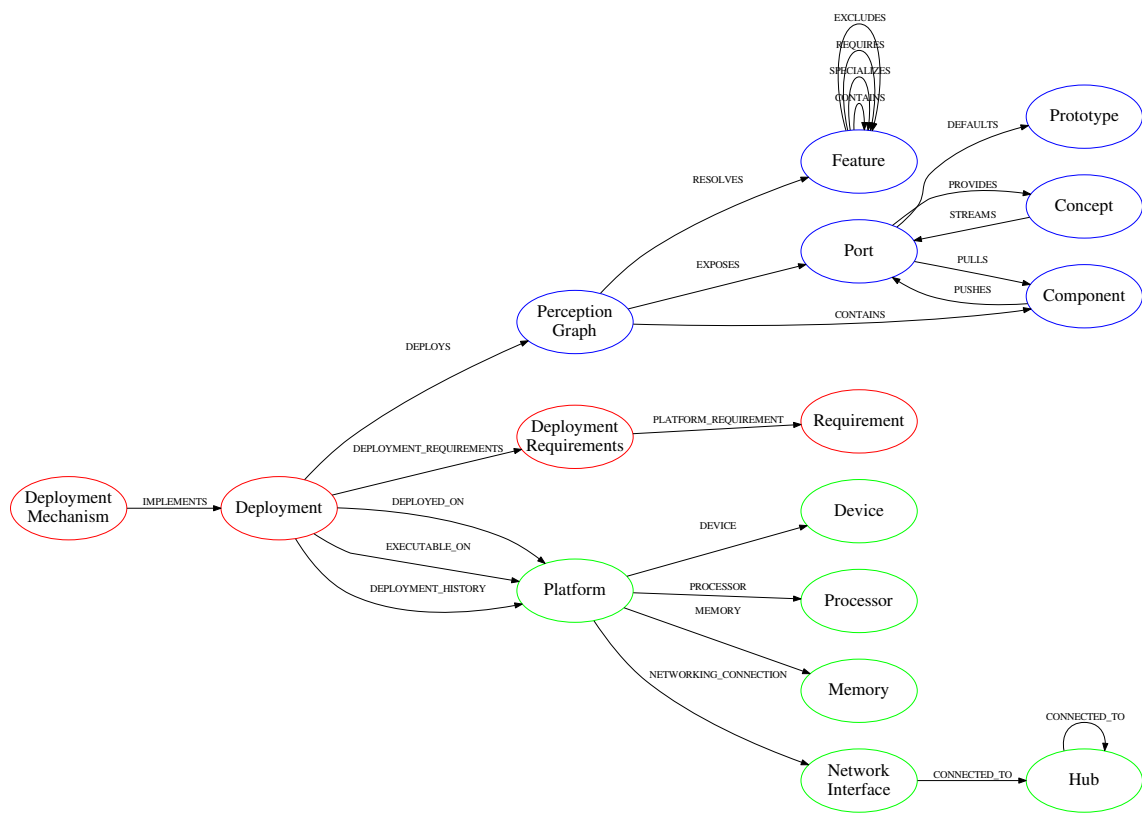
As the core contribution of this chapter is not a single, monolithic system, but a general approach to store, compose and query domain models, it is hard to quantitatively assess the contribution. Therefore, this section reports on some lessons learned while realizing the scenario described in Section 5.2.

A model-based development approach has been employed to realize the application described in Section 5.2. More precisely, the RPSL and DepSL (see Chapter 3) were used to create domain models representing *a)* different marker detection configurations suitable for varying lighting conditions, *b)* their associated deployment descriptions encoding name and location of the executable and *c)* the computational hardware of the quadcopter and remote platform used

in the application. Those domain models are then stored in a graph database and queried for the sake of adapting the system at run time.

### 5.5.1. Implementation Approach

In order to implement the labeled property graph the open-source graph database management system neo4j [165] is utilized. The neo4j graph database exposes the property-graph model with nodes and relationships as first-class citizens. By composing nodes and relationships into arbitrarily connected structures it is not necessary to depend on join operations (relational databases) or other customized operations (document-oriented databases) to infer connections between entities. This allows domain experts to both preserve the structure of domain models and to compose domain models through labeled edges. Similarly to other professional database management systems, neo4j supports full ACID (Atomicity, Consistency, Isolation, Durability) transaction rules. In addition, database drivers for several programming languages (e.g. Java, Python and Ruby) are available, which facilitates not only the integration of a robot system, but also embedding it in DSL development tools and frameworks.



**Figure 5.3.** A visualization of the labeled property graph model for RPSL and DepSL showing the different nodes and labeled edges.

### 5.5.2. From RPSL and DepSL Domain Models to Graph Models

Before RPSL and DepSL domain models could be stored in a graph database, a graph model capturing the essence of RPSL and DepSL needs to be created (see Section 5.3.2). In Figure 5.3 an excerpt of the graph model both for RPSL and DepSL is shown. The model depicts a labeled, property graph with the core structural elements of RPSL and DepSL as nodes (e.g. `PerceptionGraph`, `Feature`, and so forth). Those nodes are connected by labeled edges describing the meaning of node relations. For example, the label `RESOLVES` denotes that a `Feature` is realized by a `PerceptionGraph`.

It is worth noting that not every structural element of RPSL and DepSL is represented as a node in the graph model. For example, the graph model lacks nodes representing the concept of `Input-` and `OutputPorts` (see Chapter 3) as those can be encoded as directed edges between `Port` and `Component` nodes. More precisely, having labels such as `PUSHES` and `PULLS` allows domain experts to express the port directionality in a very comprehensive manner. In addition, a single edge can be attached with multiple labels where the meaning differs significantly. For example, the labels `EXTENDS` and `REQUIRES` denote whether a `Feature` is a specialization of another feature or whether it depends on another feature.

In order to programmatically realize the translation from RPSL and DepSL domain models to the required `neo4j` representation the `Neo4j.rb` [166] object-to-graph mapping module is used. The module implements the Active Record pattern [167] and facilitates the creation of graph models and their instantiation in terms of graph database operations (e.g. insertion of nodes, edges, update of properties, and so forth).

### 5.5.3. Semantic Querying of RPSL and DepSL Domain Models

In order to implement semantic queries the `neo4j` query language `Cypher` is used. Having `Cypher` at their disposal, developers can declaratively query and update the graph database. The general principle of a `Cypher` query on the graph is that of matching a graph pattern of the following form:  $(A) \rightarrow [R] \rightarrow (B)$ . Here,  $A$  and  $B$  are nodes and  $R$  is an edge. By using such a statement in a `MATCH` clause the graph database retrieves those nodes and edges where there is an outgoing relationship (edge) between  $A$  and  $B$  of type  $R$ . This pattern is the general principle of queries which can be arbitrarily extended and combined with directed, undirected, optional and multi-step relationships among nodes and more advanced clauses such as `RETURN` for node/edge retrieval and `CREATE` for node/edge creation to name a few.

In Figure 5.4 a simple, yet realistic example is given. Here, the domain model of Figure 5.2 where features, components and so forth are represented (stored) as nodes and connected through labeled edges encoding the relations among these nodes is queried. The query then retrieves deployment descriptions for those components that have a relation to the marker

```
MATCH (f:Feature)-[*]-(c:Component) WHERE f.name = 'MarkerDetect'  
MATCH (p:Platform)<-[:EXECUTABLE_ON]-(d:Deployment) WHERE p.name = 'Remote'  
RETURN d;
```

Figure 5.4. A Cypher query involving different domain models.

detection feature and are executable on the remote platform. This is achieved by making relations between domain models explicit. It is important to note that the EXECUTABLE\_ON relation links the platform domain with the deployment domain which subsequently allows the filtering of the results in both domains through additional WHERE clauses. As the names of edges and nodes appear directly in the queries they need to be meaningful and consistent. This fact needs to be considered during the creation of a graph model (see Section 5.5.2). For example, naming the relation EXECUTABLE\_ON only makes sense when the edge is directed from the deployment description to a platform description and not vice versa.

Generally speaking, the case study demonstrates the need to query domain models originating from different functional domains and development phases. By employing relatively simple queries (see Figure 5.4) a developer can incrementally extend them (see Figure 5.5) in order to derive the information required for the task at hand. In the same way a developer can cope with growing graph databases by concatenating several MATCH clauses. Also additional constraints can easily be included through additional and advanced WHERE clauses.

In the context of the case study the following queries have been developed:

- To retrieve those components required to realize the marker detector feature, but which are also deployable on the remote computer (see Figure 5.4).
- To retrieve those platforms meeting the memory demand of the perception graph realizing the marker detection (see Figure 5.5).
- To check whether the marker detector feature can be deployed with different camera resolutions. That is, whether or not camera components (see Figure 5.2) with different resolution properties are part of a perception graph realizing the marker detector feature.
- To retrieve those components required to realize the marker detector feature, but which have been deployed in the past and their average *memory usage* was below a certain threshold.
- To check whether the CPU workload would exceed an application-defined limit when the marker detector and the flight control were both deployed on the same platform. CPU workload profiles of software components are either acquired at run time or have been annotated at design time.

```
MATCH (f:Feature {name: 'TactileSlip'})-[*]-(c:Component)
WITH DISTINCT c
WITH SUM(c.memory_demand) as MEM
MATCH (p:Platform)-[:EXECUTABLE_ON]-(d:Deployment)
WHERE p.memory_available >= MEM
RETURN p;
```

Figure 5.5. A Cypher query retrieving the platforms meeting the memory requirements.

#### 5.5.4. Run Time Overhead

One might argue that the application of a graph database system introduces a significant run time overhead on the overall system. In order to investigate this question the experiment described in the following paragraph has been conducted.

A graph database was populated with  $N$  domain models (see Table 5.1) of potential marker detectors, all of them varying in configuration properties. Here,  $N$  does not denote the number of nodes in the graph. In fact, the number of nodes is approximately three times  $N$  as additional nodes are integrated, for example, to express deployment and platform information (see Figure 5.3).

On a standard personal computer<sup>1</sup> with Linux Ubuntu 14.04, Version 2.2.5 of the neo4j graph database different scenarios were replayed. The scenario consists of logged data, namely the current lighting condition and the available memory available on the two platforms (see Section 5.2). During each simulation step, a component decides which marker detector shall be executed on which platform. The decision is driven by the available memory and by the current lighting situation. By employing the query shown in Figure 5.5 the component derives the platforms where the marker detector can be deployed without violating the memory demands. Subsequently, the component employs the algorithm described in [161] to find out which marker detector is suitable for the current illumination condition. Once a marker detector and platform is selected, the component stops the current marker detector and starts the new one if it is not already being executed. In addition, some process meta information about the deployed marker detector is stored, namely the process start time and the process ID. This information is inserted and linked to the corresponding marker detector in the graph database. In summary, two graph database operations are performed, namely insertion and querying.

The experiment aims to investigate the run time overhead of those operations. To this end, the time to perform graph database operations were measured. Those measurements are related to the timing measurements of the adaptation operations, namely starting and stopping an executable. Here, the executable is a C program implementing the marker detector. For each  $N$ ,

<sup>1</sup>Intel Core i7-3632QM CPU 2.2GHz x 8 with 8GB RAM

	Graph DB Operations				Adaptation Operations			
	insert		query		start		stop	
$N$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$
10	26.861	5.936	4.083	1.274	1.491	2.299	0.066	0.023
100	27.766	6.385	7.323	3.198	1.602	1.906	0.072	0.025
1000	26.599	5.942	8.443	2.937	1.670	2.731	0.065	0.021
10000	26.308	6.208	7.901	2.476	1.651	3.345	0.074	0.082

**Table 5.1.** Timing results of the graph database operations versus adaptation operations given in milliseconds.

the experiment was repeated 100 times and Table 5.1 reports mean ( $\mu$ ) and standard deviation ( $\sigma$ ) of those timings. As seen in Table 5.1, the insertion operation seems to be independent of  $N$ . This can be explained with the fact that no graph traversal is required as the exact location of insertion is known, namely next to the selected marker detector. Interestingly, the impact of  $N$  on the timing of the query operation is rather limited as there is only one major increase from  $N = 10$  with  $4ms$  to  $N = 100$  with  $7ms$ . To which extent the queries can be optimized through caching or other mechanism remains to be investigated and also depends on application-specific graph structures. In summary, the graph operations are more costly than the adaptation operations. However, this also depends on how the adaptation operations are implemented. For example, not preserving the state of a component as done in this experiment is faster than saving the state before stopping the component.

## 5.6. Related Work and Discussion

The application of databases in robotics is not a new concept. In fact, Niemueller *et al.* [168] have shown that it is feasible to apply a document-oriented database like MongoDB [169], even for logging raw sensor data and analyzing robots' behavior in retrospect. Also knowledge-enabled and ontology-based approaches such as KnowRob [170] [157], RoboBrain [158] or the OpenRobot Ontology (ORO) [159] rely on knowledge bases to store and query specifications of robots, their capabilities, tasks and environments. Those approaches are complementary to the graph databases proposed in this chapter. The effort bridges the gap between *a*) the knowledge descriptions about robot's capabilities, tasks, and so forth; and *b*) the knowledge specifications about the *implementation* of the software that solves the tasks.

In [99] the RobotML language is introduced which enables a domain expert to specify robot system architectures, communication mechanisms and the behavior of components. Interestingly, the development of RobotML was based on an ontology supporting the DSL designer by providing concepts specific to the robotic domain. The process of defining an ontology is

somehow related to the process of defining a labeled property graph. However, in RobotML the ontology solely supports the DSL domain analysis, whereas in the approach presented in this chapter the domain models carry their meaning into the robot's run time.

It is worth noting that working with labeled-property graphs reveals some analogy with the four-layered metamodeling hierarchy as described by the Object Management Group's (OMG) Meta-Object Facility (MOF) [28]. In the MOF the M0 layer are real-world instances which are represented by models on the M1 layer. The M1 layer conforms to a metamodel on the M2 layer which in turn conforms to a meta-metamodel on layer M3. It can be argued that a graph, such as the one depicted in Figure 5.2, is just a different representation of one or more M1 models. Therefore, the set of available properties  $\mathcal{P}$  and labels  $\mathcal{L}$  from meta-models on the M2 layer can be derived. Consequently, the graph in Equation 5.1 aligns with the meta-metamodel of the M3 layer. Obviously the graph structure itself does not constrain the attachment of properties and labels to the nodes and edges. In the context of this thesis the well-formedness and validity of specifications is achieved by the approach presented in Chapter 3. Only then are the valid models transformed into the graph representation. To which extent M1, M2 and M3 models should be stored in the graph database remains to be investigated and also depends on application requirements.

In robotics [70] [171] and software engineering [96] the authors have already investigated the application of software-related models for robots at run time. They demonstrated how software-related models can be employed to resolve dynamic variability faced at run time such as changing environments and decreasing resources. In order to derive adaptation actions, different adaptation principles are employed (e.g constraint optimization methods [171]). However, irrespective of the underlying adaptation principle, all run time adaptation approaches in robotics need to access and query software-models originating from different domains and process phases.

Therefore, the presented approach is a complementary building block for developing adaptive robot software architectures. More precisely, the work relates to the *knowledge* building block where domain models are placed of the well-known MAPE-K (Monitor, Analyze, Plan, Execute and Knowledge) [163] reference architecture. This relation will be discussed in more detail in Chapter 6.

By storing such models in a graph database domain experts can, use their familiar, domain-specific tools and notations and use a common interface to query these models. Due to the inherent graph-based nature of most models, the storage in a graph database provides an integration point for further model-based approaches, for instance, those surveyed in Chapter 2.

In [172], the *ModelBus* tool integration technology is presented. ModelBus targets large-scale, distributed model-based software development environments and provides services to orchestrate and manage modeling artifacts. Examples of those services are model versioning,

model exchange, and so forth. Unlike the approach presented in this chapter ModelBus lacks the means to query the modeling artifacts.

The case study demonstrated that a better understanding of the implications of models on the run time architectures – as seen in the application – is required. For example, which aspects of the run time system should be represented and how to implement the required monitoring facilities such as probes for performance measurements and so forth. In order to tackle some of these issues, Chapter 6 proposes a reference architecture which structures the development of adaptive applications as those considered in this chapter.

Until now, very few systems in robotics (see Biggs *et al.* [173]) have been implemented by following a completely model-based development process. One reason for this could be the *DSL cacophony* problem (see Section 3.8), meaning that a vast number of (very relevant) DSLs exists (see Chapter 2), but their adoption and integration into an overall system remains challenging. To this end, a common interface of graph databases could offer a means to lower the burden of this integration effort.

## 5.7. Summary

The growing interest in software engineering for robotics has already resulted in models and DSLs (see Chapter 2). Thus, a logical next step is to apply these models at run time by granting robots access to software-related models. This chapter proposed labeled property graphs and graph databases as powerful means to achieve this step by storing, composing and querying domain models which in turn facilitates the development of semantic queries. Those queries make implicitly defined information accessible and raise the robots' awareness of its software capabilities.



## Chapter 6.

# Deploying Robot Perception Systems

*“Controlling complexity is the essence of computer programming.”*

— Brian Kernigan, 1942—

### 6.1. Introduction

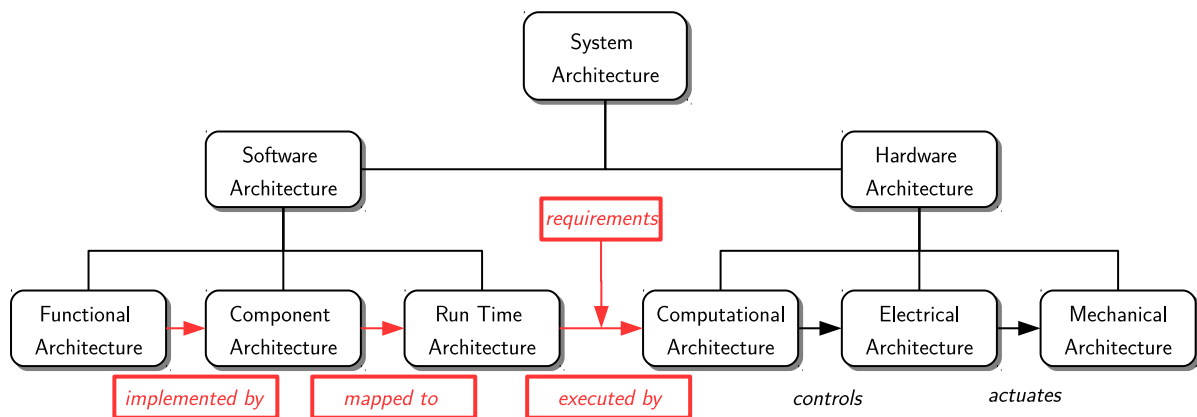
Deploying complex robot perception systems on real robots and getting it to run reliably is a challenging task to be performed by developers in the robot application development process (see Chapter 2). In order to deploy robot perception systems, domain experts need to go through each architectural view (see Figure 6.1) and answer several questions. For example, which components are realizing a certain functionality and which shall be deployed as processes or threads on which computational platform? Such a question can be considered as a verbalization of the highlighted links among architectural views as depicted in Figure 6.1.

In order to answer those questions, knowledge about different architectural aspects – which can be expressed with RPSL and DepSL (see Chapter 3) – is fundamental for deploying robot perception systems. Furthermore, deployment includes also two additional activities, namely deployment planning and execution. According to the OMG deployment specification [174], deployment planning “...is an activity that takes the requirements of the software to be deployed, along with the resources of the target environment on which the software will be executed, and decides...how and where the software will be run in that environment”.<sup>1</sup>

It is important to note that the complexity of the planning task increases significantly when having to cope with heterogenous, networked hardware on large-scale integrated robots. For those robots the question which platform should execute a given set of processes and threads

---

<sup>1</sup>In the context of this work, the software to be deployed are perception graphs and the target environment is expressed in terms of computational architectures (see Chapter 3). From now on the term platform will be used to describe a computational architecture which provides the resource needed to deploy and execute perception graphs.



**Figure 6.1.** The architectural views employed for the design and development of robot perception systems as introduced in Chapter 3. For deploying robot perception systems the links visualized in red are of importance.

is challenging to answer, as there might be several options. Thus, deployment planning as defined above is required.

Once a deployment plan is found, it needs to be executed and possibly modified, as resource conditions are likely to vary in dynamic environments. For example, sensors might break, resources such as working memory will decrease, and so forth. Those changes should not be ignored as they can lead to erroneous and dysfunctional deployments.

Thus, the core problem investigated in this chapter is: *How to (re)-plan and execute deployments of robot perception systems in the presence of changing resource conditions?* This involves *a)* declaring deployment requirements, *b)* accessing the knowledge required for deployment, *c)* creating and executing a deployment plan, *d)* monitoring the target environment, and *e)* re-plan deployment in presence of violated requirements.

This chapter introduces three contributions for deploying robot perception systems.

- Firstly, DepSL (see Chapter 3) is expanded with means to express not only resource-specific deployment requirements, but also to represent information relevant for deployment execution.
- Secondly, a reference architecture for deploying robot perception systems is proposed. The reference architecture provides a component-based template solution integrating not only the knowledge required for deployment (see Chapter 5), but also all the other means required to carry out deployment like monitoring.
- Thirdly, a deployment algorithm is developed to find an assignment of one or more perception graphs to platforms. The algorithm ensures that the deployment requirements of perception graphs are met.

The remainder of this chapter is structured as follows. Section 6.2 proposes the general elements of the reference architecture for deploying perception graphs. Section 6.3 introduces a case study by instantiating and exemplifying the proposed reference architecture. Section 6.4 discusses related work and Section 6.5 summarizes the core findings of this chapter.

## 6.2. Reference Architecture

This section proposes a reference architecture for deploying robot perception systems. According to Taylor *et al.* [115] a reference architecture is defined as “...the set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variations”. Introducing a reference architecture for robot perception systems is necessary as deployment is a recurring, yet often underestimated activity which is often prone to errors. Thus, a reference architecture paves not only the way to organize deployment, but also to provides a template solution for recurring applications. It is worth to stress that such a template solution is applicable and tailorable for multiple applications.

In this work the proposed reference architecture is depicted in Figure 6.2 as a component-based diagram. The diagram assembles variable and stable components, both providing and requiring interfaces. The former are interchangeable for different applications whereas the latter can be used without changes for different applications.

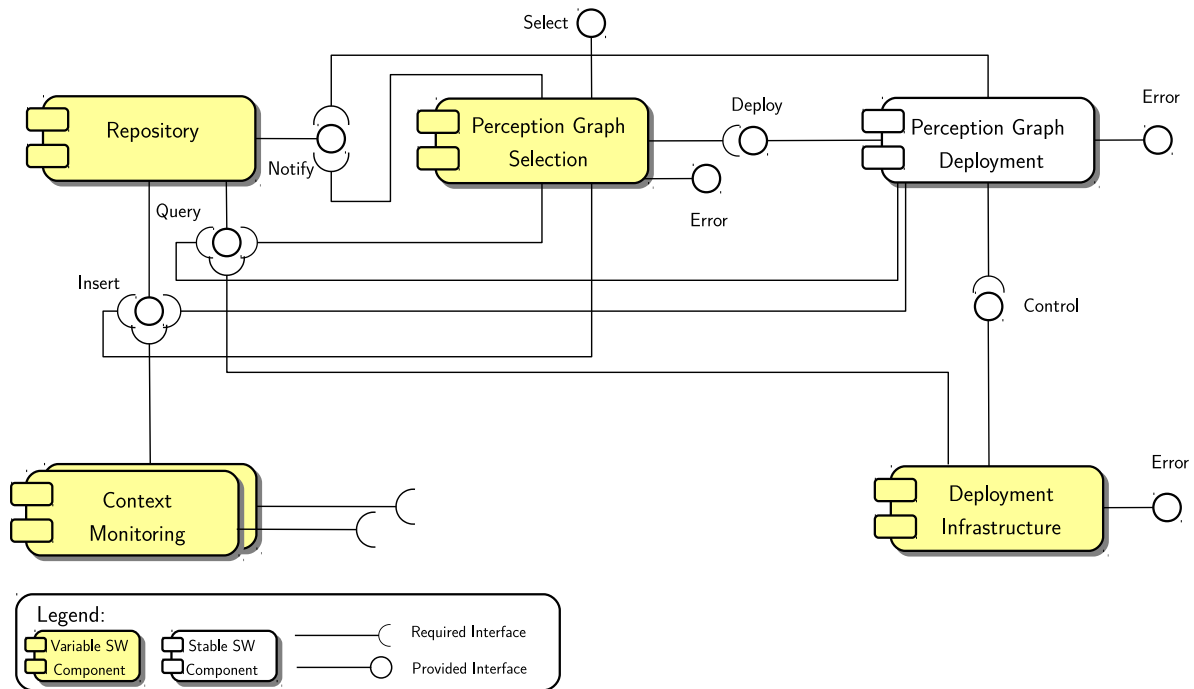
There are two principal design decisions underlying the definition of the reference architecture.

- Firstly, the knowledge relevant for deployment is explicitly stored in one place, thus making deployment knowledge accessible for other components.
- Secondly, deployment is performed in two steps. The first step deals with selecting a set of perception features and corresponding perception graphs suitable for the task at hand. The second step deals with deciding where (on which platform) the selected perception graphs will be executed. Having such a step-wise approach enables developers to implement application-specific perception graph selection components (see Chapter 7) without dealing with deployment concerns.

In the following sections each component of the proposed reference architecture is described.

### 6.2.1. Repository

The repository (see Figure 6.2) plays a central role in the reference architecture as it contains all the knowledge required for carrying out deployment activities. The knowledge can be classified in design time and run time knowledge. Examples for the former are domain models expressing knowledge about platforms, perception graphs and so forth, and examples for the



**Figure 6.2.** The reference architecture for deploying robot perception systems depicted as a UML-like component diagram with variable (interchangeable) and stable software components.

latter are information about the current memory usage of certain components or the availability of certain sensors.

The repository component provides three interfaces, namely *Insert*, *Query* and *Notify*. The *Query* interface is used to retrieve information about design and run time knowledge. The *Insert* interface is used to create and update information in the repository, and the *Notify* interface is employed to inform other components about those changes in the repository.

Having all the information relevant for deployment in one central place fosters a development practice where deployment knowledge is explicitly declared and not implicitly encoded and scattered among components. It is worth mentioning that the repository can be implemented in multiple ways by applying different technologies ranging from traditional relational databases, document-oriented database and graph databases (see Chapter 5) to highly engineered modules providing application-specific APIs.

### 6.2.2. Context Monitoring

One or more context monitoring components are composed in the reference architecture in order to provide the contextual information needed to select (see Section 6.2.3) and deploy (see Section 6.2.4) perception graphs. To this end, context monitors collect – hence requiring additional interfaces (see Figure 6.2) – and interpret all the measurements required to infer the

current state of robots' environment, platform (both mechanical structure, sensors, actuators and computational elements) and intelligence like tasks, behaviors and skills [175]. Context monitors make this information accessible to other components by inserting them in the repository (see Section 6.2.1).

Very often robotic systems already acquire those measurements and take important decisions based on their values. However, the acquisition of these measurements is usually hard-coded in the implementation of the components that reason about them. Thus, introducing dedicated context monitoring will make components more reusable. In general, context monitors are application-specific and employ various context representations (e.g. logic-based vs. probabilistic-based approaches) [176] suitable for the task at hand.

### 6.2.3. Perception Graph Selection

The perception graph selection component, or just selector, is a variable element in the reference architecture and in charge of selecting one or more perception graphs suitable for the task at hand. To this end, activation is either triggered – in a reactive manner – by changes of context conditions using the `Notify` interface or by higher-level components via the `Select` interface provided by the component. In order to select the perception graphs suitable for the task at hand, different methods and algorithms like rule-based approaches or more advanced constraint solvers could be used. Depending on the employed selection algorithm, different types of queries are eventually applied to retrieve the information required to carry out selection.

### 6.2.4. Perception Graph Deployment

As a stable, application-independent element in the reference architecture the perception graph deployment component, or just deployer, is responsible for deploying one or more perception graphs. To do so, the deployer provides an `Deploy` interface which is used by the selector to inform the deployer which perception graphs have to be deployed. After receiving such a request, Algorithm 1 is used to find those platforms which meet the deployment requirements for the given graphs. Note, more details about Algorithm 1 are given in Section 6.3. In case no platform is suitable, for example, if no platform satisfies the memory requirements, an error is reported via the `Error` interface. The implementation of the `Error` interface depends significantly on *a*) how the reference architecture is realized in an application context, and *b*) how the overall error management is implemented (see e.g. Garcia *et al.* [177] for a survey).

### 6.2.5. Deployment Infrastructure

The deployment infrastructure, or just infrastructure, is responsible for bringing up and taking down perception graphs. The infrastructure provides a `Control` interface which is used by the deployer to inform the infrastructure which graphs on which platforms should be started respectively stopped. In order to execute perception graphs on platforms the infrastructure requires additional, execution-relevant knowledge like how perception graphs are mapped to executable primitives, for example, processes and threads or where the binaries of perception graphs are located. In case perception graphs can not be executed, for example, if the binary is not available, an error is reported via the `Error` interface.

Note, the deployment infrastructure component is interchangeable, thus different mechanisms to execute perception graphs can be integrated. For example, one could integrate deployment tools available in robotic software frameworks such as `roslaunch` [178] or employing init systems such as `systemd` [179] that are capable of manage several processes on Unix systems.

## 6.3. Case Study

This section exemplifies the reference architecture introduced above with the help of a case study, namely the slip detection domain example (see Section 3.3) has been implemented.

In the context of the domain example, Sanchez *et al.* [16] suggest that the actions and motions performed by the robot during grasping should be taken into account during slip detection for improved performance. Thus, an in-hand slip detection architecture should be able to adapt to the current robot's actions at run time (cf. finding **F4** in Section 3.4). In order to achieve such adaptive behavior one requires not only to retrieve the current robot action context, but also to select and deploy the most appropriate perception graph for the task at hand.

The reference architecture proposed in Section 6.2 provides an architectural blueprint to realize such an adaptive system. In the following sections the realization of the reference architecture for the slip detection case study is described.

### 6.3.1. Repository

Both the RPSL and DepSL (see Chapter 3) were employed to create domain models representing the knowledge relevant for deployment. Those domain models represent not only the three different slip detectors as perception graphs (see Section 3.2.1), but also their associated deployment descriptions and the computational hardware of the Care-O-bot 3 (see Figure 1.1) service robot.

In order to enable the deployer (see Section 6.2.4) to identify suitable platforms, requirements have to be specified. To enable domain experts to express platform requirements, DepSL has been expanded by six requirement types. Those types are proposed by the OMG deployment specification [174] and described in the following paragraphs.

**Quantity.** This requirement allows to express a certain number of required elements. For example, a certain number of tactile sensors (cf. finding **F1** in Section 3.4) connected to a platform.

**Capacity.** This requirement allows to express a certain capacity of a platform resource which can be consumed by one or more perception graphs. For example, the size (capacity) of working memory.

**Minimum.** This requirement allows to express an acceptable lower bound of a platform property. For example, the minimum clock rate of a CPU.

**Maximum.** This requirement allows to express an acceptable upper bound of a platform property. For example, the maximum latency of a networking connection.

**Attribute.** This requirement allows to express the existence of certain platform properties. For example, a certain hardware version of a sensor or a specific operating system installed on the platform.

**Selection.** This requirement allows to express a set of elements where one or more should be available on the platform. For example, different sensors of the same modality (e.g. RGB-D), but from different manufacturers.

In the context of the case study for each perception graph, namely force, tactile and combined slip detector, deployment requirements are specified. Those are described in the following enumeration.

- The force slip detector should be deployed on a platform where the force sensor is connected to (cf. finding **F8** in Section 3.4).
- The tactile slip detector should be deployed on a platform where all tactile sensors are connected to (cf. finding **F8** in Section 3.4). As nine tactile sensors are required for the tactile slip detector the quantity type is employed to express this requirement.
- The combined slip detector should be deployed on a platform with at least 250 MB working memory, thus the capacity type is used to express this requirement.

It is important to note that the repository in the context of this case study is realized as a graph database (see Chapter 5). Thus, both RPSL and DepSL domain models have been translated to a labeled property graph. As shown in Figure 7.4a an excerpt of the graph expressing the case study is depicted. Here, the three different perception graphs are resolved by deployment models expressing – amongst other things – the above mentioned requirements. Note, the

deployment model resolving the combined slip detector is connected to the deployment models of the force and tactile slip detector. This connection is labeled `:DEPEND_ON` and expresses that if the combined slip detector is deployed also the force and tactile slip detector should be deployed. The connection between those models is automatically derived from the fact that a domain expert already declared their dependency on a perception feature level (see Section 3.5.1). Figure 7.4a depicts also the three different platforms which are available on the Care-O-bot 3. Establishing the links between deployment and platform models is the task of the deployer (cf. Section 6.2.4). Nevertheless, sometimes domain experts are already at design time capable to establish fixed links between deployment and platform models. For example, if a domain expert knows that solely one platform fulfills the requirements a fixed edge labeled `:EXECUTABLE_ON` is created (cf. Chapter 5). Obviously, such a fixed assignment should be done with caution as it limits the deployability of perception graphs.

Although not shown in Figure 7.4a the repository contains information required by the deployment infrastructure component (see Section 6.3.5).

### 6.3.2. Context Monitoring

The main objective of the context monitor is to retrieve the current action performed by the Care-O-bot 3. Three different actions described in the following paragraphs are detectable by the context monitor.

**grasp:** The fingers of the gripper close to hold the object.

**move\_base:** The robot's base moves while holding the object.

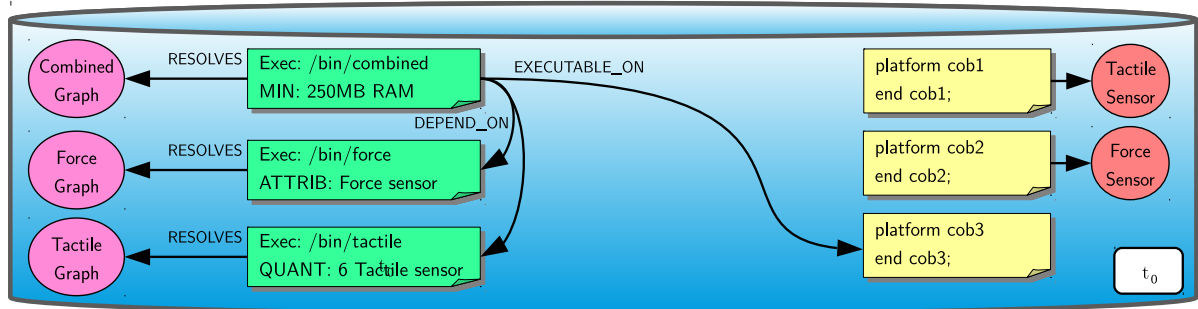
**release:** The fingers of the gripper open to release the object.

In order to retrieve the action context, symbol grounding is performed by employing the Conceptual Space knowledge representation framework (see Section 3.5.5). The framework allows to ground different action contexts through the notion of concepts and dimensions. Here, the measurable dimensions are the joint values (velocities) for each joint. Typical values for each concept are expressed in the form of prototypes (cf. Figure 3.11). For example, the prototypes for the `release` concept has only zero-values for the base velocities whereas the finger joints are non-zero. During run time, the context monitor then computes for each joint state sample the closest matching prototype by employing the Euclidian distance as a metric. Subsequently, the current action context is updated in the repository.

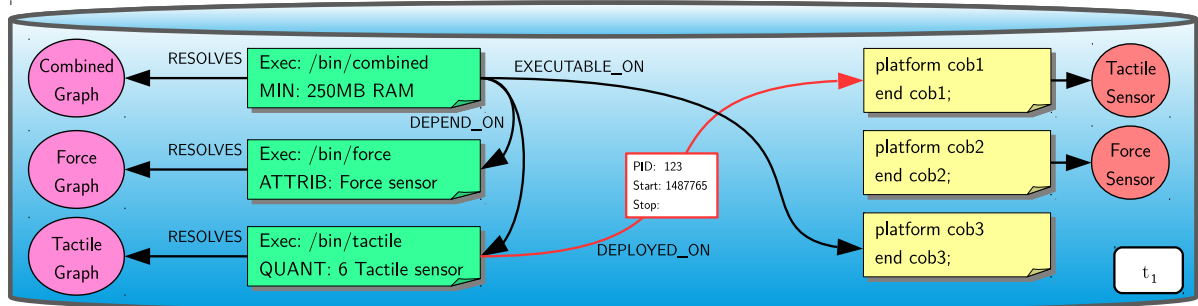
### 6.3.3. Perception Graph Selection

In order to select a slip detector which is appropriate for the current action context a simple, yet powerful rule-based approach is applied. During design time a set of decision rules

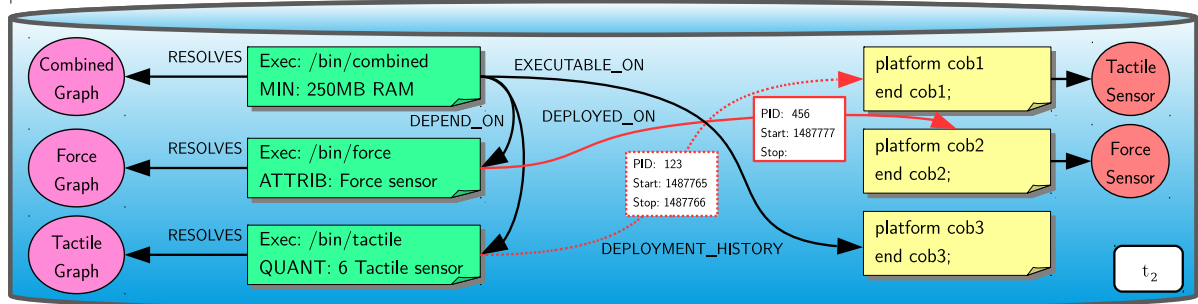




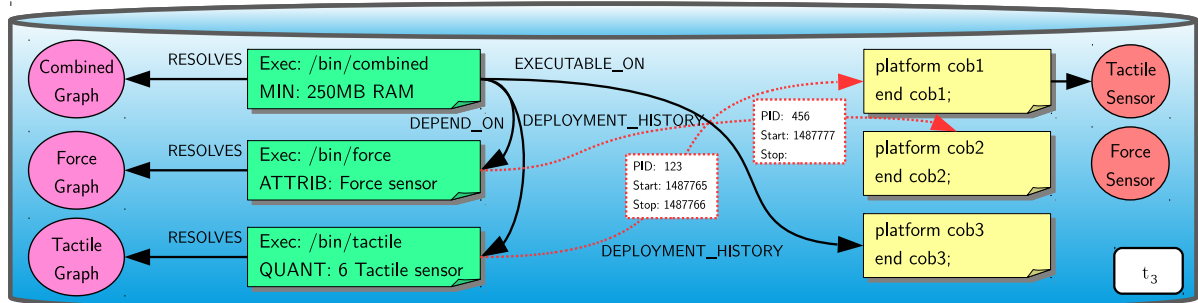
(a) An excerpt of the repository at time  $t_0$ .



(b) An excerpt of the repository at time  $t_1$ .



(c) An excerpt of the repository at time  $t_2$ .



(d) An excerpt of the repository at time  $t_3$ .

**Figure 6.3.** Snapshots of the graph-based repository during the case study. Note, the snapshots are an excerpt of the complete graph database and focus on how the links between the deployment description (and their corresponding requirements) and the available platforms are established during the case study.

have been devised. Here, the action context is part of the rule condition and the selection of a slip detection perception graph is part of the rule body.

In order to identify these rules, Sanchez *et al.* [16] evaluated the different slip detectors using different grasp shapes, namely a grasp that uses all three fingers of the robotic hand and one that only uses two fingers. Three different objects and three different actions (see Section 6.3.2) were used in the experiments.

The performance of each slip detector varies considerably depending on the action, for example, the tactile slip detector outputs a slip whenever grasping an object. Contrary, the force slip detector achieved perfect accuracy for detecting actual slips, however its performance is poor when no slippage occurs, for example, when the robot base is moving. More details about the actual experiments are given in [16].

---

**Algorithm 1** Finding a platform satisfying the deployment requirements.

---

```

1: function Deploy.perceptionGraphs( $G$ )            $\triangleright G$  is the set of graphs to be deployed.
2:   for each  $g_i \in G$  do
3:      $d_i \leftarrow$  Query.getDeploymentInformation( $g_i$ )
4:     if  $d_i \neq \emptyset$  then
5:       if Query.hasFixedDeployment( $d_i$ ) then
6:          $p \leftarrow$  Query.getFixedPlatform( $d_i$ )
7:         if Control.start( $g_i, p$ ) then
8:           return success
9:         else
10:          Error.deploymentFailed( $g_i, p$ )
11:          return error
12:       else
13:          $C \leftarrow$  Query.getConstraints( $d_i$ )    $\triangleright C$  is the set of deployment constraints.
14:          $P \leftarrow$  checkValidity( $C$ )            $\triangleright P$  is the set of acceptable platforms.
15:         if  $P \neq \emptyset$  then
16:           if not Query.isDeployed( $g_i, P$ ) then
17:             if Control.start( $g_i, p_k \in P$ ) then
18:               return success
19:             else
20:               Error.deploymentFailed( $g_i, p_k \in P$ )
21:               return error
22:           else
23:             Error.noAcceptablePlatforms( $g_i$ )
24:             return error
25:         else
26:           Error.deploymentInformationMissing( $g_i$ )
27:           return error

```

---

### 6.3.4. Perception Graph Deployment

This section explains the deployment algorithms shown in Algorithm 1 and 2 by utilizing the case study.

At time  $t_0$  the Care-O-bot 3 service robot is standing in the kitchen and has a mug in it's hand. At this point in time the graph database is composed of nodes and edges as shown in Figure 7.4a. Subsequently, a user requests the robot to deliver the mug to the living room.

At time  $t_1$  the robot starts moving it's base, hence the context monitor (see Section 6.3.2) detects the `move_base` action context and updates the repository. Based on the context update, the selector (see Section 6.3.3) requests the tactile slip detector to be deployed. Thus, the selector calls the `perceptionGraphs()` method of the `Deploy` interface provided by the deployer.

As shown in Algorithm 1 for each selected perception graph  $g_i$  corresponding deployment information  $d_i$  is retrieved. That is, the node which resolves the perception graph is retrieved. In case no deployment information for  $g_i$  is available an error is reported. Subsequently it is checked whether or not a fixed deployment is given. That is, it is checked whether  $d_i$  has an edge to a platform which is labeled `EXECUTABLE_ON`. As shown in Figure 7.4a no fixed deployment for the tactile slip detector is provided. Thus, all the deployment requirements of  $d_i$  are retrieved in order to find an acceptable platform meeting the requirements. The `checkValidity()` method takes the requirements and returns those platforms  $P$  satisfying them. In the context of this example, `checkValidity()` checks which platform provides nine tactile sensors as those are required for the tactile slip detector. Basically two situations can occur, namely no platform is meeting the requirements or one or more platforms meet the requirements. In the former case an error is reported and for the latter case it is checked whether or not  $g_i$  is already deployed on one of the acceptable platforms. If  $g_i$  is not yet deployed on one of the acceptable platforms the deployer calls the `start()` method of the `Control` interface provided by the infrastructure component (see Section 6.2.5) in order to request the execution of  $g_i$  on  $p_k \in P$ . Having successfully deployed  $g_i$  on  $p_k$  the infrastructure component creates an edge labeled `DEPLOYED_ON` from the tactile deployment node to the platform node (see Figure 7.4b).

At time  $t_2$  the robot reaches the living room and hands-over the mug to the user. The robot opens the fingers of the gripper to release the mug. Thus, the context monitor detects the `release` action. Subsequently, the selector chooses an appropriate slip detector for the observed context, namely the force slip detector. The selector requests the deployer to stop the current slip detector and to deploy the force slip detector. Depending on the implementation of the perception graphs it would be also possible to simply send a pause signal to the slip detector.

As shown in Figure 7.4c once the tactile slip detector is stopped, the edge from the deployment to the platform node is updated, namely the label is changed from `DEPLOYED_ON` to

DEPLOYMENT\_HISTORY. Like at time  $t_1$  deployment requirements are checked and the force slip detector is deployed on the platform where the force sensor is connected to.

---

**Algorithm 2** Checking whether or not deployments are valid.

---

```

1: function checkDeployment( $p_i$ )
2:    $D \leftarrow \text{Query.getActiveDeployments}(p_i)$    ▷  $D$  is the set of active deployments on  $p_i$ .
3:   for each  $d_i \in D$  do
4:      $g_i \leftarrow \text{Query.getPerceptionGraph}(d_i)$  ▷  $g_i$  is the corresponding perception graph.
5:      $C \leftarrow \text{Query.getConstraints}(d_i)$        ▷  $C$  is the set of deployment constraints.
6:      $P \leftarrow \text{checkValidity}(C)$                ▷  $P$  is the set of acceptable platforms.
7:     if  $P = \emptyset$  then
8:        $\text{Error.noAcceptablePlatforms}(g_i)$ 
9:       if  $\text{Control.stop}(g_i, p_i)$  then
10:        return success
11:      else
12:         $\text{Error.stoppingFailed}(g_i, p_i)$ 
13:        return error
14:     if  $p_i \in P$  then
15:       return success
16:     if not  $P = \emptyset$  and  $p_i \notin P$  then
17:       if  $\text{Control.start}(g_i, p_k \in P)$  then
18:        return success
19:       else
20:         $\text{Error.deploymentFailed}(g_i, p_k \in P)$ 
21:        return error

```

---

At time  $t_3$  the force sensor breaks and no force signal is provided anymore. The context monitor detects this failure and updates the corresponding platform model, namely the edge from the platform node to the sensor/device node is removed (see Figure 7.4d). The repository notifies the deployer about those changes. Subsequently, the deployer executes the `checkDeployment()` method shown in Algorithm 2. The main objective of Algorithm 2 is to ensure that deployments remain valid in the presence of platform changes. To this end, each active deployment on the updated platform  $p_i$  is checked whether or not the requirements are met (cf. Algorithm 1). Three situations can occur, namely *a*) no platform meets the requirements, *b*)  $p_i$  meets the requirements, or *c*) other platforms than  $p_i$  meet the requirements. In the context of the case study no platform satisfies the requirements, thus, the force slip detector is stopped.

### 6.3.5. Deployment Infrastructure

In the context of the case study the deployment infrastructure component is build up on the `systemd` [179] software suite for system and service management on Linux operating systems.

In summary, `systemd` is an init daemon process responsible for launching services, setting up logging facilities, mounting file systems, and so forth. Nowadays `systemd` is the de facto

standard init system on major Linux distributions and replaces other init suites like `sysvinit`. Thus, implementing a deployment infrastructure based on `systemd` is feasible not only because it can be used on many Linux distributions, but also as it provides interesting features like on-demand starting of daemons and services, utilities to manage services, and so forth.

An important concept in `systemd` is that of a unit described by a configuration file. A unit is managed by `systemd` and represents, for example, a service, mount point, device, and so forth.

In the context of this work perception graphs are mapped to services. Here, a service represents a process which is controlled and supervised by `systemd` [180]. In this work DepSL is expanded to enable domain experts to express `systemd` related execution information. For example, the name and location of the executable, exception and failure policies, dependencies to other services, and so forth. Having such information is crucial to create `systemd` service files which are used by the `systemctl` [181] command to start and stop services. Here, both the `start()` and `stop()` methods of the provided `Control` interface encapsulate the `systemctl` command provided by `systemd`. Using `systemd` as deployment infrastructure provides also a convenient way to retrieve meta-information about services. For example, information like the process ID, process status, activation time, memory consumption, and so forth.

In the context of the case study some information, for example, the process ID and the timestamp when a service has been started are written by the infrastructure component in the repository (see Figure 7.4d).

## 6.4. Related Work and Discussion

Software deployment in robotics is usually achieved by some kind of deployment infrastructure provided by the underlying robot software framework. For example, the `roslaunch` deployment tool of the popular ROS [117] framework takes a XML-based description of the ROS architecture as an input and initiates the deployment according to it. To this end, components in ROS also known as nodes are started, stopped, parameters are set and so forth.

Another notable deployment approach is proposed by Ando *et al.* [182]. Here – in the context of the OpenRTM robot software framework [183] – deployment is considered as a part of component and system lifecycle management. The approach mainly deals with implementation-level details, for example, how manager services interact and how components are instantiated. Like in ROS, the OpenRTM deployment infrastructure relies on dedicated deployment files expressing crucial deployment information such as the location of an executable and so forth. Although these approaches help to automate the deployment task they are limited as they are not capable of expressing and resolving deployment requirements as presented in this chapter. Nevertheless, those approaches can be integrated in the reference architecture as an implementation of the deployment infrastructure component (see Section 6.2.5).

Another robotics software deployment approach is proposed in [184]. Here, Reiser introduced a web-based tool for configuring and deploying robotics software on service robots. Similarly – as the approach proposed in this chapter – platform requirements are specified and automatically resolved. However, his work is limited to static environments whereas the approach presented in this work allows to deploy perception graphs at run time in presence of varying resource conditions.

This chapter demonstrated that software deployment decisions should be separated as much as possible from the core development of software functionalities. As proposed by Mikic and Medvidovic [185] this is achieved by leveraging domain-specific languages to express architectural aspects and by explicitly declaring deployment constraints. This will make the developed software more independent of a particular hardware architecture – and thus more reusable – and allow it to be deployed more flexibly on a wider variety of robot platforms as long as the requirements are fulfilled.

Clearly, the architecture proposed in Section 6.2 is inspired by the MAPE-K [163] reference architecture for self-adaptive software systems as it contains similar building blocks as those proposed in MAPE-K like monitoring, knowledge storage, analysis and so forth. However, the introduced deployment architecture is more fine-grained as, for example, a stepwise deployment is supported. The selector (see Section 6.2.3) deals with what should be deployed and the deployer (see Section 6.2.4) deals with how and where it should be deployed.

All deployment requirements are treated equally by Algorithm 1 as no preferences, weights or the like are given. Thus, if a platform is not meeting all the provided requirements it is not in the set of acceptable platforms  $P$  (cf. Algorithm 1). In addition, the current implementation ensures that the deployment requirements are not modified at run time. However, supporting dynamic, modifiable requirements could be feasible in cloud-robotic scenarios [186] [187] where resource are requested on demand. It remains to be investigated which modifications are required to support dynamic deployment requirements, but in principle Algorithm 1 and also the repository (see Section 6.2.1) could be modified to realize those scenarios.

## 6.5. Summary

This chapter introduced an approach for deploying robot perception systems in the presence of varying resource conditions. Both RPSL and DepSL were utilized and expanded to express the knowledge relevant for deployment. Thus, domain experts can reuse models and can easily make changes in the deployment setting. In order to carry out deployment, the proposed reference architecture enables developers to implement deployment mechanisms which are capable to adapt the deployment on varying resource conditions, thus meeting the deployment requirements.

## Chapter 7.

# Adapting Robot Perception Systems

*“As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications.”*

— David Lorge Parnas, 1941 —

### 7.1. Introduction

This chapter demonstrates how the reference architecture proposed in Chapter 6 can be used to implement and deploy robot perception systems which are capable to cope with varying context-conditions. To this end, three different applications are developed, each of them dealing with variations in different context categories commonly appearing in robotics [176], namely variations in the task (see Section 7.2), platform (see Section 7.3) and environment (see Section 7.4) context.

Each application uses the stable, application-independent deployer component (see Section 6.2.4) for deploying perception graphs on suitable platforms as exemplified in Chapter 6. However, as the deployer solely deals with the question which platform should execute some perception graph, but not which perception graph itself should be deployed, an additional component is required. Thus, each application proposes a selector component (see Section 6.2.3) which decides which perception graph is suitable for the current context condition. Such a decision eventually leads to an adaptation of the robot perception system, for example, switching from one perception graph to another.

In order to come up with those adaptation decisions the reference architecture foresees two additional components, namely the repository and one or more context monitors. The former contains the design time and run time knowledge required for carrying out the adaptation (see Section 6.2.1) and the latter preprocesses the contextual knowledge (see Section 6.2.2). Each application described in the following sections realize application-specific repository and context monitors. Thus, they demonstrate the applicability of the reference architecture.

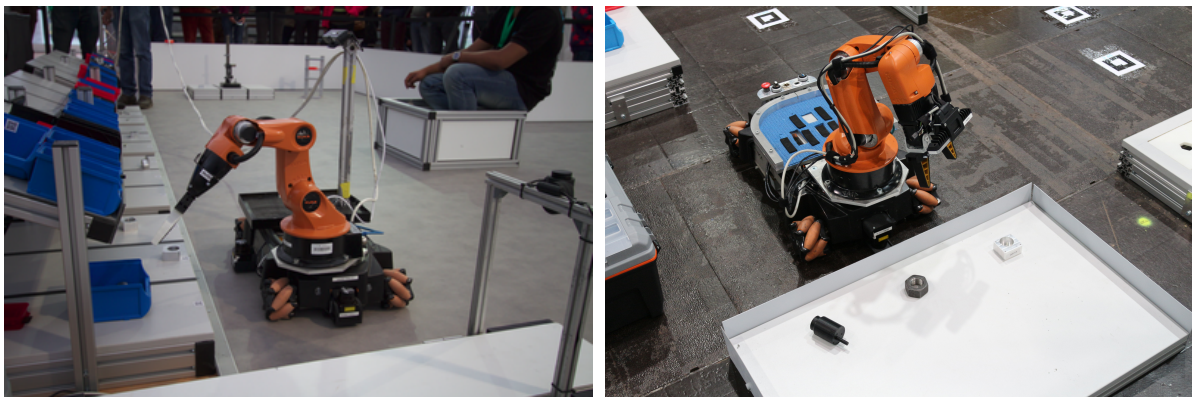
## 7.2. Task-based Adaptation

### 7.2.1. Motivation

Nowadays the robotics and automation industry is shifting its attention towards robotic scenarios involving the integration of mobility and manipulation. To this end, mobile robots and industrial robot arms are integrated on mobile manipulators. Examples of mobile manipulators include the KUKA youBot [10], KUKA KMR iiwa [188] and the rob@work platform [189] to name a few. Those mobile manipulators are expected to perform not a single, but preferably a wide range of complex tasks like assembly, automation and parts handling.

Although industry views mobile manipulators as an essential component for the Factory of the Future [190], real applications are still rare. It became quickly obvious that the control concepts and algorithms developed independently for robotic arms and for mobile robots could not easily be combined and integrated [6]. Thus, more research is necessary to exploit the capabilities of mobile manipulators in innovative applications.

One way to raise interest in solving particular problems and to create more research and development activities are scientific competitions as those developed, for example, in the context of RoboCup [191]. Examples of well-known competitions can be found in the area of soccer playing robots [192], domestic robots [2] and search and rescue robots [193].



**Figure 7.1.** Robots deployed in a factory-like RoboCup@Work environment with different service areas.

In the context of the Factory of the Future, competitions such as RoboCup@Work [6] and RoCKIn@Work [194] have been recently established. Those competitions are targeted towards innovative applications where mobile manipulators are used for industrial, work-related tasks ranging from loading and unloading of containers with industrial objects like screws, nuts and bolts, and operation of machines, for example, drilling machines and conveyors to cooperative assembly of non-trivial objects with other robots and human workers.



In the context of this section a task from the RoboCup@Work competition has been realized, namely the Basic Transportation Test (BTT) [195]. The BTT is a well-defined benchmark within RoboCup@Work and requires the combination of navigation and manipulation abilities to perform transportation tasks. The objective of the task is to get several objects from one or more source service areas and to deliver them to one or more destination service areas (see Figure 7.1).

In RoboCup@Work a service area denotes a region in a factory-like environment for a particular purpose, for example, areas to load, unload, place and/or insert objects. Service areas may contain specific objects such as cavities, conveyor belts, racks, storage areas and shelves. In addition, service areas can have different heights.

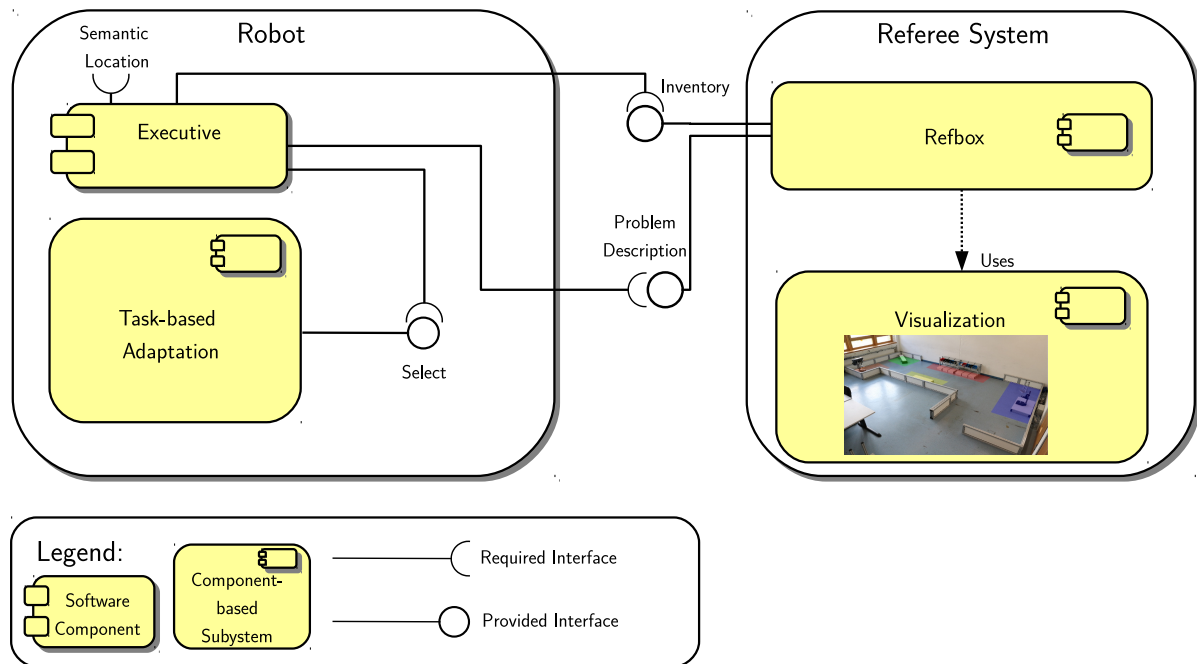
It is important to emphasize that the robots competing in RoboCup@Work initially know only which service areas are available in a factory. The concrete problem description is given by the referee system of RoboCup@Work [196] at competition time. In the context of the BTT benchmark, the problem description encodes the goal state of objects and their corresponding locations (service areas). This is somehow similar to domains and problem descriptions, for example, in PDDL [78] where the former contains domain predicates and the latter contains the initial state description and the actual goal.

In order to successfully execute the BTT a robot requires not only navigation, manipulation and control abilities, but also a broad set of perception abilities to segment, detect, recognize and possibly track service areas and objects. As the concrete problem description is not known a priori the robot is required not only to select, but also to specify at run time those perception features which are suitable for the task at hand. Depending on the concrete task expected to be performed at a specific service area those specifications differ significantly. For example, to place an object on a shelf different information about free-space, potential obstacles, and so forth are required whereas for inserting an object into a cavity other information such as dimensions of the cavity are required.

Thus, the overall goal of this section is to enable robots' to select perception features by specifying not only what kind of information they need to carry out their tasks, but possibly also how this information is provided. For example, object recognition features often differ not necessarily in their output, but how their output is delivered such as some implementations are capable of recognizing objects very fast, but not very precise whereas other implementations are slow, yet precise.

In order to realize such a task-based adaptation of perception features – as motivated in the aforementioned paragraphs – the reference architecture proposed in Section 6.2 has been employed (see Task-based Adaptation subsystem in Figure 7.2). More precisely, RPSL (see Chapter 3) is used to model a set of perception features and corresponding perception graphs which are required for the BTT benchmark. The resulting RPSL domain models are stored in the repository (see Section 7.2.3). The selector component (see Section 7.2.4) of the Task-based

Adaptation subsystem provides a `Select` interface and implements a selection algorithm. This algorithm selects a suitable perception graph based on a specification received via the `Select` interface and the perception features and graphs stored in the repository. In the following sections each component of the Task-based Adaptation subsystem is described.



**Figure 7.2.** Overview of the components involved in implementing the task-based adaptation scenario. On the robot – a KUKA youBot – the `Executive` component receives both the factory inventory and the problem description from the so-called `Refbox` [196]. The `Refbox` assists a human referee to carry out the competition by generating problem descriptions, visualizing the status of the competition, and so forth. The inventory expresses the current state of the factory whereas the problem description expresses the desired state of the factory. Both inventory and problem description are encoded as Google Protocol Buffer messages. Note, more details about the corresponding message types are given in [196] and [197]. Having the initial state (cf. inventory) and goal state (cf. problem description) the `Executive` component plans and executes a sequence of actions in order to achieve the task [113]. In order to carry out the perception actions the `Executive` employs the `Select` interface provided by the selector (see Section 6.2.3) to trigger execution of those perception features suitable for the task at hand. Note, the selector is part of the task-based adaptation subsystem following the reference architecture proposed in Section 6.2. In order to select suitable perception features for the task at hand the `Executive` also requires some contextual information, namely the semantic location. That is some high-level information about the current location of the robot, for example, in front of service area X. In this work it is assumed that such information is accessible by the `Executive` component, for example, through some world model or annotated topological map.

### 7.2.2. Context Monitoring

In order to carry out the task and to select which perception features are required for the task at hand, the `Executive` (see Figure 7.2) needs to access contextual information. The inventory and problem description are received from the referee system (see Figure 7.2) whereas the semantic location is provided by a context monitoring module. This context monitor is configured with an annotated grid map where certain regions of the map are labeled, for example, service area X. The monitor then checks to which region the current position of the robot belongs to and provides this information to the `Executive`.

```
1 rpsl.feature_resolution do
2   resolve "ServiceArea",
3   by "ServiceArea1",
4   :exposed_ports => ["Plane"],
5   :exposed_properties => ["DetectPlaneHeight"]
6 end
```

**Figure 7.3.** Resolution model for the perception feature `ServiceArea`. The feature is resolved by a perception graph `ServiceArea1`. Note, the specification of the graph itself is shown in Figure 4.6. The resolution exposes planes as an output and a property `DetectPlaneHeight` expressing the height of detectable planes.

### 7.2.3. Repository

The RPSL domain model expressing the functional variability for the case study discussed in Section 4.4 has been employed also in this work. The domain model includes five leaf features representing high-level perception abilities for detecting objects and service areas plus features for recognizing objects, containers and cavities. The containment and specializations relations among those features is discussed in Section 4.4.1.

In order to express the resolutions, RPSL is employed as exemplified in Figure 7.3. Those RPSL domain models encode not only which feature is resolved by which perception graph, but also which output ports and properties of the graph are exposed to facilitate the selection of perception graphs (see Section 7.2.4). Note, as discussed in Chapter 3 perception features and perception graphs are completely orthogonal models, thus the resolution is required to weave those models together.

The resolutions of the features is described in the following paragraphs and summarized in Table 7.1.

**ServiceArea** is resolved by three different perception graphs. All of them are capable of detecting service areas/dominant planes on different heights. The exposed property

Feature	Perception Graph	Exposed Output	Exposed Properties
ServiceArea	ServiceArea1	Plane	DetectPlaneHeight
	ServiceArea2	Plane	DetectPlaneHeight
	ServiceArea3	Plane	DetectPlaneHeight
ObDetection	ObjDetection	BBox NumberOfObj	–
ObRecognition	FastObjRecognition	Pose	AvgRecogTime RecognizableObj
	SlowObjRecognition	Pose	AvgRecogTime RecognizableObj
CavRecognition	CavityRecognition1	Pose	–
	CavityRecognition2	Pose Plane	– –
ContRecognition	RedContainer	Pose BBox	RecognizableCont
	BlueContainer	Pose BBox	RecognizableCont

**Table 7.1.** The table shows the perception graphs resolving the perception features which are used in the application. Perception graphs can expose some of their output ports and some of their properties.

DetectPlaneHeight expresses the height of detectable planes by the corresponding perception graph whereas Plane is an exposed output and encodes the detected plane itself.

**ObDetection** is resolved by exactly one perception graph exposing a bounding box BBox. As described in [113] the ObDetection graph utilizes different components for clustering and for fitting bounding boxes around clusters situated on planes.

**ObRecognition** is resolved by two perception graphs. Both graphs provide poses of recognized objects (see exposed output Pose in Table 7.1). They differ in the average recognition time (cf. AvgRecogTime) and in the set of recognizable objects (cf. RecognizableObj). The resolved graphs for the ObRecognition feature clearly demonstrate the difference between an exposed output and an exposed property. The former is used for declaring what the perception graph is capable of providing and the latter is often used to characterize this output.

**CavRecognition** is resolved by two perception graphs. Both expose the poses of recognized cavities and the graph CavityRecognition2 provides also the plane of the cavity area.

**ContRecognition** is resolved by two perception graphs. Both graphs expose two outputs, namely the pose of the recognized container and the bounding box of the container. Further, the exposed property `RecognizableCont` expresses the set of recognizable containers. As the name suggests the `RedContainer` perception graph recognizes red containers whereas the `BlueContainer` perception graph recognizes blue container.

#### 7.2.4. Perception Graph Selection

The selector component is in charge of identifying those perception graphs which are fulfilling the specification. In this work the specification is given by the `Executive` module via the `Select` interface (see Figure 7.2).

A specification  $S$  is formally defined as a quadruple

$$S = (f, O, P, \Gamma) \quad (7.1)$$

where  $f \in F$  encodes the feature which the `Executive` module would like to select from the set of available features  $F$ ,  $O$  is the set of output ports a perception graph resolving  $f$  should expose. Similarly,  $P$  is the set of properties a perception graph resolving  $f$  should expose and  $\Gamma$  is the set of prototypes. Here, a prototype  $\gamma_i \in \Gamma$  encodes an expected value for a certain property. Depending on how a property is defined (see Section 3.6.5) a prototype could have one or more symbolic and/or numeric values. The general idea of providing a specification  $S$  is that the `Executive` – or any other component using the `Select` interface – is solely aware of which high-level perception features are available, but not how they are resolved in terms of different perception graphs with varying characteristics.

In order to select a perception graph fulfilling the specification the selector implements Algorithm 3 which is explained in the following paragraphs.

In case the specified feature  $f$  exists, all perception graphs resolving the feature are assessed. For each perception graph two checks are performed. Firstly, it is checked whether the set of output ports  $O$  is exposed by the graph. Secondly, it is checked whether the set of expected properties  $P$  is exposed by the graph. In case both checks are positive, the similarity between the specified prototypes  $\Gamma$  and the one attached to the perception graph is computed.

To do so, the method `computeSimilarity()` implements a similarity measure, for example Euclidian or Hamming distance, suitable for the task at hand. The obtained similarity value  $\Delta \in \mathbb{R}$  is subsequently stored together with the assessed perception graph as a candidate. In case only the former check is positive the perception graph is stored as a candidate with an empty similarity value. In case the set of candidates  $C$  is not empty the application-specific method `bestMatchingGraph()` retrieves a best matching graph. Note, in the simplest case the `bestMatchingGraph()` method randomly selects a graph, but more advanced computations

---

**Algorithm 3** Selecting perception graphs meeting the specification.

---

```

1: function Select.perceptionFeature( $S$ )           ▷  $S$  is the feature select specification.
2:   if Query.existFeature( $f$ ) then
3:      $G \leftarrow$  Query.getResolvedPerceptionGraphs( $f$ )
4:      $C \leftarrow \emptyset$                        ▷  $C$  is the set of candidate perception graphs.
5:     if not  $G = \emptyset$  then
6:       for each  $g_i \in G$  do
7:          $EO \leftarrow$  Query.getExposedOutputPorts( $g_i$ )
8:          $EP \leftarrow$  Query.getExposedProperties( $g_i$ )
9:         if  $O \subseteq EO$  then
10:          if  $P \subseteq EP$  then
11:             $\Delta \leftarrow$  computeSimilarity( $P, EP, \Gamma$ )
12:             $C \leftarrow C \cup \{\langle g_i, \Delta \rangle\}$ 
13:          else
14:             $C \leftarrow C \cup \{\langle g_i, \emptyset \rangle\}$ 
15:        if not  $C = \emptyset$  then
16:           $g_k \leftarrow$  bestMatchingGraph( $C$ )
17:          Deploy.perceptionGraphs( $g_k$ )
18:          return success
19:        else
20:          Error.noPerceptionGraphForGivenSpecification( $S$ )
21:          return error
22:      else
23:        Error.noResolvedPerceptionGraphs( $f$ )
24:        return error
25:    else
26:      Error.featureNotAvailable( $f$ )
27:    return error

```

---

taking, for example, application-specific preferences into account are imaginable. After receiving the best matching graph  $g_k$ , the graph gets deployed. To do so, the Deploy interface provided by the deployer component (see Section 6.2.4) is employed.

In three situations Algorithm 3 returns an error, namely *a*) if the feature  $f$  is not available, *b*) if no perception graph resolves feature  $f$ , or *c*) if no perception graph fulfills the provided specification  $S$ . For the latter situation the Executive module could – if possible – refine the specification  $S$ . To do so, Algorithm 3 could provide some hints why the specification is not feasible, for example, because a specific output is not exposed by any assessed graph. However, in the current version those hints are not yet implemented.

### 7.2.5. Perception Graph Deployment Infrastructure

In this scenario the ROS [117] robot software framework is used to implement the perception graphs. Here, each perception graph (see Table 7.1) is an executable expressed as a ROS node.

In order to start and stop selected perception graphs and to configure configuration parameters, the `roslaunch` [178] API is used which provides methods to start, stop and configure ROS nodes.

### 7.2.6. Experiments

In order to study the feasibility of the task-based adaptation the BTT benchmark has been realized. To this end, the BTT has been performed in an environment composed of three service areas. Each service area is described in the following paragraphs.

- Service area  $S_1$  is situated on the ground of the factory where objects can be picked. The area itself is bordered with black tape.
- Service area  $S_2$  has a height of  $10\text{cm}$  where objects can be placed and inserted. To this end, the plane of the service area includes object-specific cavities where objects can be inserted.
- Service area  $S_3$  has a height of  $15\text{cm}$  where objects can be placed. To this end, objects can be placed in two different containers, namely a red and a blue container which are standing on the service area.

Within the marked region of  $S_1$  four industrial objects are randomly placed, namely a screw, black aluminium profile, a gray aluminium profile and a nut. The task – as expressed in the problem description – of the robot is to pick the gray profile and to insert the profile into the corresponding cavity at  $S_2$ . In addition, all the remaining objects should be placed in the red container located at  $S_2$ .




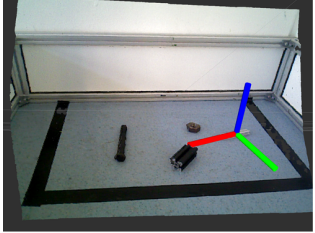
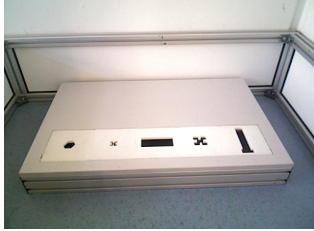
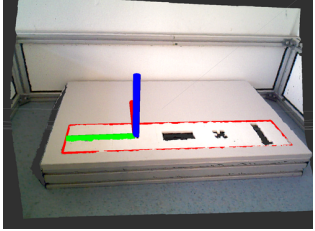

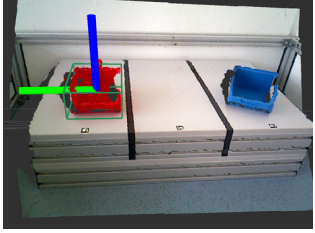
Having contextual information about the semantic location, the inventory and the problem description the `Executive` is – among other issues – in charge of specifying the perception features which are required for the task at hand. In order to provide such a specification in a computer-readable manner, RPSL has been extended through a `select` statement as shown in Table 7.2. Basically the statement is a Ruby-based representation of the specification  $S$ .

The table reports about some snapshots of selection statements recorded during the execution of the exemplified scenario. Here, each row represents a point in time where the robot perceived a scene (cf. second column), specified one or more features (cf. first column) which leads to a selection and execution of perception graphs which leads to some output (cf. third column). It is important to note that before and after each snapshot other features are possibly selected.

In the following paragraphs each selection statement shown in Table 7.2 is discussed in more detail.

- Firstly, the robot is placed in front of  $S_1$ . In order to compute the number of objects located at  $S_1$  the robot specifies the `ObDetection` feature which exposes the numbers of detected objects in the scene. As shown in Table 7.1 the `ObDetection` feature is resolved by exactly

one perception graph also exposing bounding boxes. Thus, this graph is selected and deployed as the specification is fulfilled. The bounding boxes computed by the graph are shown in the Output column of the first row in Table 7.2.

Specification	Input	Output
<pre>select :ObDetection do :with_output =&gt; "NumberOfObj" end</pre>		
<pre>select :ObRecognition do :with_property =&gt; "RecognizableObj", :match =&gt; "F20_20_G", :similarity =&gt; :JARO_WINKLER end</pre>		
<pre>select :CavRecognition do :with_output =&gt; ["Pose", "Plane"] end</pre>		
<pre>select :ContRecognition do :with_property =&gt; "RecognizableCont" :match =&gt; "RedContainer" :similarity =&gt; :EUCLIDIAN end</pre>		

**Table 7.2.** The table shows recorded snapshots of specifications, perceived scene and corresponding output.

- Secondly, the robot is expected to pick the gray aluminium profile. In the context of RoboCup@Work this object is called F20\_20\_G. To do so, the Executive module specifies the ObRecognition feature by providing a prototype F20\_20\_G for the RecognizableObj property. Furthermore, a similarity measure is expressed, namely the Jaro-Winkler distance for measuring the edit distance between two strings. This measure is used as the RecognizableObj property is simply a list of strings where each string encodes the object which is recognizable by the perception graph. In the context of this application both perception graphs resolving the feature ObRecognition are capable of recognizing F20\_20\_G. Thus, the bestMatchingGraph() method in Algorithm 3 needs to chose between two



possible candidates. In this work a graph is randomly selected. The object pose computed by the graph is shown in the Output column of the second row in Table 7.2.

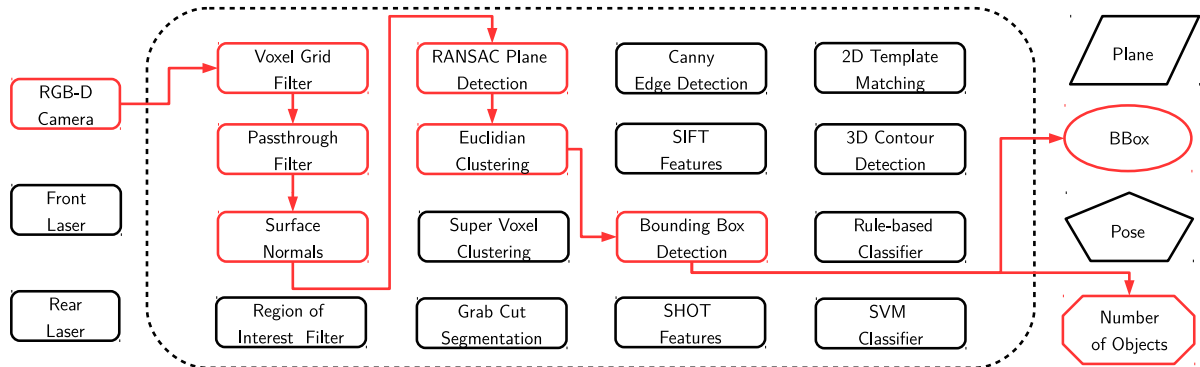
- Thirdly, the aluminium profile needs to be inserted in the corresponding cavity. To this end, the `CavRecognition` is specified in a similar manner as in the first step, namely solely via the required output. Here, the Pose information about the cavities is required for object insertion. As two graphs are fulfilling the specification the `bestMatchingGraph()` method chooses in this situation the `CavityRecognition2` perception graph which also provides a plane as shown in the Output column of the third row in Table 7.2.
- Lastly, to place all objects in the red container the pose of the red container is required. Therefore, the `ContRecognition` feature is specified. Similarly as in the second specification two perception graphs are resolving the feature, thus a richer specification is helpful. As shown in Table 7.1 both perception graphs resolving the feature expose a property called `RecognizableCont`. This property encodes the recognizable container not in terms of a simple string, but as an average RGB pixel value. Thus, the Euclidian distance is used as a similarity measure to compute the distance between the `RedContainer` prototype provided in the specification and the prototypes attached to the `RecognizableCont` property. The bounding box and pose of the red container computed by the `RedContainer` perception graph is shown in the Output column of the fourth row in Table 7.2.

The experiments demonstrated how RPSL domain models can be exploited at run time to support the task-driven selection of robot perception features. This task-driven selection is enabled by having a simple, yet powerful means to specify – in a declarative manner – high-level perception features without knowing precisely which graphs are actually realizing this feature. Thus, the approach conforms to the information hiding principle, where the specification `S` serves as a stable interface whereas the perception graphs (aka. the implementations) are likely to change.

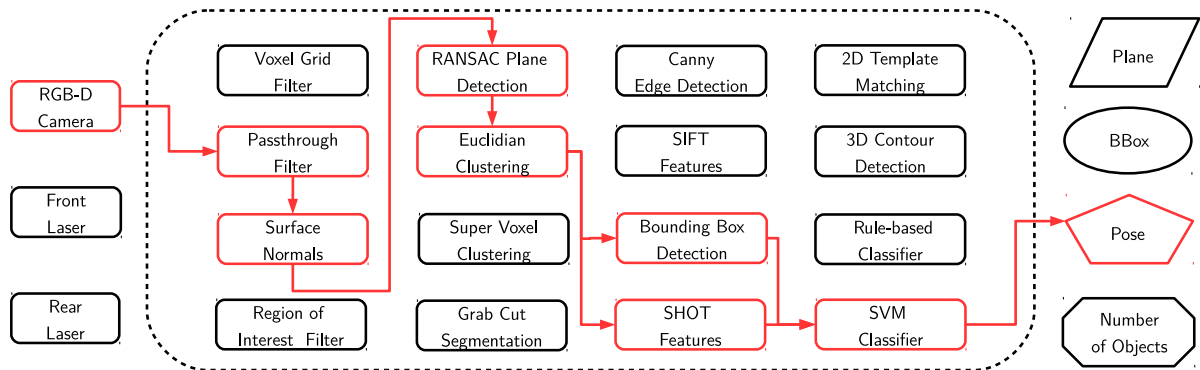
It is important to emphasize that each selection statement shown in Table 7.2 induces the deployment of a specific perception graph. Those graphs are visualized in Figure 7.4 where the graph selected at a specific point in time is visualized in red. On the left-hand side the sensor components available in this scenario are shown. Those sensor components are connected with a set of varying processing components required for the task at hand. The right-hand side depicts the output which the different graphs can generate and corresponds to the third column in Table 7.2.

### 7.2.7. Related Work and Discussion

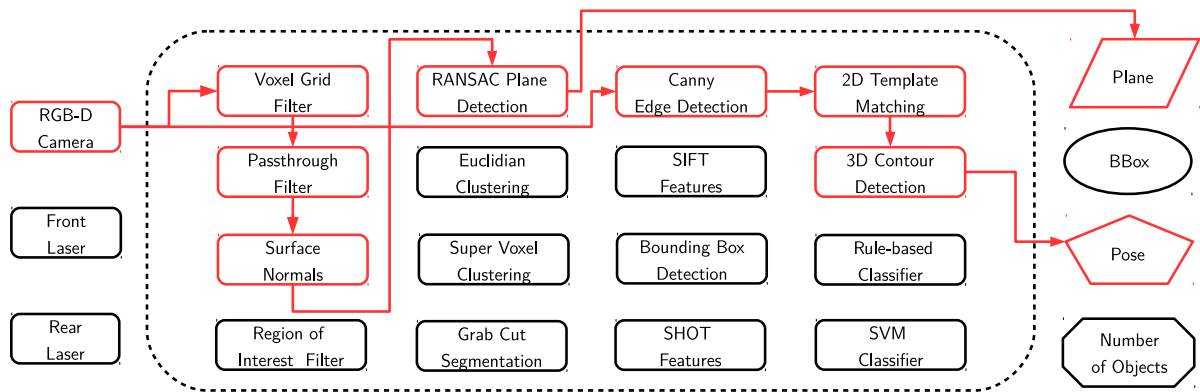
The presented approach resembles some methods and concepts found, for example, in artificial intelligence. From an AI perspective the task-based adaptation approach is inspired by case-based reasoning (CBR) methods [198] as both the repository and the selection algorithm have



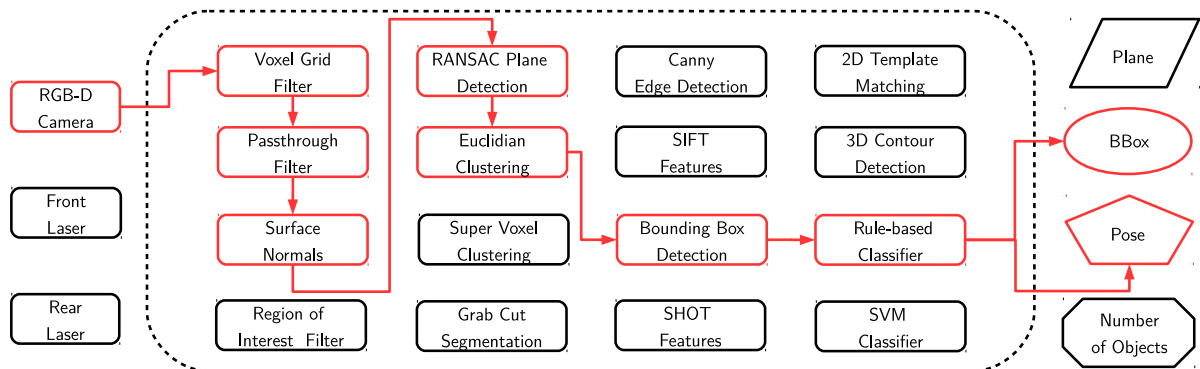
(a) The ObjDetection graph selected at time  $t_0$ .



(b) The FastObjRecognition graph selected at time  $t_1$ .



(c) The CavityRecognition2 graph selected at time  $t_2$ .



(d) The RedContainer graph selected at time  $t_3$ .

Figure 7.4. The different perception graphs deployed at different points in time.

CBR counterparts, namely case base and case retrieval mechanism. In fact, CBR approaches have been applied in robotics to select behaviors in robotic soccer [199] and to select navigation strategies [200]. The general idea is always the same, namely to reuse existing knowledge to cope with recurrently and sometimes unexpected context conditions. In this work the knowledge is expressed as RPSL and DepSL domain models.

In [201] Beetz *et al.* introduced the RoboSherlock system to adapt robot perception systems. Here, robot perception is modeled as an Unstructured Information Management problem where different perception algorithms are simultaneously employed to answer task-related questions about a scene. Those questions are on a much higher level of abstraction than the specifications expressed in this section. For example, whether a spoon is available on a table to perform some cooking task. Nevertheless the work presented in this section is complementary to RoboSherlock as the focus is on leveraging implementation-level knowledge (the *How?*) whereas RoboSherlock focus is on the *What?*.

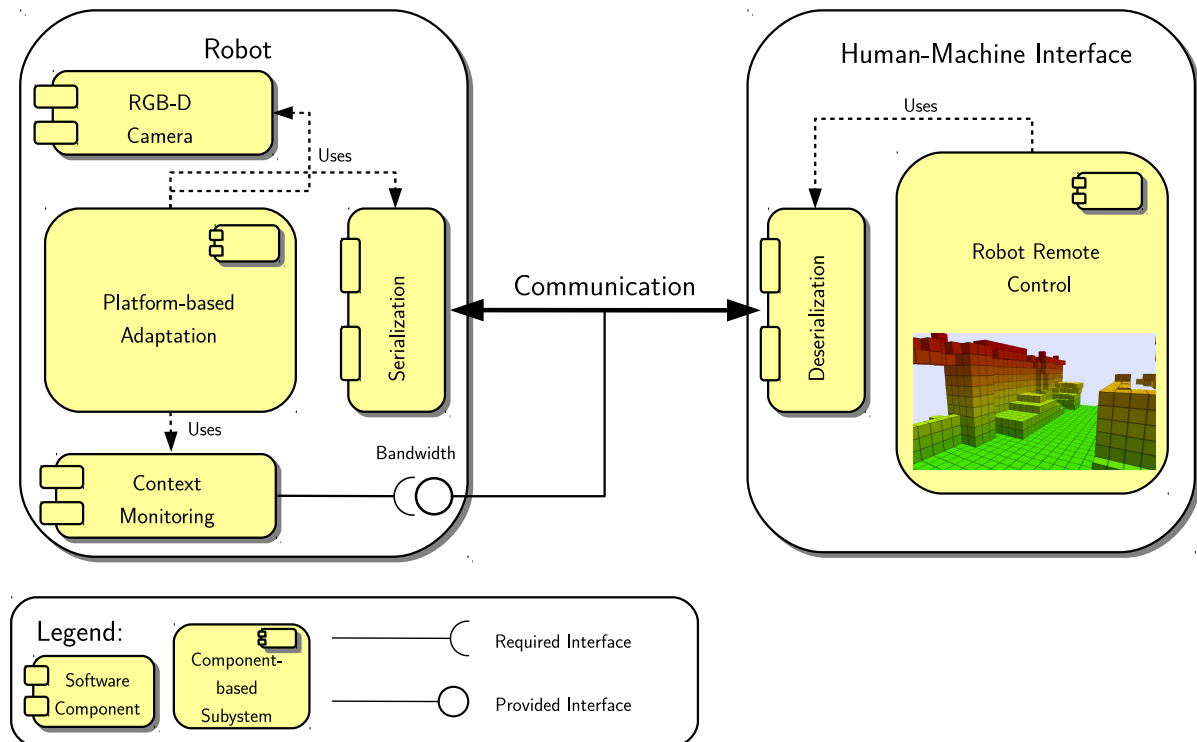
## 7.3. Platform-based Adaptation

### 7.3.1. Motivation

With the advent of agile and versatile robot platforms, operating in the air, underwater and on the ground the development of Search and Rescue (SAR) missions [202] in unstructured and harsh environments such as alpine [203], maritime [204] and disaster [8] settings is becoming a reality. In those applications mixed teams of autonomous and semi-autonomous heterogeneous agents like humans, robots and distributed sensors need to collaborate to achieve their tasks.

In order to enable collaboration among those agents a world model needs to support physically distributed data storage and shared data access such that the situational awareness of individual team members and the team as a whole is improved. Here, a distributed world model not only creates, but also maintains a digital representation of the environment over a possibly long period of time based on the results of employed perception algorithms. This representation needs to be exchanged by replication and synchronization among all agents.

In order to implement collaboration via distributed world models – for example within SAR missions – a wireless networking and communication infrastructure is required. However, as Troubleyn *et al.* [205] point out, real-world environments pose significant Quality of Service (QoS) challenges on the networking and communication infrastructure. More precisely, QoS metrics such as the time it takes to transmit a message from source to destination (delay) and the number of messages which are lost during transmission (message loss rate) are not necessarily known a priori and change over time. Those time-varying QoS variations are caused by extreme environmental conditions such as heat and cold, agent mobility and the fact



**Figure 7.5.** Overview of the scenario realized in this section. Two physically distributed platforms are used, namely a mobile robot equipped with a RGB-D camera providing point cloud data and a Human-Machine-Interface (HMI) – deployed, for example, on a PDA or tablet computer – exposed to an operator. On the robot the reference architecture proposed in Chapter 6 is implemented (see Platform-based Adaptation subsystem). The Platform-based Adaptation subsystem selects and deploys perception graphs suitable for reducing the amount of data send to the HMI. To this end, the Context Monitor (see Section 7.3.2) retrieves the available bandwidth from the communication channel. Before the data is send it gets serialized and on the receiver side deserialized (see Serialization and Deserialization components). The Robot Remote Control subsystem visualizes the received data for the sake of operating the robot.

that applications are deployed in inaccessible, unknown environments, for example, disaster areas.

It is worth to note that the QoS properties are often in conflict with the application requirements. An operator supervising a SAR mission, for example, could formulate a maximum tolerated message delay in order to receive certain information which is crucial to proceed with the mission. In order to cope with these challenges in a systematic manner a representation of QoS properties and a mechanism to interpret them and eventually adapt the system behaviour is required.

This section focuses on reducing the amount of data to be transmitted. This is achieved by an appropriate adaptation of perception capabilities for data reduction at run time. To this end, the reference architecture proposed in Section 6.2 has been employed (see Figure 7.5).

More precisely, different data reduction algorithms are modeled with RPSL (see Chapter 3) and stored in the repository. The selector component (see Section 7.3.4) integrates a simple, yet powerful algorithm which enables to adapt to QoS changes immediately. This algorithm employs the concept of Level of Detail (LoD) as a criteria to select an algorithm. Here, bandwidth as contextual information is used as a QoS metric and an according mapping to the LoD is performed.

As illustrative example application a setup involving a robot equipped with a Kinect RGB-D camera is chosen. Depending on the concrete resolution and frame rate the RGB-D camera provides a possibly high-dimensional point cloud data. This data has to be exchanged with one or multiple Human Machine Interfaces to be used by the operators for the sake of remote operation (see Figure 7.5). The experiments (see Section 7.3.6) show that the adaptation satisfies an exemplary chosen application requirement of a max. transmission delay  $t_{max\_delay} = 1s$  even in the presence of changing QoS as commonly faced in SAR missions.

### 7.3.2. Context Monitoring

This section introduces the Level of Detail (LoD) as a generic metric to define a spatial resolution of point clouds independent of the actual data representation as no byte consumption is directly inferable. In the context of this work it is used *a)* to describe a spatial resolution of a given point cloud, and *b)* to describe the boundaries of the output of a data reduction algorithm. A LoD is defined as follows

$$LoD = \frac{N_{points}}{V}, \quad \text{in } \frac{sample}{m^3} \quad (7.2)$$

where  $N_{points}$  is the number of point samples per volume  $V$ . This formula is somewhat similar to printer specifications that use *dots per inch*. In this work the context monitor measures in the domain of bandwidth which is not directly connected to the representation used in the perception domain (LoD). The intention is to keep the domains for communication (e.g. bandwidth) and perception separated otherwise the LoD would not be reusable in other applications. Therefore, a formal mapping between both domains is defined. For the mapping between LoD and bandwidth  $f$  is defined as follows  $f : B \rightarrow LoD$  where  $B$  denotes the domain of bandwidth in bytes per seconds and  $LoD$  is defined as in Equation 7.2. In order to compute the LoD the context monitor is configured with the following parameters which can be derived for an application a priori:

- $t_{max\_delay}$  is the maximum delay in seconds tolerated by the application.
- $V_{max\_space}$  is the maximum volume covered by the sensor which can be deduced from the specification of the sensor.

- $bytes_{per\_point}$  is the relation between the number of points in a point cloud and its corresponding message size in bytes.
- $t_{offset}$  is the worst case total time required for encoding and decoding a message.

As the bandwidth  $b \in B$  is a variable that varies at run time, the maximum time required to send a message can be derived as  $t_{max} = t_{max\_delay} - t_{offset}$ . The number of maximum bytes that can be send depends on the available bandwidth:  $bytes_{max} = t_{max} * b$ . By knowing the memory consumption of a point in a point cloud, the maximum number of points to be sent in a message can be estimated by  $N_{points} = bytes_{max} / bytes_{per\_point}$ .

From the  $V_{max\_space}$  maximum covered volume and the maximum allowed points a maximum LoD can be deduced:  $LoD_{max} = N_{points} / V_{max\_space}$ . This leads to mapping  $f$ :

$$LoD_{max} = f(b) = \left( \frac{(t_{max} * b) / bytes_{per\_point}}{V_{max\_space}} \right) \quad (7.3)$$

This  $LoD_{max}$  denotes an upper bound for the LoD to satisfy the application tolerance  $t_{max\_delay}$ . Continuously the  $LoD_{max}$  value is updated in the repository by the context monitor. The  $V_{max\_space}$  and  $t_{offset}$  parameters introduced above are application-specific. In the context of this work they are defined as described in the following paragraphs.

In the experiments (see Section 7.3.6) a Kinect RGB-D sensor is used. Thus,  $V_{max\_space}$  can be conservatively approximated with  $V_{max\_space} = 64m^3$  for accounting the bounding box dimensions of  $4m * 4m * 4m$ . Note, these values can be enforced with an appropriate pre-filtering<sup>1</sup> step.

The data produced by the perception algorithms needs to be serialized in order to send it to the operators interface. Depending on the serialization approach the  $t_{offset}$  value might vary. In the context of this work HDF5 [207] is used as a serialization format. HDF5 is a file format for storing large scale scientific datasets and is the de facto standard in many domains. The fact that a framework independent serialization is chosen allows to make statements on the size of the transmitted data. In order to investigate the serialization costs artificial point cloud datasets have been fed to the system with an increasing number of points. The durations for  $t_{encode}$  for encoding and  $t_{decode}$  for decoding and the resulting message size bytes are listed in Table 7.3. Here, the HDF5 encoding and decoding reveals a predictable monotonic increasing characteristics. Therefore, the context monitor is configured with  $bytes_{per\_point} = 32$  as it converges to this value for large point clouds.

<sup>1</sup>For example a pass through filter available in the Point Cloud Library [206].

$N_{\text{points}}$	$t_{\text{encode}}$	$t_{\text{decode}}$	bytes	bytes_per_point
10	1.919ms	20.621ms	11584	$\approx 1158$
100	4.130ms	36.932ms	43616	$\approx 436$
1000	5.690ms	38.541ms	32517	$\approx 32$
10000	7.659ms	40.452ms	331616	$\approx 33$
100000	9.363ms	44.643ms	3211616	$\approx 32$

**Table 7.3.** Costs of encoding and decoding point clouds with HDF5.

### 7.3.3. Repository

Different data reduction algorithms have been modeled with RPSL (see Chapter 3) and stored in the repository. The RPSL perception graphs are composed of different Octree-based [208] sub-sampling filters with different leaf sizes. Octrees are commonly used as a strategy to reduce point cloud data [209]. In order to employ Octrees in the context of this work their relation to the LoD (see Section 7.3.2) needs to be defined.

The smaller the leaf size  $N_{\text{leaf}}$  the higher the possible resolution. A leaf size  $N_{\text{leaf}} = 1m$  means all points – if any – in a leaf with size  $1m * 1m * 1m$  are discarded and represented by the center of that cube. One point per such unit cube is exactly  $LoD_{\text{max}} = 1 \frac{\text{sample}}{m^3}$ . The according formula for other leaf sizes is:  $LoD_{\text{max}} = 1 / (N_{\text{leaf}})^3$ . Having this formula allows to compute and attach the  $LoD_{\text{max}}$  values for each Octree perception graph configuration as shown in Table 7.4. In addition, a perception graph called camera is also stored in the repository to account for situations where enough bandwidth is available. In this case the raw data is not changed at all.

Name	$N_{\text{leaf}}$ in [m]	$LoD_{\text{max}}$ in $[\frac{\text{sample}}{m^3}]$
octree1	1	1
octree0.5	0.5	8
octree0.25	0.25	64
octree0.2	0.2	125
octree0.1	0.1	1000
camera	-	10000

**Table 7.4.** Different perception graphs and their Octree configurations available in the repository.

### 7.3.4. Perception Graph Selection

Whenever the requested LoD values is updated by the context monitor the selector choses the most suitable perception graph for the current bandwidth context. The Euclidian distance as a simple, yet powerful comparison metric is employed. A perception graph is chosen based on

the smallest distance between the requested – as updated in the repository – and the stored LoD values (see Table 7.4).

### 7.3.5. Perception Graph Deployment Infrastructure

As in Section 6.3.4 each perception graph is expressed as a process having a systemd service description. Based on the selected perception graph the process is started respectively stopped.

### 7.3.6. Experiments

In order to investigate the proposed approach a set of experiments have been performed. The main intention is to assess whether the approach satisfies an application-based tolerance on the maximum delay in spite of a variable bandwidth of the communication layer.

To this end, two experimental objectives have been defined:

- **Objective 1:** The tolerated maximum delay – which is defined by the application – always holds even in the presence of changing QoS. Here, QoS is considered as the available bandwidth.
- **Objective 2:** The lower the available bandwidth, the lower the number of points that are transmitted.

Furthermore, the following hypothesis is investigated:

- **Hypothesis 1:** The density of the repository affects the delay significantly.

It is worth noting that it is assumed that the latency of the connection itself remains stable. Two experiments have been designed, both use bandwidth as *controlled variable*, the parameters from Section 7.3.2 and the repository as described in Section 7.3.3. As point cloud dataset for the camera an office scene is used.

1. **Artificial data:** The first experiment employs a full bandwidth spectrum with respect to existing communication technologies. To this end, the values start at zero, are incremented each after 10s, and stop at  $10^8$  bytes per seconds. By doing so a range is created which covers typical bandwidths for cell phone networks ( $10^5$  -  $10^6$ ), WiFi technology ( $10^6$  -  $10^7$ ) and Ethernet ( $10^7$  -  $10^8$ ).
2. **Real world data:** The second experiment employs real-world bandwidth measurements from a 3G cell phone network. Motivated by the fact that some SAR missions propose to use cell phone networks [204] as one part of their communication infrastructure a real-world dataset called *Car Snaroya Smestad* [210] is used to test the system under realistic bandwidth settings.



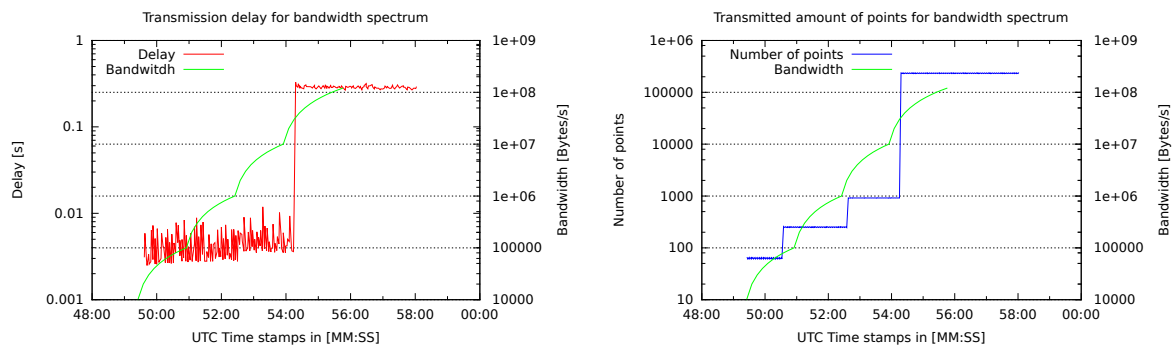


Figure 7.6. Measured transmission delay and transmitted number of points for the artificial dataset.

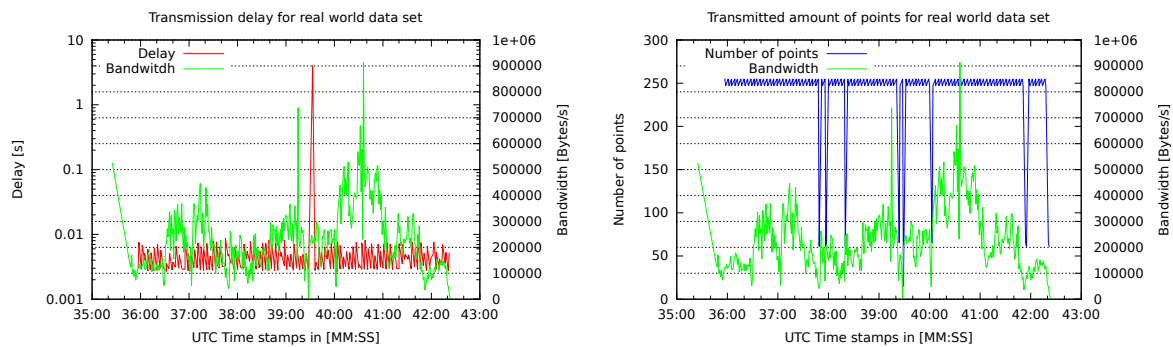


Figure 7.7. Measured transmission delay and transmitted number of points for the real-world dataset.

All experiments have been performed on a standard personal computer.<sup>2</sup> The perception graphs integrate the Octree-based subsampling methods provided by the BRICS 3D [211] C++ library. As communication framework ROS Hydro is used. Both the robot – having the Platform-based Adaptation (see Figure 7.5) subsystem – and the Human-Machine-Interface are embedded into a ROS node. In order to control the bandwidth an additional relay node is employed that introduces an artificial delay to the used topic based on a configurable bandwidth parameter. A bandwidth generator node sends simultaneously the bandwidth parameter according to the experimental design both to the relay node and the Context Monitor.

For the first experiment (see Figure 7.6) the delay tolerated by the application always holds with a maximum delay of  $\approx 0.3s$ . The adaptation decision leads to an increased or decreased number of points and is clearly visible in discrete steps. This verifies **Objective 2**. Here, each discrete step represents a different selection of a perception graph. The last step represents the transmission of the raw camera data as the bandwidth becomes sufficient. This is caused by the selection of the camera perception graph. The granularity of discrete steps is dense in the

<sup>2</sup>Intel Core i7 CPU and  $\approx 8GiB$  RAM and Ubuntu 12.04.

cell phone bandwidth range whereas in the WiFi range the selection is rather sparse. This can be explained with the selected Octree configurations (see Table 7.4) stored in the repository. Thus, an application developer needs to consider this effect at design time through a carefully population of the repository. By doing so the application developer can influence the overall performance of the adaptation. Here, the cell phone bandwidth range has been emphasized, thus **Hypothesis 1** seems to be correct.

For the second experiment (see Figure 7.7) which uses a real-world data set in the cell phone bandwidth range the delay tolerated by the application holds mostly with one exception. This outlier occurs on a sudden change to a very low bandwidth value (2736). In this case the system selected `octree1`, however the reduction was not enough. This input value is beyond the range of what the algorithms in the repository are capable of. As shown in Figure 7.7 mostly a rather similar data reduction has been applied. The selection toggles between `octree0.2` and `octree0.25` but as the leaf cell parameters are so similar to each other they produce nearly the same output. At the exceptional case that misses the application tolerance 15 points have been transmitted. 2 points would have still met the requirement. This shows a limitation of the approach, namely it only performs well if the repository has been carefully designed to cope with expected context variations. Also the extreme case for  $b = 0$  cannot be handled with this setup.

In summary, both experiments support **Objective 1**, namely the approach is able to handle QoS changes for this experimental setup.

### 7.3.7. Related Work and Discussion

In order to cope with the QoS challenges considered in this section some approaches propose to reconfigure directly the QoS properties of the communication infrastructure, for example, the work of Eich *et al.* [212] and Paikan *et al.* [213]. Although, such an approach is appealing when enough knowledge about the underlying middleware or communication infrastructure is available, but it remains technology-specific and difficult to maintain. Other approaches reduce the amount of data that is sent by applying a data reduction [214] or compression strategy [215]. However, these approaches are challenging to adapt at run time especially in the context of real-world applications where QoS changes are a given. Recent approaches for distributed world modeling such as the work of Tenorth *et al.* [216] and Dietrich *et al.* [217] do not tackle the requirements imposed by challenging SAR missions as mentioned in Section 7.3.1.

As shown in Section 7.3.6 the experiments solely exploit the LoD concept whereas further concepts, for example, the existence or absence of color information in a point cloud could be incorporated. This would yield in an extension of the search space for an appropriate perception graph. Such extensions would also yield in a heterogenous repository with different

perception graphs. Investigating suitable means to assess those repositories are promising directions for future work.

Interestingly, the results presented in this section encouraged Riestock *et al.* [218] to study whether or not it is possible to teleoperate a quadrotor solely with a gridmap-based interface in environments with bandwidth limitations. The idea is that such an interface would visualize the data produced by the employed Octree perception graphs as proposed in this section. With the help of a user study Riestock *et al.* [218] showed, for example, that a gridmap-based interface is sufficient for maintaining a safety distance to nearby obstacles. Nevertheless, their work also showed that an adaptive approach – as proposed in this section – is crucial to cope with bandwidth variations as faced in real-world environments. Thus, the work presented in this section can be used as one key building block for more intelligent, adaptive teleoperation interfaces.

## 7.4. Environment-based Adaptation

### 7.4.1. Motivation

Small, affordable and lightweight aerial robots, for example, quadrotors equipped with low-cost cameras are getting more and more used in a wide range of scenarios from guidance to assistance applications in human-populated, indoor and GPS-denied environments like warehouses [219]. In those environments it is often possible to add artificial features also known as fiducials in order to simplify, for example, pose estimation where a correspondence between points in the real-world and the 2d image projection needs to be established. To this end, fiducial markers such as Aruco [160], April tags [220] and ArTag [221] are capable of detecting artificial landmarks at high frame rate. Those landmarks provide enough correspondences to compute the camera pose if the extrinsic and intrinsic parameters of the camera are known.

Although fiducial markers simplify pose estimation for a moving platform, detecting markers is challenged by motion blur and varying environmental conditions like changing lighting situations. Those varying conditions significantly influence the detection rate and accuracy of the fiducial marker detectors. In order to cope with those conditions, marker detectors provide – like many other perception algorithms – a set of configurable parameters which can be tuned to improve the performance even in the presence of varying environmental situations. However, as Crowley, Hall and Emonent point out in [222] tuning parameters is a labor intensive exercise performed by highly skilled experts.

Simply ignoring varying environmental conditions will lead to a degraded performance of the marker detector and thus robots' ability to navigate as pose estimation would not be possible anymore. In robotics where increased autonomy is desired the tuning of those parameters should be performed preferably without human intervention. Thus, this section focuses on

enabling a robot to autonomously select a marker detector configuration in the presence of continuous and discrete lighting changes faced by robots in real-world scenarios.

The motivation to focus solely on lighting changes is based on the fact that robots deployed in long-term scenarios – from several hours to several days – need to cope with them in order to robustly provide their services as their perception functions are immediately effected by them.

The selection is achieved by instantiating the reference architecture proposed in Section 6.2. To this end, different marker detector configurations have been modeled with RPSL (see Chapter 3) and evaluated during a training phase. The resulting RPSL domain models are stored in the repository. The selector component integrates a simple, yet powerful algorithm which enables to adapt to lighting changes immediately. This algorithm employs the lighting condition represented as a histogram as a criteria to select a marker detector configuration.


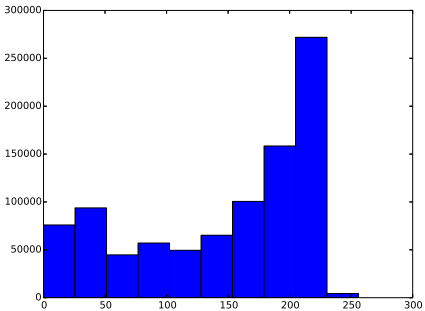
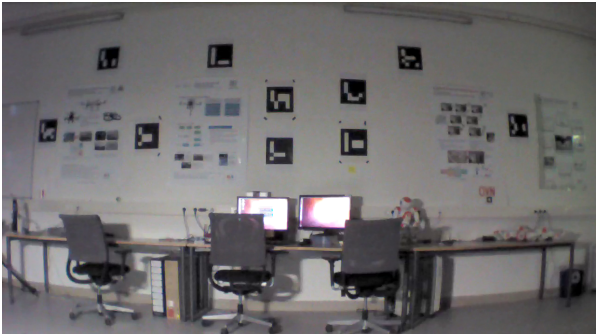
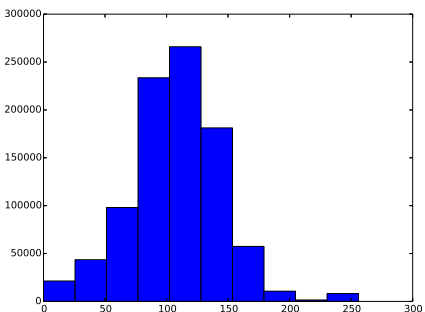

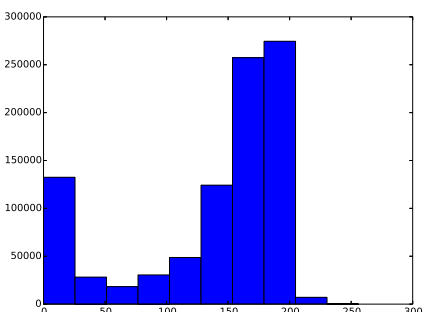
#### 7.4.2. Context Monitoring

In order to interpret the current lighting condition as environmental context, the context monitor takes the RGB camera image as an input, converts it to a grayvalue image and computes a corresponding histogram (see Table 7.5) which is stored in the repository.

#### 7.4.3. Repository

In this work RPSL is used to represent perception graphs expressing different configurations of the Aruco [160] marker detection algorithm. The resulting domain models are stored in the repository. The Aruco fiducial marker approach allows to tune two parts of the core Aruco detection algorithm, namely configurations for the adaptive thresholding and the type of method to refine detected corners, for example, the well-known Harris [223] corner detector. In order to configure the adaptive thresholding two parameters are employed, namely the block size of the pixel neighborhood which is used to calculate a threshold value for the pixel and the constant subtracted from the mean. All of those parameters are real-valued. The adaptive thresholding takes a grayscale image and computes a binary image in which Aruco fiducial markers are identified. Thus, wrongly configuring the adaptive threshold parameters will – in some lighting conditions – lead to binary images where the marker is only partially visible or even disappears.

In order to identify feasible configurations for different lighting situations a training phase has been performed. The training were carried out in a static scene with ten visible Aruco markers (see Table 7.5). Here, different Aruco configurations were applied for different lighting situations. In case a specific Aruco configuration performed good – at least 5 out of 10 markers were detected – the histogram of the grayvalue image of the current scene with a bin size of 10 were stored together with the current Aruco configuration. In total five different Aruco

Scene	Histogram
	
	
	

**Table 7.5.** Some examples of different lighting situations and their corresponding grayvalue histograms occurred during the training phase.

configurations and their corresponding histograms were stored in the repository. Those configurations differ mainly in the parameters for the adaptive thresholding.

#### 7.4.4. Perception Graph Selection

The main objective of the perception graph selection module is to identify a Aruco configuration suitable for the current lighting situation. The execution of the selection module is triggered by a notification of the repository whenever the context monitor inserts a new histogram (see Section 7.4.2). The selector iterates over each perception graph implementing the Aruco marker

detector feature (see Algorithm 3) and computes the similarity of the histogram attached to each perception graph with the current one obtained from the repository. As a similarity measure the Kullback-Leibler divergence is used. The divergence is defined as follows

$$KL(P, Q) = \sum_{x \in X} P(x) \cdot \log \frac{P(x)}{Q(x)} \quad (7.4)$$

and is a suitable similarity measure for this task as the lighting situation is modeled as a distribution of pixels over intervals (histogram).

#### 7.4.5. Perception Graph Deployment Infrastructure

As in Section 6.3.4 each perception graph is expressed as a process having a systemd service description. Based on the selected perception graph the process is started respectively stopped.

Scenario	$\mu$	$\sigma$	Number of switches
Continuous	$\approx 0.57\text{s}$	$\approx 0.23\text{s}$	7
Discrete	$\approx 0.64\text{s}$	$\approx 0.15\text{s}$	45

**Table 7.6.** The timing behavior for the adaptation.

#### 7.4.6. Experiments

Two challenging scenarios with time-varying lighting conditions have been defined for evaluating the proposed approach. Both scenarios are performed in a laboratory and the lighting conditions were controlled manually by turning several lights on/off and/or activating/deactivating rolling shutters. The first scenario – called continuous scenario – contains continuously decreasing and increasing lighting conditions. The second scenario – called discrete scenario – contains rapid and sporadic lighting changes. In both scenarios a camera was placed in front of a wall labeled with Aruco markers; a similar setup as for training (see Table 7.5). For the sake of evaluation an image stream of around 120 seconds has been recorded for both scenarios. It is important to note that the evaluation scenarios are different in terms of the concrete lighting situation and their particular duration than the one used for training. In order to evaluate whether or not the adaptive approach is beneficial in terms of detecting more or fewer markers than individual configurations, both scenarios have been replayed to the individual configurations of the Aruco marker detector and the adaptive approach. As seen in Table 7.7 the adaptive approach performs best – just looking at the average value of detected markers – for the continuous scenario and third for the discrete scenario. However, assessing

Scenario	Graph	$\mu$	$\sigma$
Continuous	with adaptation	$\approx 5.88$	$\approx 3.50$
	A	$\approx 1.32$	$\approx 1.38$
	B	$\approx 5.57$	$\approx 3.82$
	C	$\approx 3.45$	$\approx 3.14$
	D	$\approx 4.63$	$\approx 3.71$
	E	$\approx 0.39$	$\approx 1.33$
Discrete	with adaptation	$\approx 6.37$	$\approx 3.34$
	A	$\approx 0.73$	$\approx 1.04$
	B	$\approx 7.70$	$\approx 2.20$
	C	$\approx 3.56$	$\approx 2.86$
	D	$\approx 6.45$	$\approx 2.80$
	E	$\approx 0.00$	$\approx 0.03$

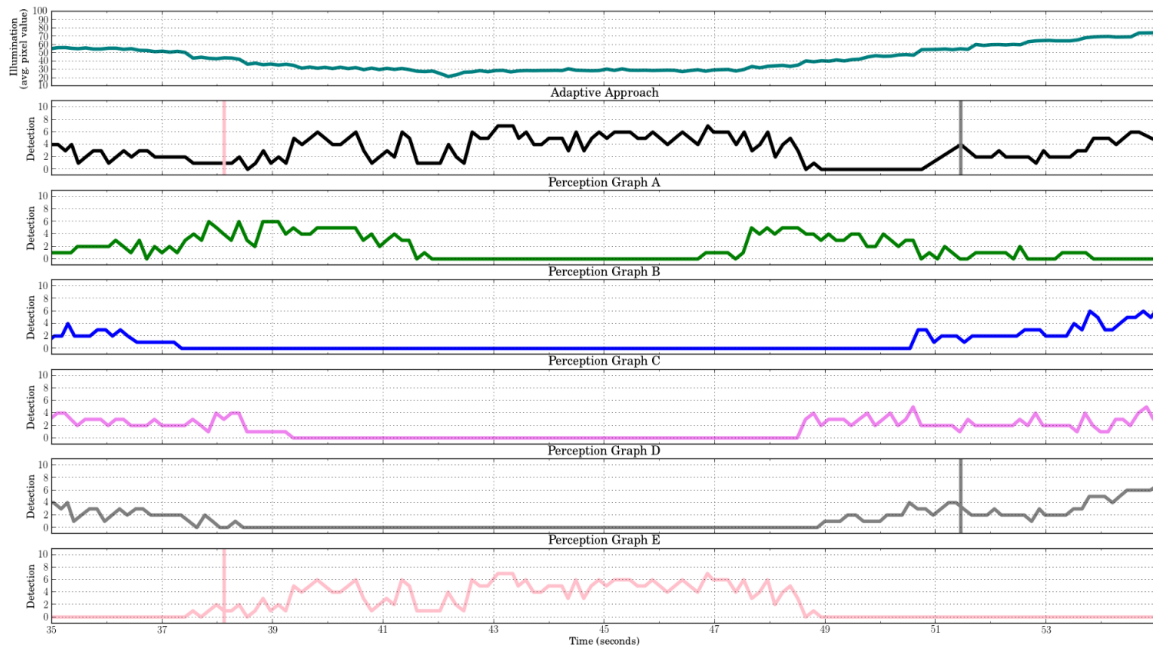
**Table 7.7.** Performance – number of detected markers – of the adaptive approach with respect to the individual configurations.

solely the average performance is not beneficial as it is expected that an adaptive approach is in particular beneficial in situations where a configuration needs to be selected in order to outperform all the other available configurations. Such a situation is visualized in Figure 7.8 which shows an excerpt of the continuous scenario. The first row depicts the lighting condition as an average graylevel pixel value over time. The second row shows the performance of the adaptive approach – the number of detected markers over time – and the remaining rows show the performance of the individual configurations. In this particular situation only configuration *E* is able to detect markers which is also selected by the adaptive approach. Afterwards and with some delay the adaptive approach switches from configuration *E* to *B*.

The average adaptation time, that is the time required to select and execute a perception graph has been measured as well. As shown in Table 7.6 for both scenarios the adaptation time is quite similar. For the continuous scenario seven switches were performed and for the discrete scenario 45 switches were performed.

#### 7.4.7. Related Work and Discussion

From a machine learning perspective the proposed approach can be considered as an instance-based learning approach as whenever a new lighting situation is perceived it is compared with those seen in training. Although the proposed approach is – from a learning and computer vision perspective – not novel, it demonstrates how RPSL models can be used for training.



**Figure 7.8.** An excerpt of the continuous scenario showing the performance of the adaptive approach (second row) vs. individual configurations. The colored vertical bars denote the point in time where a selected perception graph became active.

Interestingly, those models are employed later to realize adaptation in the presence of varying context variations. The results are promising, but it remains to be investigated whether the approach can be used also for realistic robotic applications like long-term navigation where more data is required for learning. In order to tackle these more challenging scenarios it might be beneficial to combine several contextual information, namely lighting, occlusion and blur. However, this would also require to investigate more advanced similarity measures which can incorporate several attributes expressing heterogeneous context conditions.

## 7.5. Summary

This chapter presented three different robot perception systems which are capable to cope with varying context-conditions, namely task, platform and environment context. All of them are based on the reference architecture proposed in Chapter 6. The architecture served not only as a blueprint for structuring the development, but also supported the identification and implementation of innovative components required to meet the application requirements. For example, the platform-based adaptation system (see Section 7.3) focuses on elaborated context monitoring whereas the task-based adaptation (see Section 7.2) focuses on advanced perception feature selection. Having said this, the selection of suitable perception abilities is done on different levels of abstraction, namely two systems are working on the perception



graph level (see Sections 7.3 and 7.4) and one system works on the perception feature level (see Section 7.2). Although adaptation is considered in this work as a selection of features respectively perception graphs and not as a modification of existing features/graphs it is sufficient to cope with rather complex context variations.



# Chapter 8.

## Conclusion

*“Once we accept our limits, we go beyond them.”*

— Albert Einstein, 1879–1955

This section concludes the thesis by summarizing the main contributions from Chapters 2–7 and providing suggestions for future work.

### 8.1. Contributions

This thesis sets out to improve robotic software engineering with a particular focus on the specification, deployment and adaptation of complex robot perception systems. This is achieved by establishing a structured, formal and model-based development approach. The structure is based on architectural views which serve as guiding principles to facilitate the analysis of the assessed domain. Based on the domain analysis, concepts and their constraints among each other have been identified and rigorously formalized as metamodels. The analysis has been performed in a well-defined way, which ensures that both commonalities and variabilites of robot perception systems are captured in the metamodels.

In order to enable domain experts to specify their design decisions of robot perception systems, two internal, domain-specific languages are proposed, namely RPSL and DepSL. Both languages conform to the metamodels and provide suitable abstractions and notations to specify robot perception systems. Having RPSL and DepSL at their disposal domain experts can now not only express and communicate their design decisions, but also verify them as the specifications are interpretable. This is achieved by implementing checks which ensure that RPSL and DepSL domain models comply with the formalized constraints and metamodels.

Another challenge tackled in this thesis is how to cope with the inherent complexity of robot perception systems which roots in their functional, architectural and deployment variability. In order to tame this complexity and to deal with variability in a systematic manner this thesis

introduces a design space exploration approach. The approach is supported by software tools which enables domain experts to explore and assess – in a semi-automatic manner – all possible implementations yielding from combining functional and architectural variability.

The approach paves the way to establish a step-wise, iterative design and development of robot perception systems comprising three steps. Firstly, RPSL is utilized to specify functional and architectural variability in an individual manner which leads to orthogonal domain models. Secondly, those models are combined in the exploration phase to assess expected and sometimes even unexpected design alternatives. Thirdly, based on the insights gained in the previous step, RPSL domain models are possibly refined.

Model-based approaches in robotics and other domains are often tailored and limited to be used by humans at design time. This work in contrast goes one step further by proposing labeled property graphs and graph databases as powerful means to apply models at run time and to grant robots access to those models. The labeled property graph model is a simple, yet powerful means to store, compose and query domain models originating not only from the robotic perception domain. Also heterogenous models originating from different functional domains and development phases can be combined in the graph to derive implicitly defined information. This is achieved by implementing domain and application-specific semantic queries operating on the labeled property graph. Thus, deriving implicitly defined knowledge raises the robots awareness about its capabilities and corresponding software implementations.

Making RPSL and DepSL domain models accessible for robots' – as part of their knowledge repository – enables them to deploy robot perception systems on their own. To this end, this thesis proposes a reference architecture which provides not only the means to integrate the repository, but also to develop additional components required for deploying robot perception systems even in the presence of varying resource conditions.

The reference architecture served as a blueprint for the implementation of three different robot perception systems. All of them are capable to cope with varying context conditions faced at run time and to provide their service by integrating application-specific adaptation components. Those components derive adaptation actions based on the perceived context and on explicitly expressed knowledge about their perception abilities. Thus, RPSL and DepSL domain models are used to close the gap between design time and run time concerns.

## 8.2. Suggestions for Future Work

This thesis opens up new, interesting directions for further research in robotic software engineering. Some of those directions are motivated and discussed in the following paragraphs.

Generally speaking, it is desirable that model-based approaches are used throughout a complete development process such that each development phase is enhanced by DSLs. As both

RPSL and DepSL are developed for specific tasks, they obviously support only some phases, namely the functional design, capability building and the system deployment phase of the RAP (see Chapter 2), whereas other phases are not supported. A future direction of research is not to extend RPSL and DepSL with means to enhance additional development phases, but to integrate and link them with approaches tailored for phases like the scenario building phase. Integrating RPSL and DepSL with approaches, for example from the scenario building phase, would enable domain experts and developers to trace requirements and to link them with specifications of perception abilities, perception graphs and their deployment. Such an integration would also pave the way to extend the design space exploration approach presented in Chapter 4 to include additional variability dimensions such as deployment and platform variability.

In order to foster the above mentioned research, another promising direction of research is to perform methodologically sound user studies in the context of real-world robot application development. In this connection, RPSL and DepSL could be evaluated by users to model not only robot perception systems, but also other robotic sub-systems like navigation, manipulation and so forth. Such a user study would help to identify – if any – missing concepts in the robot perception domain and whether or not the concepts available in RPSL and DepSL can be employed to model other systems. It is worth to note that such a user study should take into account the complete application development process in order to identify requirements for software tools.

Although this research aims to close the gap between design time and run time, it is worth noting that robots are mainly considered to be consumer and not producer of domain models. However, for long-term autonomy scenarios robots need also to be producer of domain models. To achieve this, more research to define richer run time models incorporating experience data and performance profiles of perception graphs and their deployment is required. Having such run time models would also pave the way to include this kind of information also for the sake of run time adaptation. Thus, improving the performance of robot perception systems throughout the whole life-cycle.

Last but not least another promising future direction of research is to investigate in more detail the costs which are involved in adapting robot perception systems at run time. Those costs are typically on different levels of abstraction, for example, settling time, resources required to perform adaptation, time to come up with adaptation actions and so forth. Investigating those costs should be done always with respect to some application and domain-requirements as, for example, in one application a particular cost might be tolerable whereas for another application they are not acceptable. In order to include those costs and requirements in the adaptation algorithms means to express them are required.



## Appendix A.

# Alloy Model of Perception Features

```
abstract sig FeatureTree{
  root: Feature
}{
  root.~(contain+spec)=none
}

abstract sig Feature {
  spec: lone Feature,
  contain: set Feature,
  excluded: set Feature,
  required: set Feature
}{
  some contain implies no spec
  some spec implies no contain
  let rel= (@spec+@contain){ // rel is the set of all edges composing the feature tree
    this not in this.^rel // prevent loops in the feature tree
    this.*rel & required.@excluded =none // given a feature f, features required by f
    can't exclude f and f's parents
    one (this.*rel & FeatureTree.root) // there's only one root feature in the parents
    of a given feature.
    lone (this.~@contain + this.~@spec )// any feature has at most one parent.
    excluded & required = none // a feature can't exclude and require the same feature

    excluded & this.*rel=none // a feature can't exclude itself or its children
    excluded & this.*~rel=none // a feature can't exclude itself or its parent
    required & this.*rel=none // a feature can't require itself or its children
    required & this.*~rel=none // a feature can't require itself or its parent
  no disj f1,f2:FeatureTree | this in f1.root.*rel and ((excluded+required) & f2.root
  .*rel) !=none // features of one feature tree can't exclude or require features
  of another feature tree.
  all x: excluded+required | some f:Feature | some disj f2,f3 :Feature | f2+f3 in f.
  @contain and (this+x) in (f2+f3).*rel and (this & f2.*rel) +(x & f3.*rel) =none
  // a feature can require or exclude another one only if both have ancestors
  which are (or are themselves) different alternatives of a same containment
  // for a given feature "this", let x be the set of excluded and required feature.
  We don't want another feature f that is not an ancestor of this but which is an
  specialisation alternative of an ancestor of this and an ancestor of x.
  all x:excluded+required | no f:Feature | f not in this.*~rel and one f.~@spec and
  f.~@spec in this.*~rel and x in f.*rel // required and excluded features should
  be in the same specialisation branch.
}
}
```

Figure A.1. The Alloy model for the perception features.





## Bibliography

- [1] M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mösenlechner, D. Pangercic, T. Rühr, and M. Tenorth, "Robotic roommates making pancakes," in *Proceedings of the IEEE-RAS International Conference on Humanoid Robots*, 2011.
- [2] L. Iocchi, D. Holz, J. Ruiz-del Solar, K. Sugiura, and T. van der Zant, "RoboCup@Home: Analysis and results of evolving competitions for domestic and service robots," *Artificial Intelligence*, vol. 229, pp. 258–281, 2015.
- [3] S. Srinivasa, D. Berenson, M. Cakmak, A. Collet Romea, M. Dogar, A. Dragan, R. A. Knepper, T. D. Niemueller, K. Strabala, J. M. Vandeweghe, and J. Ziegler, "Herb 2.0: Lessons learned from developing a mobile manipulator for the home," *Proceedings of the IEEE*, vol. 100, no. 8, pp. 2410–2428, 2012.
- [4] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," in *Proceedings of the 19th National Conference on Innovative Applications of Artificial Intelligence*, 2007.
- [5] T. Niemüller, D. Ewert, S. Reuter, A. A. Ferrein, S. Jeschke, and G. Lakemeyer, *RoboCup Logistics League Sponsored by Festo: A Competitive Factory Automation Testbed*, vol. 8371 of *Lecture Notes in Artificial Intelligence*, pp. 336–347. Springer, 2014.
- [6] G. K. Kraetzschmar, N. Hochgeschwender, W. Nowak, F. Hegger, S. Schneider, R. Dwiputra, J. Berghofer, and R. Bischoff, *RoboCup@Work: Competing for the Factory of the Future*, vol. 8992 of *Lecture Notes in Computer Science*, pp. 171–182. Springer, 2015.
- [7] L. Marconi, C. Melchiorri, M. Beetz, D. Pangercic, R. Siegwart, S. Leutenegger, R. Carloni, S. Stramigioli, H. Bruyninckx, P. Doherty, A. Kleiner, V. Lippiello, A. Finzi, B. Siciliano, A. Sala, and N. Tomatis, "The sherpa project: smart collaboration between humans and ground-aerial robots for improving rescuing activities in alpine environments," in *Proceedings of the IEEE International Symposium on Safety, Security, and Rescue Robotics*, 2012.
- [8] R. Sheh, T. Kimura, E. Mihankhah, J. Pellenz, S. Schwertfeger, and J. Suthakorn, "The RoboCupRescue robot league: Guiding robots towards fieldable capabilities," in *Proceedings of the IEEE Workshop on Advanced Robotics and its Social Impacts*, 2011.
- [9] U. Reiser, C. Connette, J. Fischer, J. Kubacki, A. Bubeck, F. Weisshardt, T. Jacobs, C. Parlitz,

- M. Hägele, and A. Verl, "Care-O-bot 3: Creating a product vision for service robot applications by integrating design and technology," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.
- [10] R. Bischoff, U. Huggenberger, and E. Prassler, "Kuka youBot - a mobile manipulator for research and education," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2011.
- [11] G. K. Kraetzschmar, A. Shakhimardanov, J. Paulus, N. Hochgeschwender, and M. Reckhaus, "Specifications of architectures, modules, modularity, and interfaces for the brocre software platform and robot control architecture workbench," 2010. BRICS Project Deliverable D2.2.
- [12] B. Siciliano and O. Khatib, *Springer Handbook of Robotics*. Springer, 2007.
- [13] A. Nordmann, N. Hochgeschwender, D. Wiegand, and S. Wrede, "A survey on domain-specific modeling and languages in robotics," *Journal for Software Engineering Robotics*, vol. 7, no. 1, 2016.
- [14] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A survey on domain-specific languages in robotics," in *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Springer, 2014.
- [15] S. Schneider, N. Hochgeschwender, and G. K. Kraetzschmar, "Structured design and development of domain-specific languages in robotics," in *Proceedings of the International Conference on Simulation, Modeling and Programming for Autonomous Robots*, Springer, 2014.
- [16] J. Sanchez, S. Schneider, N. Hochgeschwender, G. K. Kraetzschmar, and P. G. Plöger, "Context-based adaptation of in-hand slip detection for service robots," in *Proceedings of the IFAC Symposium on Intelligent Autonomous Vehicles*, 2016.
- [17] L. Gherardi and N. Hochgeschwender, "RRA: Models and tools for robotics run-time adaptation," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2015.
- [18] D. Brugali, "Model-driven software engineering in robotics: Models are designed to use the relevant things, thereby reducing the complexity and cost in the field of robotics," *IEEE Robotics Automation Magazine*, vol. 22, pp. 155–166, Sept 2015.
- [19] H. Schichl, *Modeling Languages in Mathematical Optimization*, ch. Models and the History of Modeling, pp. 25–36. Springer, 2004.
- [20] P. J. Ashenden, *The Designer's Guide to VHDL, Volume 3, Third Edition (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., 2008.
- [21] K. Kundert and O. Zinke, *The Designer's Guide to Verilog-AMS*. Springer, 2013.
- [22] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*,

- vol. 39, pp. 25–31, Feb. 2006.
- [23] A. R. da Silva, “Model-driven engineering: A survey supported by the unified conceptual model,” *Computer Languages, Systems & Structures*, vol. 43, pp. 139 – 155, 2015.
- [24] K. Pohl, H. Hnninger, R. Achatz, and M. Broy, *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*. Springer, 2012.
- [25] C. Atkinson and T. Kühne, “Model-driven development: A metamodeling foundation,” *IEEE Software*, vol. 20, pp. 36–41, Sept. 2003.
- [26] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.
- [27] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.
- [28] Object Management Group (OMG), “Meta Object Facility (MOF) Core Specification Version 2.0.” <http://www.omg.org/cgi-bin/doc?formal/2006-01-01>, 2006. Accessed: 2016-02-25.
- [29] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2 ed., 2003.
- [30] J. M. Spivey, *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge University Press, 1988.
- [31] D. Jackson, “Alloy: A lightweight object modelling notation,” *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 2, pp. 256–290, 2002.
- [32] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.
- [33] B. Selic, “The pragmatics of model-driven development,” *IEEE Software*, vol. 20, pp. 19–25, Sept. 2003.
- [34] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, “Explicit transformation modeling,” in *Models in Software Engineering: Workshops and Symposia at MODELS 2009*, Springer, 2010.
- [35] B. Selic and S. Grard, *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. Morgan Kaufmann, 1st ed., 2013.
- [36] A. van Deursen, P. Klint, and J. Visser, “Domain-specific languages: An annotated bibliography,” *SIGPLAN Notes*, vol. 35, pp. 26–36, June 2000.
- [37] J. Bentley, “Programming pearls: Little languages,” *Communications of the ACM*, vol. 29, no. 8, pp. 711–721, 1986.
- [38] M. Fowler, *Domain Specific Languages*. Addison-Wesley Professional, 1st ed., 2010.

- [39] T. Dinkelaker, M. Monperrus, and M. Mezini, "Supporting Variability with Late Semantic Adaptations of Domain-Specific Modeling Languages," in *Proceedings of the First International Workshop on Composition and Variability*, 2010.
- [40] S. Günther and T. Cleenewerck, "Design principles for internal domain-specific languages: A pattern catalog illustrated by Ruby," in *Proceedings of the 17th Conference on Pattern Languages of Programs*, ACM, 2010.
- [41] F. Campagne and F. Campagne, *The MPS Language Workbench, Vol. 1*. USA: CreateSpace Independent Publishing Platform, 1st ed., 2014.
- [42] L. C. Kats and E. Visser, "The spoofax language workbench: Rules for declarative specification of languages and ides," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.
- [43] S. Kelly, K. Lyytinen, and M. Rossi, "Metaedit+: A fully configurable multi-user and multi-tool case and came environment," in *Proceedings of the 8th International Conference on Advances Information System Engineering*, Springer, 1996.
- [44] S. Erdweg, T. van der Storm, M. VÃlter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning, "Evaluating and comparing language workbenches: Existing results and benchmarks for the future," *Computer Languages, Systems & Structures*, vol. 44, Part A, pp. 24 – 47, 2015.
- [45] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.
- [46] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Prentice Hall PTR, 2nd ed., 2002.
- [47] A. Cockburn, *Agile Software Development*. Addison-Wesley, 2002.
- [48] C. B. Nielsen, P. G. Larsen, J. Fitzgerald, J. Woodcock, and J. Peleska, "Systems of systems engineering: Basic concepts, model-based techniques, and research directions," *ACM Computing Survey*, vol. 48, pp. 1–41, Sept. 2015.
- [49] R. Höhn and S. Höppner, *Das V-Modell XT*. Springer, 2009.
- [50] Y. Brodskiy, R. Wilterdink, S. Stramigioli, and J. Broenink, "Fault avoidance in development of robot motion-control software by modeling the computation," in *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Springer, 2014.
- [51] M. Frigerio, J. Buchli, and D. G. Caldwell, "Code generation of algebraic quantities for robot controllers," in *Proceedings of the IEEE/RSJ International Conference on Intelligent*

- Robots and Systems*, 2012.
- [52] Y. J. Kanayama and C. T. Wu, "It's time to make mobile robots programmable," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2000.
- [53] C. A. Jara, F. A. Candelas, P. Gil, F. Torres, F. Esquembre, and S. Dormido, "Ejs+ej srl: An interactive tool for industrial robots simulation, computer vision and remote operation," *Robotics and Autonomous Systems*, vol. 59, pp. 389–401, June 2011.
- [54] T. D. Laet, S. Bellens, H. Bruyninckx, and J. D. Schutter, "Geometric relations between rigid bodies (part 2): From semantics to software," *IEEE Robotics Automation Magazine*, vol. 20, pp. 91–102, June 2013.
- [55] A. Shakhimardanov, *Composable Robot Motion Stack*. PhD thesis, KU Leuven, 2015.
- [56] M. Frigerio, J. Buchli, and D. G. Caldwell, "Model based code generation for kinematics and dynamics computations in robot controllers," in *Workshop on Software Development and Integration in Robotics*, St. Paul, Minnesota, USA, 2012.
- [57] E. Aertbeliën and J. De Schutter, "eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014.
- [58] I. Kitagishi, T. Machino, A. Nakayama, S. Iwaki, and M. Okudaira, "Development of motion data description language for robots based on extensible markup language-realization of better understanding and communication via networks," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2002.
- [59] G. S. Hornby, H. Lipson, and J. B. Pollack, "Evolution of generative design systems for modular physical robots," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2001.
- [60] S. Schneider, N. Hochgeschwender, and G. K. Kraetzschmar, "Declarative specification of task-based grasping with constraint validation," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014.
- [61] T. Henderson and E. Shilcrat, "Logical sensor systems," *Journal of Robotic Systems*, vol. 1, no. 2, pp. 169–193, 1984.
- [62] A. K. Ramaswamy, B. Monsuez, and A. Tapus, "Solution space modeling for robotic systems," *Journal for Software Engineering Robotics*, vol. 5, no. 1, pp. 89–96, 2014.
- [63] J. L. Gordillo, "Le: a high level language for specifying vision verification tasks," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 1991.
- [64] A. Feniello, H. Dang, and S. Birchfield, "Program synthesis by examples for object repositioning tasks," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014.

- [65] D. C. MacKenzie, J. M. Cameron, and R. C. Arkin, "Specification and execution of multiagent missions," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1995.
- [66] N. Dantam, A. Hereid, A. D. Ames, and M. Stilman, "Correct software synthesis for stable speed-controlled robotic walking.," in *Robotics: Science and Systems*, 2013.
- [67] J. P. Buch, J. S. Laursen, L. C. Sørensen, L.-P. Ellekilde, D. Kraft, U. P. Schultz, and H. G. Petersen, "Applying simulation and a domain-specific language for an adaptive action library," in *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Springer, 2014.
- [68] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann, "A new skill based robot programming language using uml/p statecharts," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2013.
- [69] M. Klotzbucher, R. Smits, H. Bruyninckx, and J. De Schutter, "Reusable hybrid force-velocity controlled motion specifications with executable domain specific languages," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011.
- [70] A. Steck and C. Schlegel, "Managing execution variants in task coordination by exploiting design-time models at run-time," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011.
- [71] S. Joyeux, F. Kirchner, and S. Lacroix, "Managing plans: Integrating deliberation and reactive execution schemes," *Robotics and Autonomous Systems*, vol. 58, no. 9, pp. 1057–1066, 2010.
- [72] A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede, "A survey on domain-specific modeling and languages in robotics," *Journal of Software Engineering for Robotics*, vol. 7, no. 1, pp. 75–99, 2016.
- [73] R. M. Keller, "Fel programmer's guide," tech. rep., University of Utah, 1982.
- [74] C. Finucane, G. Jing, and H. Kress-Gazit, "Ltlmop: Experimenting with language, temporal logic and robot control," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010.
- [75] M. Schindler, *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML-P*. PhD thesis, RWTH Aachen University, 2012.
- [76] H. Krahn, *MontiCore: agile Entwicklung von domÄd' nenspezifischen Sprachen im Software-Engineering*. PhD thesis, RWTH Aachen University, 2010.
- [77] M. T. Mason, "Compliance and force control for computer controlled manipulators," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 11, pp. 418–432, June 1981.
- [78] D. Mcdermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and

- D. Wilkins, "Pddl - the planning domain definition language," Tech. Rep. TR-98-003, Yale Center for Computational Vision and Control, 1998.
- [79] E. P. D. Pednault, "Adl and the state-transition model of action," *Journal of Logic and Computation*, vol. 4, no. 5, pp. 467–512, 1994.
- [80] F. Stulp and M. Beetz, "Combining declarative, procedural and predictive knowledge to generate and execute robot plans efficiently and robustly," *Robotics and Autonomous Systems Journal (Special Issue on Semantic Knowledge)*, vol. 56, no. 11, pp. 967–979, 2008.
- [81] E. Erdem, K. Haspalamutgil, C. Palaz, V. Patoglu, and T. Uras, "Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2011.
- [82] G. Havur, K. Haspalamutgil, C. Palaz, E. Erdem, and V. Patoglu, "A case study on the tower of hanoi challenge: Representation, reasoning and execution," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2013.
- [83] C. Schlegel, A. Steck, D. Brugali, and A. Knoll, "Design abstraction and processes in robotics: From code-driven to model-driven engineering," in *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, 2010.
- [84] F. J. Ortiz, D. Alonso, F. Rosique, F. Sánchez-Ledesma, and J. A. Pastor, "A component-based meta-model and framework in the model driven toolchain c-forge," in *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Springer, 2014.
- [85] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand, "Genom3: Building middleware-independent robotic components," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2010.
- [86] T. Abdellatif, S. Bensalem, J. Combaz, L. De Silva, and F. Ingrand, "Rigorous design of robot software: A formal component-based approach," *Robotics and Autonomous Systems*, vol. 60, pp. 1563–1578, Dec. 2012.
- [87] L. Gherardi and D. Brugali, "Modeling and reusing robotic software architectures: The hyperflex toolchain," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2014.
- [88] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.
- [89] B. Dittes and C. Goerick, "Intelligent system architectures - comparison by translation," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*,

- 2011.
- [90] M. Morelli and M. D. Natale, "Control and scheduling co-design for a simulated quadcopter robot: A model-driven approach," in *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Springer, 2014.
- [91] N. Gobillot, C. Lesire, and D. Doose, "A modeling framework for software architecture specification and validation," in *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Springer, 2014.
- [92] C. Datta, C. Jayawardena, I. H. Kuo, and B. A. MacDonald, "Robostudio: A visual programming environment for rapid authoring and customization of complex services on a personal service robot," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012.
- [93] S. Tousignant, E. Van Wyk, and M. Gini, "Xrobots: A flexible language for programming mobile robots based on hierarchical state machines," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2012.
- [94] G. Biggs and B. A. MacDonald, "Specifying robot reactivity in procedural languages," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006.
- [95] A. Rusakov, J. Shin, and B. Meyer, "Simple concurrency for robotics with the roboscoop framework," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014.
- [96] F. Fleurey and A. Solberg, "A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, Springer, 2009.
- [97] A. Ramaswamy, B. Monsuez, and A. Tapus, "SafeRobots: A model-driven framework for developing robotic systems," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014.
- [98] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan, "Silver: An extensible attribute grammar system," *Electronic Notes Theoretical Computer Science*, vol. 203, pp. 103–116, Apr. 2008.
- [99] S. Dhoubib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "RobotML, a domain-specific language to design, simulate and deploy robotic applications," in *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Springer, 2012.
- [100] D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor, and B. Alvarez, "V3cmm: A 3-view component meta-model for model-driven robotic software development," *Journal of Software Engineering for Robotics*, vol. 1, no. 1, 2010.



- [101] H. Wei, X. Duan, S. Li, G. Tong, and T. Wang, "A component based design framework for robot software architecture," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.
- [102] A. Nordmann, S. Wrede, and J. Steil, "Modeling of movement control architectures based on motion primitives using domain-specific languages," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2015.
- [103] D. Cassou, S. Stinckwich, and P. Koch, "Using the diaspec design language and compiler to develop robotics systems," *CoRR*, vol. abs/1109.2806, 2011.
- [104] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Survey*, vol. 37, pp. 316–344, Dec. 2005.
- [105] R. d. A. Falbo, G. Guizzardi, and K. C. Duarte, "An ontological approach to domain engineering," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, pp. 351–358, ACM, 2002.
- [106] A. Abran, P. Bourque, R. Dupuis, and J. W. Moore, eds., *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, 2001.
- [107] P. Kruchten, "The 4+1 view model of architecture," *IEEE Software*, vol. 12, pp. 42–50, Nov. 1995.
- [108] J. Sanchez, "Robust and safe manipulation by sensor fusion of robotic manipulators and end-effectors," Master's thesis, Bonn-Rhein-Sieg University, March 2015.
- [109] J. A. Alcazar and L. G. Barajas, "Estimating object grasp sliding via pressure array sensing," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2012.
- [110] F. Hegger, N. Hochgeschwender, G. K. Kraetzschmar, and P. Plöger, "People detection in 3d point clouds using local surface normals," in *Proceedings of the Annual RoboCup International Symposium*, Springer, 2012.
- [111] C. A. Mueller, N. Hochgeschwender, P. G. Ploeger, and P. G. Plöger, "Towards Robust Object Categorization for Mobile Robots with Combination of Classifiers," in *Proceedings of the RoboCup International Symposium*, (Istanbul, Turkey), 2011.
- [112] L. Camargo, I. Ivanovska, A. Moriarty, M. Nguyen, S. Thoduka, D. Vazquez, A. Kuestenmacher, and P. G. Ploeger, "The b-it-bots@home 2016 team description paper," in *RoboCup*, (Leipzig, Germany), 2016.
- [113] S. Ahmed, T. Jandt, P. Kulkarni, O. Lima, A. Mallick, A. Moriarty, D. Nair, S. Thoduka, I. Awaad, R. Dwiputra, F. Hegger, N. Hochgeschwender, J. Sanchez, S. Schneider, and G. K. Kraetzschmar, "b-it-bots robocup@work team description paper," in *RoboCup*, (Leipzig, Germany), 2016.

- [114] “Robotics 2020: Multi-annual roadmap for robotics in eu-rope.” <http://sparc-robotics.eu/wp-content/uploads/2014/05/H2020-Robotics-Multi-Annual-Roadmap-ICT-2016.pdf>, 2015. Accessed: 2016-08-25.
- [115] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.
- [116] D. Brugali and P. Scandurra, “Component-based robotic engineering (part i) [tutorial],” *IEEE Robotics Automation Magazine*, vol. 16, pp. 84–96, December 2009.
- [117] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in *ICRA Workshop on Open Source Software*, 2009.
- [118] H. Bruyninckx, P. Soetens, and B. Koninckx, “The real-time motion control core of the Orocos project,” in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2003.
- [119] N. Ando, T. Suehiro, and T. Kotoku, “A software platform for component based rt-system development: Openrtm-aist,” in *Proceedings of the International Conference on Simulation, Modeling and Programming for Autonomous Robots*, Springer, 2008.
- [120] S. Fleury, M. Herrb, and R. Chatila, “G<sup>e</sup>nom: A tool for the specification and the implementation of operating modules in a distributed robot architecture,” in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1997.
- [121] A. Shakhimardanov, N. Hochgeschwender, and G. K. Kraetzschmar, “Component models in robotics software,” in *Proceedings of the Performance Metrics for Intelligent Systems Workshop*, ACM, 2010.
- [122] P. Gärdenfors, *Conceptual spaces - the geometry of thought*. MIT Press, 2000.
- [123] A. Svendsen, Ø. Haugen, and B. Møller-Pedersen, “Specifying a testing oracle for train stations—going beyond with product line technology,” in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, 2011.
- [124] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, “How amazon web services uses formal methods,” *Communications of the ACM*, vol. 58, no. 4, 2015.
- [125] A. Lin, M. Bond, and J. Clulow, “Modeling partial attacks with alloy,” in *Proceedings of the International Workshop on Security Protocols*, 2007.
- [126] M. Acher, P. Collet, P. Lahire, and R. France, “Comparing approaches to implement feature model composition,” in *Proceedings of the European Conference on Modelling Foundations and Applications*, Springer, 2010.

- [127] D. Brugali and M. Valota, "Software variability composition and abstraction in robot control systems," in *Proceedings of the International Conference on Computational Science and Its Applications*, Springer, 2016.
- [128] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 1st ed., 2012.
- [129] B. Adams and M. Raubal, "A metric conceptual space algebra," in *Proceedings of the International Conference on Spatial Information Theory*, pp. 51–68, Springer, 2009.
- [130] D. Ghosh, *DSLs in Action*. Greenwich, CT, USA: Manning Publications Co., 1st ed., 2010.
- [131] D. Flanagan and Y. Matsumoto, *The Ruby Programming Language*. O'Reilly, first ed., 2008.
- [132] S. Günther, "Agile dsl-engineering with patterns in ruby," Technical report (Internet) FIN-018-2009, Otto-von-Guericke-Universität Magdeburg, 2009.
- [133] "Sinatra, a DSL for quickly creating web applications." [www.sinatrarb.com/](http://www.sinatrarb.com/). [Online; accessed 03-January-2016].
- [134] "Rake, a make-like build utility for Ruby." <https://ruby.github.io/rake/>. [Online; accessed 03-January-2016].
- [135] P. Kruchten, *The Rational Unified Process: An Introduction, Second Edition*. Addison-Wesley, 2nd ed., 2000.
- [136] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*. Prentice Hall PTR, 1st ed., 2001.
- [137] D. S. Kolovos, R. F. Paige, T. Kelly, and F. A. C. Polack, "Requirements for domain-specific languages," in *Proceedings of the ECOOP Workshop on Domain-Specific Program Development*, (Nantes, France), 2006.
- [138] R. Tairas, M. Mernik, and J. Gray, "Using ontologies in the domain analysis of domain-specific languages," in *Proceedings of the Workshops and Symposia on Models in Software Engineering*, Springer, 2009.
- [139] C. Atkinson and C. Tunjic, "Criteria for orthographic viewpoints," in *Proceedings of the Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, ACM, 2014.
- [140] J. Hugues, "Aadl for cyber-physical systems: Semantics and beyond, validate what's next," in *The Robotics and Embedded Systems Seminar (RESS)*, (Berkeley, US), pp. 1–27, 2011.
- [141] L. Delligatti, *SysML Distilled: A Brief Guide to the Systems Modeling Language*. Addison-Wesley Professional, 1st ed., 2013.
- [142] J. T. M. Ingbergsson, U. P. Schultz, and D. Kraft, "Towards declarative safety rules for perception specification architectures," *CoRR*, vol. abs/1601.02778, 2016.

- [143] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.
- [144] E. Kang, E. Jackson, and W. Schulte, "An approach for effective design space exploration," in *Proceedings of the Monterey Conference on Foundations of Computer Software: Modeling, Development, and Verification of Adaptive Systems*, Springer, 2011.
- [145] L. Gammaitoni, P. Kelsen, and C. Glodt, "Designing languages using lightning," in *Proceedings of the International Conference on Software Language Engineering*, 2015.
- [146] T. Saxena and G. Karsai, "Mde-based approach for generalizing design space exploration," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, 2010.
- [147] L. Gammaitoni and P. Kelsen, "F-alloy: An alloy based model transformation language," in *Proceedings of the International Conference on Theory and Practice of Model Transformations*, 2015.
- [148] P. Abeles, "Speeding up surf," in *Proceedings of the International Symposium on Advances in Visual Computing*, Springer, 2013.
- [149] F. Amigoni, E. Bastianelli, J. Berghofer, A. Bonarini, G. Fontana, N. Hochgeschwender, L. Iocchi, G. Kraetzschmar, P. Lima, M. Matteucci, P. Miraldo, D. Nardi, and V. Schiavonati, "Competitions for benchmarking: Task and functionality scoring complete performance assessment," *IEEE Robotics Automation Magazine*, vol. 22, no. 3, pp. 53–61, 2015.
- [150] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, pp. 381–395, June 1981.
- [151] N. Sörensson and N. Een, "Minisat v1.13 - a sat solver with conflict-clause minimization.," tech. rep., Chalmers University of Technology, Sweden, 2002.
- [152] D. Le Berre and A. Parrain, "The Sat4j library, release 2.2," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, pp. 59–64, 2010.
- [153] R. Hourani, R. Jenkal, W. R. Davis, and W. Alexander, "Automated design space exploration for dsp applications," *Journal of Signal Processing Systems*, vol. 56, no. 2, pp. 199–216, 2009.
- [154] H. Oh and S. Ha, "Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, 2002.
- [155] C. Kiesner, G. Taentzer, and J. Winkelmann, "Visual ocl: A visual notation of the ob-

- ject constraint language." <http://tfs.cs.tu-berlin.de/voc1>. [Online; accessed 31-October-2016].
- [156] S. Kent, "Constraint diagrams: visualizing invariants in object-oriented models," in *ACM SIGPLAN Notices*, vol. 32, pp. 327–341, ACM, 1997.
- [157] M. Tenorth and M. Beetz, "KnowRob – A Knowledge Processing Infrastructure for Cognition-enabled Robots," *International Journal of Robotics Research*, vol. 32, pp. 566 – 590, April 2013.
- [158] A. Saxena, A. Jain, O. Sener, A. Jami, D. K. Misra, and H. S. Koppula, "Robo brain: Large-scale knowledge engine for robots," 2015.
- [159] S. Lemaignan, *Grounding the Interaction: Knowledge Management for Interactive Robots*. PhD thesis, CNRS - Laboratoire d'Analyse et d'Architecture des Systemes, Technische Universität München - Intelligent Autonomous Systems lab, 2012.
- [160] S. Garrido-Jurado, R. Muñoz Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, pp. 2280–2292, June 2014.
- [161] N. Hochgeschwender, M. Olivares-Mendez, H. Voos, and G. Kraetzschmar, "Context-based selection and execution of robot perception graphs," in *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation*, 2015.
- [162] E. Scioni, N. Huebel, S. Blumenthal, A. Shakhimardanov, M. Klotzbuecher, H. Garcia, and H. Bruyninckx, "Hierarchical hypergraphs for knowledge-centric robot systems: a composable structural meta model and its domain specific language npc4," *Journal of Software Engineering for Robotics*, vol. 7, no. 1, pp. 55–74, 2016.
- [163] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, Jan. 2003.
- [164] J. Bailey, F. Bry, T. Furche, and S. Schaffert, "Web and semantic web query languages: A survey," in *Proceedings of the International Summer School on Reasoning Web* (N. Eisinger and J. Małuszyński, eds.), Springer, 2005.
- [165] "neo4j." <http://neo4j.com/>. [Online; accessed 31-October-2016].
- [166] "Neo4j object-graph-mapper (ogm)." <http://neo4jrb.io/>. [Online; accessed 31-October-2016].
- [167] M. Fowler, "Active record." <http://www.martinfowler.com/eaCatalog/activeRecord.html>. [Online; accessed 31-October-2016].
- [168] T. Niemueller, G. Lakemeyer, and S. S. Srinivasa, "A Generic Robot Database and its Application in Fault Analysis and Performance Evaluation," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012.

- [169] E. Plugge, T. Hawkins, and P. Membrey, *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Berkely, CA, USA: Apress, 1st ed., 2010.
- [170] M. Tenorth and M. Beetz, "KnowRob – Knowledge Processing for Autonomous Personal Robots," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4261–4266, 2009.
- [171] J. F. Inglès-Romero, A. Lotz, C. Vicente-Chicote, and C. Schlegel, "Dealing with runtime variability in service robotics: Towards a dsl for non-functional properties," *CoRR*, vol. abs/1303.4296, 2013.
- [172] C. Hein, T. Ritter, and M. Wagner, "Model-driven tool integration with modelbus," in *Proceedings of the Workshop on Future Trends of Model-Driven Development, INSTICC*, 2009.
- [173] G. Biggs, T. Sakamoto, K. Fujiwara, and K. Anada, "Experiences with model-centred design methods and tools in safe robotics," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013.
- [174] Object Management Group, "Deployment and configuration of component-based distributed applications specification." <http://www.omg.org/spec/DEPL/4.0/>, 2004. [Online; accessed 03-January-2016].
- [175] L. Gherardi, *Variability Modeling and Resolution in Component-based Robotics Systems*. PhD thesis, University of Bergamo, 2013.
- [176] D. Calisi, A. Farinelli, G. Grisetti, L. Iocchi, D. Nardi, S. Pellegrini, D. Tipaldi, and V. A. Ziparo, "Uses of contextual knowledge in mobile robots," in *Proceedings of the Congress of the Italian Association for Artificial Intelligence*, Springer, 2007.
- [177] A. F. Garcia, C. M. Rubira, A. Romanovsky, and J. Xu, "A comparative study of exception handling mechanisms for building dependable object-oriented software," *Journal of Systems and Software*, vol. 59, no. 2, pp. 197 – 222, 2001.
- [178] "roslaunch." <http://wiki.ros.org/roslaunch>. [Online; accessed 03-January-2016].
- [179] "systemd System and Service Manager." <https://www.freedesktop.org/wiki/Software/systemd/>. [Online; accessed 03-January-2016].
- [180] "systemd Service." <https://www.freedesktop.org/software/systemd/man/systemd.service.html>. [Online; accessed 03-January-2016].
- [181] "systemctl." <https://www.freedesktop.org/software/systemd/man/systemctl.html>. [Online; accessed 03-January-2016].
- [182] N. Ando, S. Kurihara, G. Biggs, T. Sakamoto, H. Nakamoto, and T. Kotoku, "Software deployment infrastructure for component-based rt-systems," *Journal of Robotics and Mechatronics*, vol. 23, no. 3, pp. 350–359, 2011.

- [183] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based rt-system development: Openrtm-aist," in *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Springer, 2008.
- [184] U. Reiser, *Eine webbasierte Integrations -und Testplattform zur Unterstützung des verteilten Entwicklungsprozesses von komplexen Serviceroboter-Applikationen*. PhD thesis, Universität Stuttgart, 2014.
- [185] M. Mikic-Rakic and N. Medvidovic, "Architecture-level support for software component deployment in resource constrained environments," in *Proceedings of the IFIP/ACM Working Conference on Component Deployment*, Springer, 2002.
- [186] G. Mohanarajah, D. Hunziker, R. D'Andrea, and M. Waibel, "Rapyuta: A cloud robotics platform," *IEEE Transactions on Automation Science and Engineering*, vol. 12, pp. 481–493, April 2015.
- [187] J. Furrer, K. Kamei, C. Sharma, T. Miyashita, and N. Hagita, "Unr-pf: An open-source platform for cloud networked robotic services," in *Proceedings of the IEEE/SICE International Symposium on System Integration*, 2012.
- [188] "Kmr iiwa." <https://www.kuka.com/en-us/products/mobility/mobile-robot-systems/kmr-iiwa>. [Online; accessed 31-October-2016].
- [189] "Fraunhofer rob@work." <http://www.care-o-bot.de/en/rob-work.html>. [Online; accessed 31-October-2016].
- [190] T. Bauernhansl, M. ten Hompel, and B. Vogel-Heuser, eds., *Industrie 4.0 in Produktion, Automatisierung und Logistik*. Wiesbaden: Springer Vieweg, 2014.
- [191] A. Ferrein and G. Steinbauer, "20 years of robocup," *KI - Künstliche Intelligenz*, vol. 30, no. 3, pp. 225–232, 2016.
- [192] R. Soetens, R. van de Molengraft, and B. Cunha, "Robocup msl - history, accomplishments, current status and challenges ahead," in *RoboCup 2014: Robot World Cup XVIII*, Springer, 2015.
- [193] J. Pellenz, D. Dillenberger, and G. Steinbauer, "Novel rule set for the robocup rescue robot league," in *Proceedings of the IEEE International Symposium on Safety, Security, and Rescue Robotics*, 2016.
- [194] R. Dwiputra, J. Berghofer, A. Ahmad, I. Awaad, F. Amigoni, R. Bischoff, A. Bonarini, G. Fontana, F. Hegger, N. Hochgeschwender, L. Iocchi, G. Kraetzschmar, P. Lima, M. Matteucci, D. Nardi, V. Schiaffonati, and S. Schneider, "The rockin@work challenge," in *Proceedings of the International Symposium on Robotics*, 2014.
- [195] "Robocup@work rulebook 2017." <http://www.robocupatwork.org/download/rulebook-2017-01-24.pdf>. [Online; accessed 01-April-2017].

- [196] T. Niemueller, S. Zug, S. Schneider, and U. Karras, "Knowledge-based instrumentation and control for competitive industry-inspired robotic domains," *KI – Zeitschrift Künstliche Intelligenz*, pp. 1–11, 2016.
- [197] S. Schneider, F. Hegger, N. Hochgeschwender, R. Dwiputra, A. Moriarty, J. Berghofer, and G. K. Kraetzschmar, "Design and development of a benchmarking testbed for the factory of the future," in *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation*, 2015.
- [198] M. M. Richter and R. O. Weber, *Case-Based Reasoning: A Textbook*. Springer, 2013.
- [199] R. Ros, R. L. de Mantaras, J. L. Arcos, and M. Veloso, "Team Playing Behavior in Robot Soccer: A Case-Based Reasoning Approach," in *Proceedings of the International Conference on Case-Based Reasoning Research and Development*, 2007.
- [200] M. Likhachev, M. Kaess, and R. C. Arkin, "Learning behavioral parameterization using spatio-temporal case-based reasoning," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2002.
- [201] M. Beetz, F. Balint-Benczedi, N. Blodow, D. Nyga, T. Wiedemeyer, and Z.-C. Marton, "RoboSherlock: Unstructured Information Processing for Robot Perception," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Seattle, Washington, USA), 2015.
- [202] R. Murphy, S. Tadokoro, D. Nardi, A. Jacoff, P. Fiorini, H. Choset, and A. Erkmen, "Search and rescue robotics," in *Springer Handbook of Robotics* (B. Siciliano and O. Khatib, eds.), pp. 1151–1173, Springer Berlin Heidelberg, 2008.
- [203] L. Marconi, S. Leutenegger, S. Lynen, M. Burri, R. Naldi, and C. Melchiorri, "Ground and aerial robots as an aid to alpine search and rescue: Initial sherpa outcomes," in *Proceedings of the IEEE International Symposium on Safety, Security, and Rescue Robotics*, 2013.
- [204] G. De Cubber, D. Doroftei, D. Serrano, K. Chintamani, R. Sabino, and S. Ourevitch, "The eu-icarus project: Developing assistive robotic tools for search and rescue operations," in *Proceedings of the IEEE International Symposium on Safety, Security, and Rescue Robotics*, 2013.
- [205] E. Troubleyn, I. Moerman, and P. Demeester, "QoS Challenges in Wireless Sensor Networked Robotics," *Wireless Personal Communications*, vol. 70, no. 3, pp. 1059–1075, 2013.
- [206] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2011.
- [207] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the hdf5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on*



- Array Databases*, ACM, 2011.
- [208] D. Meagher, "Geometric modeling using octree encoding," *Computer Graphics and Image Processing*, vol. 19, pp. 129–147, June 1982.
- [209] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, vol. 34, no. 3, 2013.
- [210] H. Riiser, P. Vigmostad, C. Griwodz, and P. Halvorsen, "Commute path bandwidth traces from 3g networks: analysis and applications," in *Proceedings of the 4th ACM Multimedia Systems Conference*, 2013.
- [211] S. Blumenthal, E. Prassler, J. Fischer, and W. Nowak, "Towards identification of best practice algorithms in 3d perception and modeling," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2011.
- [212] M. Eich, R. Hartanto, S. Kasperski, S. Natarajan, and J. Wollenberg, "Towards coordinated multirobot missions for lunar sample collection in an unknown environment," *Journal of Field Robotics*, vol. 31, no. 1, pp. 35–74, 2014.
- [213] A. Paikan, U. Pattacini, D. Domenichelli, M. Randazzo, G. Metta, and L. Natale, "A best-effort approach for run-time channel prioritization in real-time robotic application," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2015.
- [214] A. Birk, S. Schwertfeger, and K. Pathak, "A networking framework for teleoperation in safety, security, and rescue robotics," *Wireless Communications, IEEE*, vol. 16, no. 1, pp. 6–13, 2009.
- [215] J. Kammerl, N. Blodow, R. Rusu, S. Gedikli, M. Beetz, and E. Steinbach, "Real-time compression of point cloud streams," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2012.
- [216] M. Tenorth, A. C. Perzylo, R. Lafrenz, and M. Beetz, "Representation and exchange of knowledge about actions, objects, and environments in the roboearth framework," *IEEE Transactions on Automation Science and Engineering (T-ASE)*, vol. 10, no. 3, pp. 643–651, 2013.
- [217] A. Dietrich, S. Zug, S. Mohammad, and J. Keiser, "Distributed management and representation of data and context in robotic applications," in *Proceedings of the IEEE/RSI International Conference on Intelligent Robots and Systems*, 2014.
- [218] M. Riestock, F. Engelhardt, S. Zug, and N. Hochgeschwender, "Exploring gridmap-based interfaces for the remote control of uavs under bandwidth limitations," in *Proceedings of the Companion of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*,

- 2017.
- [219] M. Beul, N. Krombach, M. Nieuwenhuisen, D. Droschel, and S. Behnke, "Autonomous navigation in a warehouse with a cognitive micro aerial vehicle," vol. 2, 2017.
- [220] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2011.
- [221] M. Fiala, "Designing highly reliable fiducial markers," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, pp. 1317–1324, July 2010.
- [222] J. L. Crowley, D. Hall, and R. Emonet, "Autonomic computer vision systems," in *Proceedings of the International Conference on Computer Vision Systems*, Springer, 2007.
- [223] C. Harris and M. Stephens, "A combined corner and edge detector," in *In Proceedings of the Fourth Alvey Vision Conference*, 1988.

## List of Figures

1.1. The general-purpose, mobile and versatile robot platforms considered in this thesis. . . . .	2
1.2. The architectural views employed for the design and development of robot perception systems. . . . .	5
2.1. The core ingredients of a modeling language. . . . .	12
3.1. The different architectural views decomposed into two main categories, namely software and hardware architecture. . . . .	27
3.2. The DSL development process composed of activities, artifacts and information flow. . . . .	29
3.3. The schematic visualization of the in-hand slip detection system. . . . .	33
3.4. The relation between variability dimensions and architectural views. . . . .	35
3.5. The design space as a combination of functional, architectural and deployment variability. . . . .	39
3.6. The visualization of the constraints corresponding to the Feature signature. . . . .	42
3.7. The Alloy model declaring components as the core architectural building blocks. . . . .	43
3.8. The Alloy model declaring perception graphs as a composition of components. . . . .	44
3.9. The Alloy model declaring the elements required to model the computational architecture. . . . .	46
3.10. The Alloy model formalizing conceptual spaces. . . . .	47
3.11. The domain model of the force-based slip detector represented in RPSL. . . . .	50
3.12. The error messages printed when interpreting the RPSL domain model. . . . .	51
4.1. The proposed design space exploration process composing activities and repositories of artifacts. . . . .	62
4.2. The Alloy model defining a design alternative. . . . .	63

4.3.	The F-Alloy excerpt showing all the mappings used to define how a design alternative is visualized. . . . .	64
4.4.	The KUKA youBot robot deployed in a factory-like environment. . . . .	65
4.5.	The functional variability expressed in RPSL. . . . .	66
4.6.	The specification of a perception graph performing a service area detection. . .	67
4.7.	The visualization of a an design alternative. . . . .	69
4.8.	The Alloy constraints used to filter design alternatives. . . . .	70
5.1.	The GPS-denied indoor environment under varying lighting conditions seen from the perspective of the quadcopter. . . . .	75
5.2.	The representation of a labeled property graph. . . . .	77
5.3.	The visualization of the labeled property graph for RPSL and DepSL. . . . .	80
5.4.	The query involving different domain models. . . . .	82
5.5.	The query retrieving platforms. . . . .	83
6.1.	The architectural views employed for deploying robot perception systems. . . .	88
6.2.	The reference architecture for deploying robot perception systems. . . . .	90
6.3.	The snapshots of the graph-based repository during the case study. . . . .	95
7.1.	The robots deployed in the factory-like environment. . . . .	102
7.2.	The component involved in implementing the task-based adaptation. . . . .	104
7.3.	The resolution model for the perception feature <code>ServiceArea</code> . . . . .	105
7.4.	The different perception graphs deployed at different points in time. . . . .	112
7.5.	The components involved in implementing the platform-based adaptation. . .	114
7.6.	The measured transmission delay for the artificial dataset. . . . .	119
7.7.	The measured transmission delay for the real-world dataset. . . . .	119
7.8.	The excerpt showing the performance of the adaptive approach. . . . .	126
A.1.	The Alloy model for the perception features. . . . .	133

## List of Tables

2.1. A summary of DSLs belonging to the Kinematics, Dynamics, Mechanism & Actuation, Sensing & Estimation, Motion Planning, Motion Control, Force Control and Reasoning Methods subdomains. . . . .	18
2.2. A summary of DSLs belonging to the Architectures & Programming subdomain.	22
3.1. The table summarizes related work from the architectural views perspective. .	54
5.1. The timing results of the graph database operations. . . . .	84
7.1. The table showing the resolutions for the task-based adaptation. . . . .	106
7.2. The table showing recording snapshots of specifications and corresponding output. . . . .	110
7.3. The costs of encoding and decoding point clouds. . . . .	117
7.4. The different perception graphs and their Octree configuration. . . . .	117
7.5. The grayvalue histograms occurred during the training phase. . . . .	123
7.6. The timing behavior for the adaptation. . . . .	124
7.7. The performance of the adaptive approach with respect to the individual configurations. . . . .	125