Modeling of Motion Primitive Architectures using Domain-Specific Languages

by
Dipl.-Ing. Arne Nordmann

Dissertation

Faculty of Technology
Bielefeld University

Bielefeld, August 2015

to the most wonderful family one can wish for

to the friends and colleages who joined me
on this crazy journey they call *doing a phd*

Printed on permanent paper according to ♾ ISO 9706.

Abstract

Many aspects in robotics, and their omnipresent ideal models, animals and humans, are still not understood or explored well enough, for example producing motions of animal- and human-like complexity. To explore the inner workings of systems studying this complexity, the essential concepts of interest need to be made explicit and raised from the code-level to a higher level of abstraction to be able to reason about them.

This work introduces a model-driven engineering approach for complex movement control architectures based on motion primitives, which in recent years have been a central development towards adaptive and flexible control of complex and compliant robots. The goal is to realize rich motor skills through the composition of motion primitives. This thesis proposes a design process to analyze the control architectures of representative example systems to identify common abstractions. Identified and formalized concepts can then be used to automate software development of motion primitive architectures through model-driven engineering methods and domain-specific languages. It turns out that the introduced notion of motion primitives implemented as dynamical systems with machine learning capabilities, provide the computational building block for a large class of such control architectures. Building on the identified concepts, a set of modularized domain-specific languages allows the compact specification of motion primitive architectures. This paves the way for domain experts rather than computing technology specialists to produce systems, which is one of the main goals of this work.

The approach and the accompanying model-driven engineering toolchain is evaluated in a task of the European Robotics Challenges (EuRoC) and a real world example of automatic laundry grasping with the KUKA Lightweight Robot IV, where executable source-code is automatically generated from the domain-specific language specification.

Contents

List of Figures vii

List of Tables ix

I. Introduction 1

1. Introduction 3
1.1. Problem Statement . . . 3
1.2. Contribution and Research Questions . . . 5
1.3. Outline . . . 7

II. Conceptual Perspective 11

2. Model-Driven Engineering in Robotics 13
2.1. Model-Driven Engineering . . . 13
2.2. Domain-Specific Languages . . . 16
2.3. State of the Art . . . 19
2.4. Design Processes . . . 21
2.5. Discussion . . . 23

3. Motion Primitive Architectures 25
3.1. Motion Primitives . . . 26
3.2. Learning Motion Primitives . . . 27
3.3. Domain Analysis and Related Work . . . 28
3.4. Discussion . . . 34

4. Model-Driven Engineering for Motion Primitive Architectures 37
4.1. Project and Domain Context . . . 37
4.2. Objectives and Requirements . . . 39
4.3. Process . . . 41
4.4. Discussion . . . 44

III. Developer Perspective **45**

5. Technology-independent Architecture and Metamodel **47**
5.1. Structural Models . . . 48
5.2. Behavioral Models . . . 57
5.3. Algorithmic Models . . . 59
5.4. Discussion . . . 60

6. Language Modularization and Design **63**
6.1. Language Workbenches . . . 64
6.2. Language Modularization . . . 65
6.3. Language Design . . . 69
6.4. Transformations . . . 79
6.5. Discussion . . . 83

7. Programming Model and Technology Mapping **85**
7.1. Programming Model . . . 86
7.2. Technology Mapping . . . 91
7.3. Mock Platforms and Vertical Prototypes . . . 93
7.4. Deployment Descriptors . . . 96
7.5. Discussion . . . 97

IV. User Perspective **101**

8. Toolchain **103**
8.1. Integrated Development Environment . . . 103
8.2. Validation . . . 105
8.3. Component Repository . . . 106
8.4. Code Generation . . . 106
8.5. Deployment . . . 111
8.6. Discussion . . . 112

9. Modeling Motion Primitive Architectures **115**
9.1. Hypothesis Test Cycle . . . 116
9.2. Discussion . . . 123

10. Evaluation and Application **125**
10.1. Qualitative Evaluation . . . 126
10.2. Quantitative Evaluation . . . 132
10.3. Discussion . . . 135

V. Conclusion **137**

11. Conclusion **139**
11.1. Summary 139
11.2. Outlook 141

References **143**

VI. Appendix **155**

A. Related References by the Author **157**

B. Domain Analysis **159**
B.1. Feature Models 159
B.2. Adaptive Module Survey 163

List of Figures

1.1. Left hand trajectories generated with a motion primitive. 4
1.2. A typical modeling approach with DSLs. 6
1.3. Outline 7

2.1. Four levels of modeling, from meta-metamodel to GPL code. 14
2.2. Language and fragment dependencies of language modularization types. 18

3.1. Example of a motion primitives on the iCub robot. 27
3.2. Context diagram of motion architectures. 30

4.1. Proposed design process 42

5.1. Quadruped robot Oncilla in simulation. 48
5.2. Main abstractions and relations of the motion primitive architecture. . 49
5.3. Object diagram of Spaces, Space Types, and Data Types 50
5.4. Specializations of Mappings and Transformations. 51
5.5. Inverse Kinematics Mapping for Foot Position and Leg Joints. 52
5.6. Composite structure diagram of an Adaptive Module. 53
5.7. Adaptive Module lifecycle 54
5.8. Adaptive Module for the walking dynamics 55
5.9. Specializations of the Adaptive Component concept. 56
5.10. Reaching Controller for placement of the left fore foot. 57
5.11. Metamodel of the behavioral aspects. 58
5.12. Metamodel of the algorithmic aspects. 59
5.13. Specializations of the Expression concept. 60
5.14. Reaching dynamics expressed in the algorithmic model 61

6.1. Parser-based approaches vs. projectional editing. 65
6.2. Language modularization and their dependencies. 66
6.3. Metamodel of the Component DSL. 67
6.4. Language composition of the Primitive Coordination DSL. 68
6.5. Language design aspects in Jetbrains MPS. 70
6.6. Language structure definition in MPS. 72
6.7. Language editor definition in MPS. 73
6.8. DSL expression of structural motion primitive architecture aspects. . . 75
6.9. Dynamical System DSL integrated into the Motion Primitive DSL. . . 75
6.10. Integration of domain concepts with the MPS base language. 77
6.11. Coordination DSL snippet with Primitive Coordination DSL extensions. 78

6.12. Model-to-model transformations. 80
6.13. Transforming an Adaptive Module to a Component. 82
6.14. Generator rule to map a Coordination DSL state to SCXML. 83

7.1. Class diagram of the programming model. 86
7.2. Mapping of a Reaching Controller to Component DSL concepts. . . . 88
7.3. Domain-specific programming model 89
7.4. Technology mapping for the structural aspects. 92
7.5. Technology mapping for the behavioral aspects 93
7.6. Oncilla roundtrip. 94
7.7. The Dynamical System in the running example replaced by the MVITE. 98
7.8. Levels of modeling in relation to the deployment descriptors. 99

8.1. Customized, MPS-based IDE themed for the AMARSi project. 104
8.2. Screenshot of the DSL IDE editor view. 105
8.3. Using models from the repository in the DSL IDE. 107
8.4. Model-to-text transformations. 108
8.5. Auto-generated visualization of the coordination of the running example.108
8.6. MSM transformation rule for the Robot Converged condition. 109
8.7. TextGen rules for C++ main file generation. 110
8.8. Toolchain and software builds on continuous integration server. 111

9.1. Hypothesis Test Cycle 117
9.2. Data Type incompatibility solved by adding a suitable Mapping. . . . 120
9.3. Space annotations of middleware scopes. 121
9.4. Space annotations of transport configuration. 122

10.1. Task 4 of the Shop Floor Logistics and Manipulation task 127
10.2. Example State using Middleware Coordination DSL extensions. 128
10.3. Example State using Primitive Coordination DSL extensions. 130
10.4. Automated laundry handling use case. 131
10.5. System visualization of the laundry handling use case 133

B.1. 160
B.2. 162

List of Tables

10.1. Total source lines of code of the system specification in the proposed domain-specific languages (DSLs) and with the targeted technology mapping. 135

B.1. Exemplary completion of the Adaptive Modules survey for the Dynamical Movement Primitive. 163

Part I.

Introduction

Chapter 1.

Introduction

"What I cannot make, I do not understand"
– Richard Feynman

This quote by physicist Richard Feynman sets the tone for this thesis. Feynman meant here that understanding something is not just about working through advanced mathematics. *"One must also have a notion that is intuitive enough to explain to an audience that cannot follow the detailed derivation."* Feynman's position was somewhat radical: Once, a colleague of Feynman said to him:

"Explain to me, so that I can understand it, why spin one-half particles obey Fermi-Dirac statistics." Feynman said, *"I'll prepare a freshman lecture on it."* However, he came back a few days later to say, *"I could not do it. I could not reduce it to the freshman level. That means we do not really understand it."*

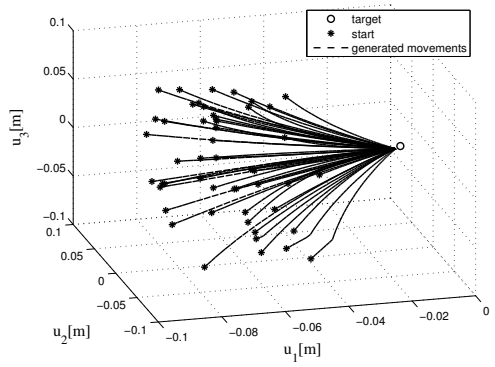
Lots of aspects in robotics, and their omnipresent ideal models, animals and humans, are still not understood or explored well enough. To explore the inner workings of such complex systems, however, producing hundreds and thousands of lines of source code, constructing the n-th prototype with isolated functionality is certainly not the answer, as it hides the essential concepts behind general-purpose language source code. Yet, this approach is still fashionable in robotics.

This thesis aims at an environment where formulation of scientific problems and their solution hypotheses is done in natural notation with domain-specific languages. Technological support then allows validation of these on real executable robotics systems without the need to read hundreds and thousands of lines of source code or write them by hand.

1.1. Problem Statement

The current state of affairs in robotics research is still to a large extent building on isolated islands of functionality that are implemented by hand. One of the problems with this kind of development is that manual implementation takes time, requires software development skills, and is inherently error-prone. This is especially hard since robotics is an inter-disciplinary field, requiring handling of multiple disciplines and technologies to create a working system. Experts of a certain discipline (i.e. "domain experts") often either need to learn the necessary software development skills, or software developers need to be trained in a certain domain, learn about the essential

(a) The iCub humanoid robot performing a motion primitive in simulation.



(b) Left hand trajectories generated with a motion primitive from various starting points to the same target point.

Figure 1.1.: Left hand trajectories generated with a motion primitive and executed on the iCub humanoid robot to execute a reaching motion.[Reinhart and Steil, 2011]

questions, concepts and methods, in order to realize an executable robotics system to test a certain scientific hypothesis.

The domain targeted by this thesis is the domain of robotics movement architectures based on so-called *motion primitives*. Motion primitives resemble building blocks of complex motions that can generalize to new situations or environments and are robust to perturbations. Motion primitives as flexible and adaptive building blocks for complex motion skills build on insights from biology that suggests that complex motions in animals and humans arise from the composition of these primitives [Bizzi et al., 2008]. Animals and humans orchestrate a vast number of muscles to move in a very flexible way and to optimize complex motions for each task. By using the representation of motion primitives, the need to generate detailed trajectories for all variables of all muscles is reduced.

Applying this theory to the field of robotics is motivated by the idea that accordingly the large number of parameters of all degrees of freedom in an advanced compliant robot and the according computational requirements can be handled through computational building blocks that mimic their biological counterparts. An example of a motion primitive reproducing a motion on a robot is shown in Fig. 1.1. A motion primitive with a trained movement shape reproduces a motion of the robot hand from varying start points to the same target point with the trained dynamics that determine the shape of the motion, e.g., for grasping. Combination, sequencing, and composition of several of such modular movement building blocks then form complex, rich motions that can be adapted to various tasks and environments.

Several research projects in the domain of motion primitives based movement generation in robotics already successfully applied this idea, combining discrete and periodic movements to, e.g., catch objects in flight [Shukla and Billard, 2011], walk a quadruped

robot on unperceived rough terrain [Ajallooeian et al., 2013], and a humanoid robot dancing [Nakaoka et al., 2004] or drumming [Degallier et al., 2006]. This domain, however, shares the overall problem of robotics research, being dominated by single, yet incompatible, and handcrafted experiments and solutions [Nordmann and Wrede, 2012]. This means that tremendous code-bases (robotic frameworks, tools, libraries, middleware systems, etc.) coexist without interoperability, which is the base hypothesis for motion primitive, though.

This calls for a unifying conceptual framework, which allows the combination of different experiments and their motion primitives to explore the design space of motion control architectures. This is challenging both because of the intrinsic complexity of the underlying control problems and due to the conceptual fragmentation of the domain.

1.2. Contribution and Research Questions

The problems discussed above indicate that research in motion primitive architectures for robotics, especially combination of motion primitives, needs support. Model-driven engineering methods [Schmidt, 2006] are known to cope with the challenges of building complex heterogeneous systems in domains such as aerospace, telecommunication and automotive [van Deursen et al., 2000], which face similarly complex integration and modeling challenges as advanced robotics. Model-driven engineering (MDE) approaches primarily focus on creating and exploiting domain models, i.e., conceptual models of the topics related to a specific problem. A model in this sense represents a particular viewpoint or perspective of a concept. It is *"an abstraction of a system often used to replace the system under study"* [Rodrigues da Silva, 2015] and is created with the purpose to reason about its properties and relationships. In recent years, several model-driven engineering approaches have been adapted to the robotics domain [Nordmann et al., 2014], yet not to the domain of motion primitive architectures.

A particularly promising method in the field of MDE is to formalize the domain knowledge with so-called *domain-specific languages* (DSLs) and thereby make it available for automating development of executable robotics systems. A domain-specific language (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 problem domain*" [van Deursen et al., 2000]. The abstractions are "*natural/suitable for the stakeholders who specify that particular concern*" [Völter et al., 2013]. The concrete syntax of a DSL enables higher understandability and efficiency because the syntax and notation is closer to the problem domain than general-purpose languages (GPLs).

Fig. 1.2 shows a typical DSL based modeling approach as it is targeted by this work. The metamodel defines the concepts and abstractions to cover the domain, such as motion primitive architectures in this work. A DSL adds a concrete, e.g., textual or graphical, syntax to the metamodel and allows formulation of domain models, i.e. a concrete system or application. The actual executable robotics system can be created

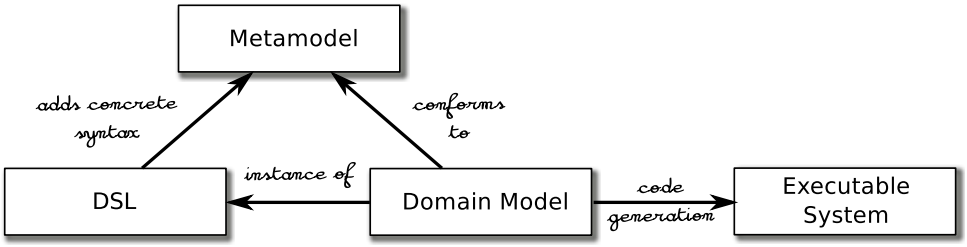


Figure 1.2.: A typical modeling approach with DSLs: The DSLs add a concrete syntax to a metamodel and are used to implement a concrete domain model. Code generation is one approach to make the domain models executable.

from the domain models, e.g., by means of code generation.

This thesis aims at a development process that lifts specification of motion primitive architectures from general-purpose language code level to a higher level of abstraction in terms of the essential concepts of this domain. If this specification is formalized, e.g., with the help of domain-specific languages, it can be technologically supported to allow validation of these architectures and their execution on real robotics systems without the need to understand general-purpose language source code or write it by hand. Therefore, the overarching **goal of this thesis** is to

> *establish a model-driven engineering process to efficiently bootstrap and exploit motion primitive architectures in order to ease and speed-up research for experts of the domain.*

To the best of the author's knowledge, no such process or DSL for motion primitive architectures in advanced robotics exists so far. The following research questions will be investigated and discussed in the course of this thesis to pursue this goal:

RQ1 Can the complex domain of motion primitive architectures consisting of advanced robotics hardware, distributed software, dynamical systems, and machine learning approaches be exploited and supported by means of model-driven engineering methods?

RQ2 Do these methods make the domain accessible for efficient development processes to ease creation and execution of motion primitive experiments on real robot systems for domain-experts?

RQ3 Can these methods be open and flexible enough to allow continued research in the domain while at the same time producing stable systems for experimentation?

RQ4 Can legacy work in terms of both, established development processes as well as existing software artifacts, be incorporated to ease and lower the risk of introducing model-driven engineering methods into robotics research?

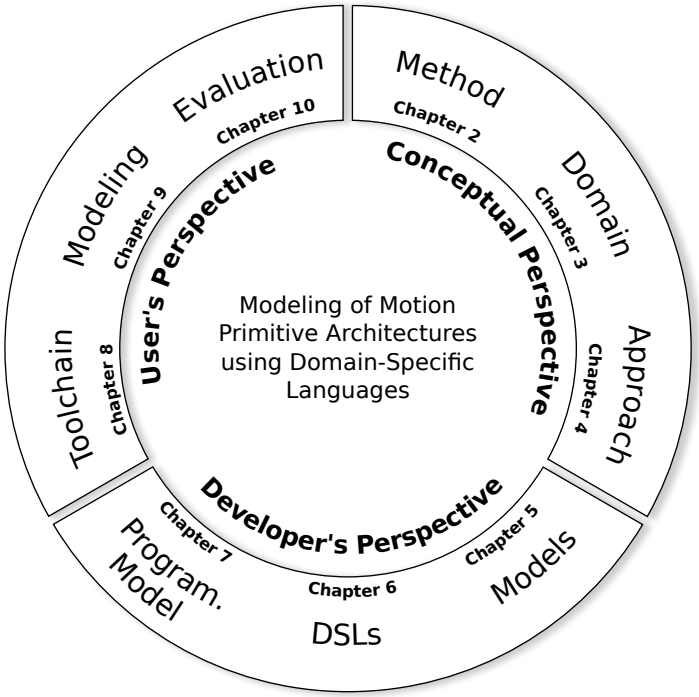


Figure 1.3.: Outline of this thesis, discussing the modeling of motion primitive architectures from a conceptual perspective, a developer's perspective, and a user's perspective.

Once this is established, domain experts can formulate their scientific hypothesis in a domain-specific language that is natural to them and restricted to their problem domain. Domain-specific model validation and verification based on the formal models can prevent errors already during specification and code generation allows execution of the motion primitive hypothesis and experiments.

1.3. Outline

The remainder of this thesis investigates and discusses the research questions from three perspectives: from conceptual point of view, from a developer's perspective, and finally from a user's perspective. A *developer* in this sense is the software developer who designs a model-driven engineering process to ease research and development for the *user*, the domain expert. Fig. 1.3 shows an overview of these three perspectives and their respective chapters.

The conceptual view in Chapter 2 – Chapter 4 discusses model-driven engineering methods and domain-specific languages, introduces the motion primitive domain in more detail, and proposes a design process to systematically set up a model-driven engineering process for motion primitive architectures:

Chapter 2 introduces the basic notation and related work of model-driven engineering in robotics. It introduces the language modularization, extension, and composition (LMEC) approach proposed by Völter [2013] to decompose a metamodel into a set of composable domain-specific languages and briefly discusses model-driven engineering approaches related to this work.

Chapter 3 introduces the basic notion of motion primitives as adaptive building blocks for complex motions and introduces the domain of flexible movement generation in robotics with motion primitives. It briefly outlines a domain analysis conducted in the motion primitive domain and provides an overview on standard methods.

Chapter 4 proposed a design process for the motion primitive domain based on the language modularization, extension, and composition approach proposed by Völter [2013]. It relates to the introduced problem formulation and discusses how the process will lead to the main two artifacts that will be detailed in the course of this thesis: i) a conceptual framework and domain-specific languages targeted to the motion primitive domain, as well as ii) a toolchain that links these concepts with executable artifacts by means of code generation.

The developer's perspective discussed in Chapter 5 – Chapter 7 details development and specification of a motion primitive architecture metamodel, creation of domain-specific languages by adding concrete syntax, and presents a programming model to homogenize software development for motion primitive architecture in a compatible manner.

Chapter 5 Starting with the developer's perspective, Chapter 5 introduces the conceptual framework and metamodel developed in this work as a result of the domain analysis and challenges introduced in Chapter 3 and Chapter 4. It details the models to describe the static structural aspects of motion primitive architectures as well as their dynamic behavioral aspects.

Chapter 6 details design dimensions of domain-specific languages and how the introduced metamodel is decomposed into a set of DSLs along the LMEC approach. It discusses the language modularization, design, and transformations of the domain-specific languages developed in this work.

Chapter 7 introduces a programming model compatible with the metamodel that provides an application-programming interface for developers to implement motion primitives and proposes a method to integrate legacy code into the proposed model-driven engineering process. It then introduces an exemplary technology that is used for development and as basis for code generators in the course of the thesis.

The user's perspective, i.e. the perspective of domain experts to be supported in their research, is discussed and evaluated in Chapter 8 – Chapter 10 based on the toolchain and the model-driven engineering process provided to the domain expert.

Chapter 8 starts with the user's perspective and discusses a toolchain developed in this work that results from the proposed design process. It is the tool exposed to the domain expert to support them with proper editors, model verification and validation, as well as code-generation to produce executable motion primitive architecture experiments.

Chapter 9 builds on this toolchain and shows how modeling of a motion primitive architecture is done in the proposed model-driven engineering process, termed *Hypothesis Test Cycle*. It discusses the different modeling steps that are conducted by different roles of the overall development process.

Chapter 10 evaluates the presented model-driven engineering process in two complex examples. A qualitative evaluation investigates the complexity of the systems that can be developed with the proposed methods in two concrete case studies and how the raised level of abstractions helps in their development. An additional quantitative evaluation compares the required DSL code and the generated source code in terms of source lines of code to investigate the gained expressiveness of the proposed DSLs.

Finally, **Chapter 11** discusses the presented results and concludes with lessons learned and future perspectives.

Part II.

Conceptual Perspective

Chapter 2.

Model-Driven Engineering in Robotics

This thesis proposes a *model-driven engineering* approach [Rodrigues da Silva, 2015, Stahl et al., 2006, Zhang and Cheng, 2006] as an answer to the challenges of the robotics domain, especially challenges of motion primitive architectures. This chapter focuses on the *methodical approach* for the development of the necessary software architecture for enabling the integration, composition and control of motion primitives and the adjacent disciplines and subdomains.

Model-driven and domain-specific development methods are recognized to cope with the challenges of building complex heterogeneous systems in domains such as aerospace, telecommunication, and automotive [van Deursen et al., 2000], which face similarly complex integration and modeling challenges as today's advanced robotics. For this reason, model-driven engineering (MDE) approaches are more and more explored and adopted by the robotics domain to cope with the huge problem space and increasing complexity of robotics systems [Schlegel et al., 2010, Nordmann et al., 2014] eventually aiming *"towards industrial-strength robotics systems"* [Schlegel et al., 2009] in terms of their maturity and robustness.

This chapter is intended to provide an overview on MDE and discuss the state of the art in robotics and related domains to provide a baseline for this work. Section 2.1 introduces important concepts, terms, and definitions of MDE. The discussion is focused on the concepts and terms that are important in the robotics domain and the context of this work, especially domain-specific languages. Section 2.3 discusses the state of the art of model-driven engineering approaches in robotics and Section 2.4 discusses systematic processes to the design MDE approaches, their models and languages.

2.1. Model-Driven Engineering

The main characteristic behind MDE is that its primary software development focus emphasizes creating and exploiting domain models rather than general-purpose language (GPL) source code. Domain models are conceptual models that represent a concept of the domain from a certain viewpoint or perspective. It allows reasoning about its properties and relationships in a system. Hence, it aims at expressing systems in abstract representations of the knowledge and activities of a particular application domain, rather than the technological aspects. A major advantage of this approach is that systems expressed in models are independent of an underlying implementation technology and closer to the problem domain than GPLs. This makes the systems

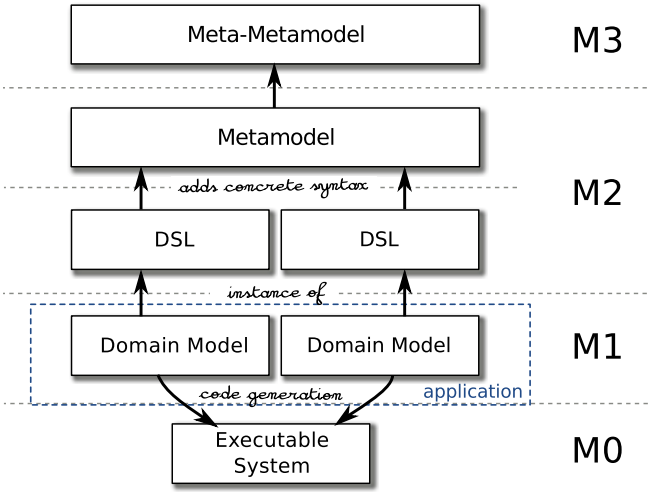


Figure 2.1.: Four levels of modeling of modeling, from metamodel to GPL code, conforming to the OMG's MDA standard [OMG, 2014].

easier to specify, understand, and maintain. It thereby paves the way for domain experts rather than computing technology specialists to produce systems, which is one of the main goals of this work.

2.1.1. Levels of Modeling

The typical levels of modeling are shown in Fig. 2.1, conforming to the Model Driven Architecture (MDA) standard of the Object Management Group (OMG) [OMG, 2014]. *M3*, the **m**eta-**m**eta**m**odel is the highest level of abstraction and provides the concepts and constraints that the metamodel and the domain-specific languages (DSLs) have to satisfy. *M2*, the **m**eta**m**odel, provides abstractions to cover the domain, in the case of this thesis the domain of motion primitive architectures. DSLs are on the same level, as they realize the metamodel by adding a concrete, e.g., textual or graphical, concrete syntax to it. *M1*, the domain **m**odels, uses *M2* to express applications or systems. Level *M0* is the actual real-world robotics systems implementation, i.e., executable GPL code that can be generated from *M1*.

One of the main advantages of MDE is that models are less sensitive to the chosen computing technology and to technological changes [Selic, 2003]. This is especially beneficial in a robotics research context, where not only the concepts but also the platforms and technologies are often still evolving and therefore changing rapidly. The concept of platform independent models (PIMs) is often closely connected to MDE. Apart from the general advantages and disadvantages of MDE, some of its features are especially relevant in the context of robotics and the targeted domain of motion primitive architectures.

2.1.2. Separation of Concerns

Ramaswamy et al. [2014b] identify several main features to model-driven engineering approaches in robotics. One of the most important in robotics is separation of concerns to manage the complexity of robotics systems, typically involving functional components of different domains such as visual perception, navigation, control, as well as various hardware and software components. Especially in research, technology neutrality and explicit modeling of the domain knowledge is important to keep models and their properties and specification independent from a particular technology. In research, the target technology is often not as clear as in industrial applications, sometimes still evolving and under construction. Targeting different technologies, even in parallel, his can be achieved by providing various transformations from a PIM to different platform specific models (PSMs) for code generation.

Since modern robotics applications, e.g., service robotics applications and applications involving physical human-robot interaction (pHRI) are often applied in unstructured environment, static and dynamic variability is another key point. Static variability denotes the *configuration* of the system and dynamic variability its *coordination* in terms of the 5Cs (Communication, Computation, Configuration, Coordination, and Composition) [Vanthienen and Bruyninckx, 2014]. Especially in unstructured environment and open-ended scenarios, models play to their full potential when they can also be used at run time, e.g., to explicitly model the variabilities and variation points during the system design to help finding the best possible solution or adapt during run time. This allows to reason on the models at run time, which, however, is not yet widely used in robotics MDE approaches [Ramaswamy et al., 2014b].

2.1.3. Viewpoints

In the robotics domain due to its intrinsic interdisciplinary nature the creation of multiple models is *"usually necessary to better represent and understand the system under study""* [Rodrigues da Silva, 2015]. There is a *"growing consensus on the need to move to comprehensive, view-based approaches"* [Atkinson and Tunjic, 2014] covering the different concerns, aspects, and disciplines, as it is done in most of the MDE approaches reviewed above. A *view* or *viewpoint* in this sense is a representation of a whole system from the perspective of a related set of concerns. The term *view* is often used for the *technical* view, defined by *IEEE STD 1471-2000* [IEE, 2000] as "a representation of a whole system from the perspective of a related set of concerns. Views are not necessarily orthogonal, but each view generally contains specific information. In MDA [OMG, 2014], a view is a collection of models that represent one aspect of an entire system." The term *viewpoint* describes the concern that is shown in a view, defined by *IEEE STD 1471-2000* [IEE, 2000] as "a specification of the conventions for constructing and using a view." It is "a form of abstraction achieved using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within a system."[ORMSC, 2001]

According to Atkinson and Tunjic [2014], it should be possible to completely repre-

sent the subject using the available views, while the number of viewpoints should be minimal and have minimal overlap. It is largely unclear what the best number of views and viewpoints is, but separating one particular concern into separate viewpoints can be useful, especially if different stakeholders specify different concerns of a domain. However, if different viewpoints are used for modeling of systems, constraints have to check for their consistency. A simple check is to validate that the target elements of references between viewpoints actually exist, since references will break if they do not. Different roles and stakeholders can the create and maintain the various fragments separately in the development process, while referential integrity is taken care of by the modeling environment. The viewpoint separation is therefore strongly aligned with the development process.

2.2. Domain-Specific Languages

To allow intuitive editing of domain models, targeting the automation of the software development and their execution, domain-specific languages additionally extract and add agreed-upon syntax and semantics from the problem domain to these models. According to van Deursen et al. [2000], 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 problem domain*". The abstractions and notations of a DSL must be "*natural/suitable for the stakeholders who specify that particular concern*" [Völter et al., 2013], e.g., by reviewing existing code examples and application-programming interfaces (APIs), through the analysis of formal descriptions found in the literature, or the application of further analysis patterns [Mernik et al., 2005]. The identified abstractions and desired notations can be realized as a language that expresses the domain models.

In contrast to GPLs such as C++, Java, or Python, DSLs usually contain only a restricted set of notations and abstractions. Compared to *external* DSLs that define their own syntax and semantics, so-called *internal* DSLs are embedded in extensible GPLs such as Python, Lua, or Ruby. They extend the syntax and potentially the semantics of the host language with domain-specific notations and abstractions. This adds the expressive power of the DSL to the GPL. While internal DSLs typically rely on (and are bound to) the execution semantics of their host language, external DSLs can be transformed to a format that directly allows execution on a target platform or interpretation, e.g., through a virtual machine.

Similarly, domain-specific modeling languages (DSMLs) that use graphical notations must be differentiated from general purpose modeling languages (GPMLs) such as Unified Modeling Language (UML) or Systems Modeling Language (SysML). While it is still possible to add domain-specific abstractions to these languages, e.g., using UML Profiles (cf. Modeling and Analysis of Real Time and Embedded systems (MARTE) [Gerard and Selic, 2008] that extends UML with abstractions to describe and analyze real-time systems), adding domain-specific notation to graphical modeling languages is much harder. GPMLs typically provide a larger number of generic

constructs and notation, which allows their application in different domains, e.g., the modeling of object-oriented software systems in UML. In contrast, DSMLs (as well as textual DSMLs) are typically comprised of a smaller set of concepts and graphical notations that are close to the respective application domain [Rodrigues da Silva, 2015]. A common practice for the definition of GPMLs is the use of the UML Profile mechanism that allows adding domain-specific abstractions to UML, e.g., MARTE [Gerard and Selic, 2008] for modeling and analyzing real-time systems.

In order to efficiently implement and apply a DSL approach for the development of robotics systems and to fully exploit its benefits, DS(M)Ls are typically realized in toolchains tailored to model-driven development such as the Eclipse Modeling Project (EMP) [Gronback, 2009]. These so-called *language workbenches* offer extensive support for the development of the DSLs themselves and for the actual system modeling tasks performed by a language user. DSLs developed in these environments facilitate the users modeling tasks typically with textual and/or graphical editors with rich code completion and dynamic constraint checking. Furthermore, these environments typically provide model-to-model transformations (M2Ms) and model-to-text transformations (M2Ts) in order to generate code from system models that integrates with the overall environment used for the development.

One of the main difference between metamodels and DSLs is that they add concrete syntax and a natural notation to them, e.g., textual syntax, tabular syntax, mathematical expressions, or a graphical notation. This allows targeting a language to domain experts by providing them with their natural notation and reasonable defaults, easing the overall development. Two fundamental characteristics of well-designed DSLs are therefore their expressive power targeted to a specific domain and the definition of formal notations intuitively understandable for domain experts while being machine processable, eventually yielding executable applications.

Advantages of using DSLs in development are that they *"can reduce the costs related to maintaining software."* [Deursen and Klint, 1998] They enable users without software engineering experience to create programs as long as they possess knowledge of the targeted domain. On top of that, DSLs provide the *"ability to generate more verification on the syntax and semantics than a general modeling language does."* [Nguyen et al., 2014] A disadvantage of DSLs is a long learning curve for a new language, even though as it is a domain-specific language, it would be a lot easier to learn than a general programming language.[Nguyen et al., 2014]

2.2.1. Language Modularization

In the same way, modularization of software in object-oriented programming (OOP) makes software development more efficient and software more reusable, being able to reuse domain-specific languages or parts of them in new contexts makes designing DSLs more efficient. This is especially relevant in multi-disciplinary fields such as robotics. Language modularization allows defining separate concern-specific DSLs, each addressing different concern of the domain. A program specified in these DSLs then consists of concern-specific fragments. This approach promotes separation of

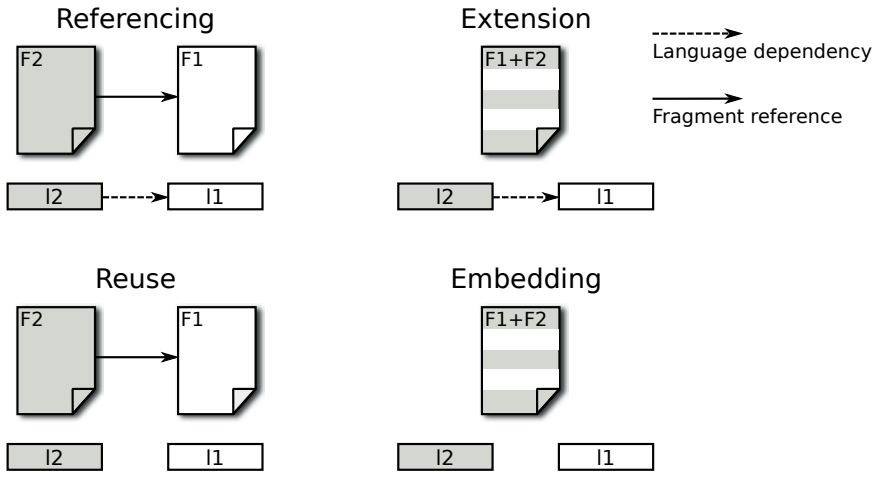


Figure 2.2.: Language and fragment dependencies in the different types of language modularization [Völter et al., 2013, Section 4.6].

concerns and language reuse. Domain experts and stakeholders can work on aspects they are experts in, or particularly interested in, independent of others. It also allows independent reuse of the language fragments as well as the languages themselves. This avoids that new DSLs are developed for new problems, each covering several aspects of the targeted domain, but being incompatible with other languages that also cover similar aspects.

Völter et al. [2013] discuss an approach termed *language modularization, extension, and composition* (LMEC) for language modularization that allows the design and composition of the syntax, constraints, editors, and code generators of different languages. Völter et al. [2013] distinguishes four different ways of composing languages. Fig. 2.2 shows how languages and fragments of these languages depend on each other. The document symbols represent DSL fragments, i.e. programs represented in the DSL; the boxes represent DSLs. Dashed lines show language dependencies, solid lines show references between fragments. White and gray elements indicate the languages used in the fragments:

Referencing A *referencing* language ($l2$) depends on the *referenced* language ($l1$) when at least one concept of language $l2$ references a concept from $l1$. This introduces a direct dependency between these two languages, in that language $l2$ cannot be used without language $l1$.

Extension Language extension enables mixing of concepts from two or more different languages. The *extending* language ($l2$) adds additional language concepts to the *extended* language ($l1$). This also introduces a direct dependency between these two languages, in that language $l2$ depends on language $l1$.

Reuse Language reuse is a combination of referencing and extension. An *adapter language* l_a is created that extends a language $l1$ with concepts that reference

concepts of a language $l2$ and thereby allows using both languages $l1$ and $l2$ without introducing a direct dependency between these two. This still allows using $l1$ and $l2$ independent of each other in different context, when the adapter language l_a is not used. In this context, $l2$ is the *context* language and $l1$ is the *reused* language.

Embedding Language embedding is similar to reuse, but instead of establishing references between concepts of the two languages $l1$ and $l2$, instances of concepts from both language $l2$ are embedded into the same fragment. This also needs an additional adapter language l_a. In this case, $l1$ is the host language that *embeds* the language $l2$.

While referencing and extension introduce direct dependencies between languages, reuse and embedding use so called *adapter languages* to make them usable together, but does not introduce direct dependencies. This means, both languages can also be used independent of each other. In the same way as for software components a smaller dependency footprint eases reuse, the latter two types of modularization are preferred in this thesis for languages of different concerns, as discussed in more detail for the motion primitive architecture domain in Chapter 6.

2.3. State of the Art

In the last years, various model-driven engineering approaches were actively adapted to the robotics domain to handle the complexity of robotics systems and help with the separation of concerns regarding the *functional architecture* and *software architecture*. Motivated from the positive results from the application of MDE in other domains such as automotive and avionics, robotics software engineering is gradually moving in that direction.[Schlegel et al., 2009, Ramaswamy et al., 2014b, Nordmann et al., 2014] A common goal of these approaches is to support the development and ease design space exploration. This requires supporting the entire experimental toolchain ranging from purely functional modeling to software architectural and technical aspects such as software deployment. This section does not and cannot cover the huge body of related work, but rather highlights some relevant related MDE approaches in robotics research and discusses how their design choices, commonalities, and differences relate to the aspects introduced above. A more complete discussion on the state of the art for MDE and DSLs approaches in robotics can be found in [Nordmann et al., 2014].

Most of the approaches emphasize the separation of concerns and employ code generation to map the models to executable systems. One of the most prominent examples, being featured in the European research project *BRICS* ("Best Practice in Robotics") [Bischoff et al., 2010], is the BRICS Component Model (BCM). Bruyninckx et al. [2013] introduce a Component-Port-Connector (CPC) based model, which is mapped to the Communication, Computation, Configuration, Coordination, and Composition (5Cs) semantics [Vanthienen and Bruyninckx, 2014] and integrated with

an Eclipse-based integrated development environment (IDE). Modeling in BRICS follows three steps: i) modeling the structural aspects in the CPC model, ii) adding coordination aspects to the components using state machines, and iii) generating code that targets Orocos [Bruyninckx, 2001] and Robot Operating System (ROS) [Quigley et al., 2009] platforms. The first two steps belong to the PIM, the third step being the PSM.

While BCM distinguishes the PIM into two models, the structural and the coordination aspects, V^{3}CMM [Alonso et al., 2010] provides a metamodel for structural, behavioral, and algorithmic views. The structural model describes the static structure of the components, the coordination model describes the event driven behavior of the components and the algorithmic model specifies the algorithm executed by the components based on their current state. The structural view also maps to a Component-Port-Connector model, the coordination model reuses UML state charts, and the algorithmic models rely on UML activity diagrams.

Another approach with strong focus on separation of concerns, also using UML as base technology, is *SmartSoft* [Schlegel et al., 2009, 2015, Steck and Schlegel, 2010]. It is a MDE approach focused on service-oriented architectures for service robotics. It uses a DSML that is defined as a UML profile and separates middleware aspects from algorithmic aspects. Component skeletons (termed *component hulls*) mediate the external visible services of a component and internal component implementation. A component hull provides links to four artifacts: i) internal user code, ii) communication to external components, iii) platform-independent concepts (threads, synchronization, etc.), and iv) platform-specific middleware and operating system. It is implemented in an EMP based toolchain.

All the mentioned approaches emphasize their separation of concerns, which is typically separated into structural, coordination, and optionally computation views. From the discussed approaches, V^{3}CMM has a particular strong focus on viewpoint separation, providing only loose coupling among different viewpoints. It has, however, only a uni-directional relationship between its structural, behavioral, and algorithmic views, so that these have to be manually corrected if one of the viewpoints changes [Ramaswamy et al., 2014b].

BCM and *SmartSoft*, amongst others, clearly separate platform independent model and platform specific model, e.g., to keep the functional modeling independent from the actual target platform. This is opposed to the *Robot Construction Kit* (Rock) [Reichardt et al., 2012, Reichardt and Berns, 2013] for example, which clearly states its dependency to Orocos as a platform and targets easier development for Orocos. Its tool *oroGen* supports specification and code generation for three artifacts: i) toolkits, ii) task contexts, and optionally iii) the static deployment of components. It does not aim for platform-independence, but rather supports specification of systems and applications for Orocos in an easy and reliable manner.

Another MDE example closely related to the domain targeted with this thesis is *MiRPA* ("Middleware for Robotic and Process Control Applications") [Kröger et al., 2004, Thomas et al., 2003, Finkemeyer et al., 2007]. MiRPA is an architectural approach centered on so-called *manipulation primitives* (MaP), which are small primitive

tasks that can be combined to complex tasks. MaPs are the *expert mode*, formulated in the task-frame formalism (TFF) [Kröger et al., 2004]. They are supposed to be combined to more complex tasks (*skills*) by non-experts in the form of directed graphs of MaPs (*primitive nets*). A Matlab/Simulink interface allows models from within Matlab to be executed as MiRPA-modules. Interpretation and execution of primitive nets generates setpoints for the respective robot controllers.

2.3.1. Domain-Specific Languages

A significant number of MDE approaches in robotics facilitate DSLs as shown in a survey on DSLs in robotics by Nordmann et al. [2014]. Those approaches have an inherently strong focus on separation of concerns; in fact, most of the DSLs are dedicated to a specific concern. A recent example is the *SafeRobots* framework ("Self Adaptive Framework for Robotic Systems") [Ramaswamy et al., 2014a] for developing software for robotic systems. The first phase of the SafeRobots development process is modeling of the domain knowledge in terms of ontologies, DSLs, or knowledge graphs, before these are used for modeling of an actual problem, its solution, and finally generating GPL code for execution. Ramaswamy et al. [2014a] show two examples with i) a system integration and knowledge representation problem, and ii) the issues associated with robotic system development in an industrial scenario.

RobotML ("Robot Modeling Language") introduced by Dhouib et al. [2012] is an example explicitly following a language modularization approach. *RobotML* is from the European PROTEUS project, which covers design, simulation, and deployments aspects of robotics applications and is based on an ontology. It is separated in what they call "packages", separate platform-independent languages for structural, behavioral and communication aspects, as well as platform-specific languages for deployment. Schlegel et al. [2015] integrate two language modules [Steck and Schlegel, 2011, Lotz et al., 2013] for different runtime variability mechanism in their *SmartSoft* development environment.

These are also two of only a few examples that exploit models not only for generating executable artifacts, but also at run time, e.g., to allow reasoning based on these models. Steck and Schlegel [2011], Lotz et al. [2013] explicitly model run time variability at design time to allow the robot to access these models, e.g., for resources management and reason about different execution variants. Most MDE approaches in robotics, however, target and support execution of their models via code generation [Nordmann et al., 2014], e.g., [Dhouib et al., 2012, Schlegel et al., 2015, Ramaswamy et al., 2014a, Reichardt et al., 2012, Bruyninckx et al., 2013]. Artifact generation from DSLs becomes especially powerful and suited for reuse if the toolchain supports different M2Ms and M2Ts transformations. Either to generate different artifacts like visualization, computational routines and glue code [Nordmann and Wrede, 2012], or executable code for different programming languages or software platforms [Frigerio et al., 2013, Klotzbücher et al., 2011, Dhouib et al., 2012].

2.4. Design Processes

While several MDE approaches in robotics propose development approaches targeting a similar level of development support, only little is written on the systematic design processes that lead to the proposed models and languages [Schneider et al., 2014, Nordmann et al., 2014]. One reason may be that the development of MDE and DSL based approaches is often performed in an ad-hoc and implicit manner. Mernik et al. [2005] discuss, however, that the identification and formalization of domain-specific abstractions is an important decision pattern for DSL development. This is especially important in the robotics domain, which contains a large-body of knowledge found and developed outside robotics. In the course of this thesis, the term *design process* is used to denote the *systematic process of developing and designing models, DSLs, and accompanying toolchain and establishing a model-driven engineering process* to distinguish it from the model-driven and DSL-based *development process* that is the outcome of the design process.

Schneider et al. [2014] provide one of the most recent and most explicit discussions of the design process of a DSL. They present and discuss a design process which they reverse-engineered from the development of a concrete DSL for the specification of grasping problems. Their design process distinguishes different roles of involved persons and consists of several development phases. It starts with extraction and analysis of domain knowledge and subdomains based on use cases, formalization of this knowledge, development of tool support based on this formalization, and finally modeling of use cases to validate the developed concepts and potentially iterate the process. Dhouib et al. [2012] base their DSML on a robotics ontology to reuse the incorporated expert knowledge in their DSL domain models. Laet et al. [2012c] discuss how a thorough definition of the semantics of geometric relations [Laet et al., 2012a,b] is formalized into an external DSL and an internal Prolog DSL. Ramaswamy et al. [2014a] do not distinguish between the design process and the resulting MDE development process, but consider the "General Domain Knowledge Modeling" that comprises modeling of the domain knowledge with the help of e.g., ontologies, DSLs, etc., as first part of their overall development process.

Some of the UML based approaches relate themselves to the MDA OMG [2014] approach as defined by the OMG, which provides a standard for MDE. Its primary focus is on platform independence by explicitly separating into platform independent models and platform specific models. The core of the MDA concepts comprises the UML, eXtensible Markup Language (XML) as well as the related XML Metadata Interchange (XMI) specification, SOAP and other OMG standards. MDA defines an approach to system specifications that separates the specification of the system functionality from the specification of the platform specific implementation. UML and XML are widely used and therefore open this approach to a large amount of tool support, e.g., editing, visualization, code generation, and reverse engineering. An issue with UML is that it is often too generic and vague to be efficient, missing more abstract or specialized concepts. This issue can, however, be resolved by creating domain-specific subsets of UML using so-called UML profiles. These allow tightening

up the UML semantics and adding additional validation to it. The basic structure, however, is still largely defined by UML, e.g. it is hard to remove parts of UML that are not relevant or need to be restricted in a specialized language.[Dalgarno and Fowler, 2008]

Völter [2005] proposes a different approach to software architecture that targets automation of many software development aspects by using domain-specific languages instead of UML or UML profiles respectively. He proposes three phases of his architectural process: First, in the elaboration phase, a technology-independent architecture is defined, realized in a programming model and concrete implementation (technology mapping). It is tested with a mock-platform (e.g. a simulator) and then with a full vertical prototype (e.g. the robot hardware). The second iteration phase updates results of the elaboration phase, especially the technology mapping and vertical prototype, to incorporate lessons learned in the first phase and correct mistakes. Finally, when the iteration phase produced a good result, a third automation phase sets of tools and support to automate the architectural process, e.g., through DSLs and code generation.

2.5. Discussion

MDE methods are recognized to cope with the challenges of building complex heterogeneous systems, such as motion primitive architectures for robotics systems as targeted by this thesis. Its main characteristic, focus on creating conceptual domain models rather than general-purpose language source code, suits the context of this work, where users might reside in different disciplines without software engineering background.

DSLs and explicit models allow eased communication about challenges of the domain, early technical integration, quantitative as well as qualitative validation and later also automation of engineering tasks to ease experimentation and scientific analysis. DSLs make these models conveniently editable by adding syntax, natural notation, and reasonable defaults. Language modularization allows to define separate concern-specific DSLs, each addressing different concern of the domain to allow reuse on a language level, as it is successfully done on a software-level, e.g., in OOP, for many years. As this modularization makes the separation of concerns explicit, it seems to be especially suitable for the robotics domain in general and the motion primitive architecture domain with its several included and adjacent subdomains, as shown in Chapter 3.

Several of the approaches discussed in this chapter were developed or published in the last years, in parallel to the work presented in this thesis. While this raises the question of potential reuse [Nordmann et al., 2014] of the approaches, it can also serve as an indicator for the need of model-driven engineering approaches in robotics as the complexity of today's advanced robotics systems surpasses the complexity that can be handled with classical software development approaches. The MDE approaches discussed in this chapter can serve to extract best practice in the domain and serve as

a baseline to discuss the proposed approach.

While only a few approaches detail the design process that led to their models, languages and development process, Chapter 4 introduces an explicit and systematic process of applying the introduced model-driven engineering methods in the targeted domain of motion primitive architecture.

Chapter 3.

Motion Primitive Architectures

This thesis and the proposed model-driven engineering (MDE) approach target the domain of flexible movement generation in robotics based on motion primitives. This chapter provides the basic problem formulation and introduces the basic notion of motion primitives that resemble adaptive building blocks for complex motions. It turns out that the domain of motion primitive architectures is a complex domain encompassing and bordering multiple different subdomains, to which related MDE and domain-specific language (DSL) approaches are discussed.

One of the most basic and most important abilities of a robot is its ability to move, either to navigate through its environment or to interact with it. This should be done in a flexible and adaptive manner to be able to move and react in various, cluttered and unforeseen environments. The *embodiment* position in cognitive science emphasizes the role that the body and movement plays in higher level capabilities, even stating that sensorimotor skills are necessary for all higher levels, e.g., reasoning and intelligence. Typically, though, robots are equipped with a set of pre-defined movements and movement capabilities, for example point-to-point (`PTP`) movements in joint space, and linear (`LIN`), circular (`CIRC`) or spline-shaped (`SPL`) task space movements. This is especially true for industrial robotics, where precision, speed, and accuracy are important, usually more important than flexibility and adaptation.

Combinations of these movement capabilities allow creating precise movements that can be repeated for thousands of cycles with high precision and high repeat precision. While this serves the *classical* automation with large product quantities, new robotics and automation disciplines are on the rise that call for more flexibility and adaptive movement generation. Service robotics and flexible automation are two prominent new and future application domains for robots that ask for more flexibility.

One of the most revolutionary and challenging features of the next generation of robots will be their capability for physical human-robot interaction (pHRI). Robots targeting at pHRI will be designed to coexist and cooperate with humans in applications such as assisted industrial manipulation, collaborative assembly, domestic work, entertainment, rehabilitation, or medical applications. The robot application domain more and more extend from factories to human environments, due to the aging society in industrialized countries, automating more common daily tasks, and high cost of human expertise. Especially in pHRI scenarios, it is desirable to have robots, which do not only move accurately along pre-programmed paths, but show motion behavior similar to humans in order to enhance the predictability and acceptance of the robot

to the human interaction partner [Oztop et al., 2004, Chaminade et al., 2005].

These challenges require a repertoire of flexible motions and behavioral abilities, which can cope with the complexity of the real world. Recent research in robotics aims to extend their motion generation capabilities to the biological richness of humans and animals. One of the research paths to achieve this is complex movement generation based on combination and composition of so-called motion primitives.

In robotics, motions are typically represented either in joint space or in Cartesian space, i.e. the hand or end effector coordinate systems. The concept of primitives exists in both spaces, called motor primitive and movement primitives respectively [Mussa-Ivaldi et al., 1994]. In the course of the thesis, the term *motion primitives* will encompass both of these types.

3.1. Motion Primitives

To make a robot perform a desired movement, all degrees of freedoms (DoFs) (all actuators) need to be supplied with appropriate motor commands. The commands must be coordinated to fulfill the desired task, are within the range of the capabilities of the movement system, and comply with external constraints, e.g., obstacles and other constraints from the environment. Due to the number of DoFs in modern robots, e.g., humanoid robots, a large number of possible movement plans have to be considered for every task. This large variety of movements renders it hard to make them available at run time or to learn them.

The idea of motion primitives [Degallier, 2000] restricts this problem to a set of combinable building blocks, typically consisting of autonomous dynamical systems that can generate either discrete (point-to-point) or periodic movements. This approach enables expressing and learning complex movements by adjusting a relatively small set of parameters, both of the motion primitives and their composition.

In context of robotic systems, the representation of motion primitives is typically given in form of dynamical systems [Schaal et al., 2005, Kajita and Espiau, 2008]. Either non-autonomous dynamical systems:

$$\dot{\mathbf{u}} = g(\alpha, \mathbf{u}, t) \tag{3.1}$$

where $\mathbf{u}$ is a state vector, t is the time parameter and α gives a modulation parameter, which can be used to modulate the speed of the system, or autonomous dynamical systems:

$$\dot{\mathbf{u}} = g(\alpha, \mathbf{u}) \tag{3.2}$$

where no explicit time dependency exists.

Dynamical systems are primarily used for their convenient features regarding flexibility and robustness. They appear to be one of the most promising candidates as computational basis for exploitation of flexible motion capabilities featured by modern humanoid robots [Luksch et al., 2012, Kajita and Espiau, 2008]. Autonomous

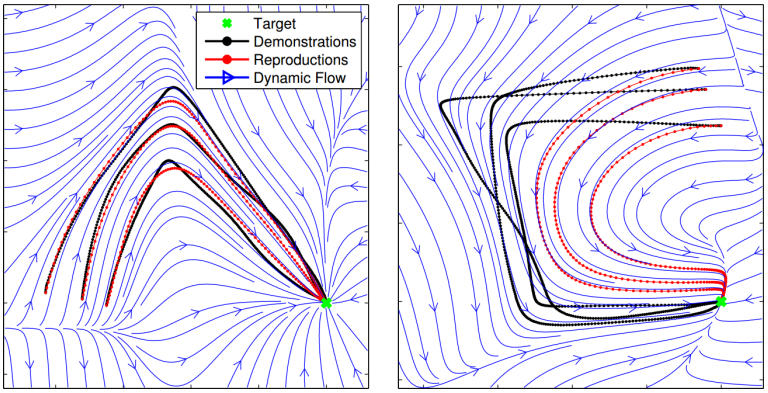


Figure 3.1.: Example of a motion primitive reproducing an A-shape (left) and C-shape (right) from different start configurations, learned from demonstration [Neumann et al., 2013].

dynamical systems can model goal-directed motion primitives to generate motions in a variety of manipulation tasks.

An example of a motion primitive reproducing movements of different shapes and from flexible start points to the same target point is shown in Fig. 3.1. A motion primitive reproduces a trajectory (red, dotted) from flexible start points to the same target point (green cross), e.g., to move an end effector. The dynamics of the motion primitive, determining the shape of the motion, can be trained and adapted to the context, for example the task or the environment, without losing the desired features of the dynamical systems. In the example shown in Fig. 3.1, the dynamical system is trained in an Extreme Learning Machine (ELM) [Huang et al., 2006] based on demonstrations (black, dotted) [Neumann et al., 2013].

3.2. Learning Motion Primitives

The flexibility to adapt motion primitives to different situations and environments is accomplished by means of machine learning techniques that can then fine-tune certain additional parameters to improve the movement.[Schaal et al., 2005]

Several approaches have been proposed in the last years to represent motion primitives for computation. A widely known recognized approach is called Dynamical Movement Primitives (DMPs) [Ijspeert et al., 2002, 2013, Schaal, 2006], which is a technique to generate motions with non-autonomous dynamical systems modeling spring damper systems. The original DMP approach uses the following notation [Ijspeert et al., 2013]:

$$\tau \ddot{\mathbf{u}} = \alpha_u(\beta_u(\mathbf{g} - \mathbf{u})\dot{\mathbf{u}}) + \mathbf{f}(s)(\mathbf{g} - \mathbf{u}_0)s, \tag{3.3}$$

coupled with the canonical system:

$$\tau \dot{s} = -\alpha_s s, \tag{3.4}$$

where α_u, β_u are stiffness and damping constants. The stability of this dynamical system is ensured in case that the perturbation $\mathbf{f}$ becomes zero at the end of the movement, which results in linear convergence to the goal point $\mathbf{g}$.

Having a robotics expert programming all the necessary motion primitives is costly and time consuming, especially when programming has to be repeated every time the task conditions changes, and often not feasible at all due to the large number of DoFs and tasks. An alternative approach to prepare a robotic system for given tasks is programming-by-demonstration (PbD) [Kajita and Espiau, 2008] that allows to teach motions relevant for a task by analyzing recorded trajectory data and generalize from them to new situations. A tutor demonstrates the task either with their own body (observed by a tracking system) or in physical interaction with the robot itself i.e. kinesthetic teaching [Argall et al., 2009, Lee and Ott, 2011, Nordmann et al., 2015].

It is an established paradigm of recent years to use motion primitives to learn motion skills [Pastor et al., 2009, Mühlig et al., 2012, Reinhart and Steil, 2012, Phung et al., 2011], for example to get a library of motion primitives to compose complex motions [Lemme, 2015, Mussa-Ivaldi and Bizzi, 2000].

3.3. Domain Analysis and Related Work

Control and learning architectures based on motion primitives is considered as domain of this thesis that should to be supported by MDE and DSLs. The following section describes the domain analysis conducted within this work to find out what the targeted model-driven approach and the targeted DSLs should be able to support and what relevant abstractions and notations they should be able to express. This is one of the key issues in the development of DSLs. The core problem is to not only understand one problem, but to understand a class of problems and its solutions [Völter et al., 2013].

In order to do so, a feature-oriented domain analysis (FODA) [Kang et al., 1990] is conducted on compliant robotics control among common robot control frameworks, interfaces of compliant robots and their applications. The FODA is a formal domain analysis method developed that introduced features models and feature modeling. A *feature* in this context is defined as a *"prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system"* [Kang et al., 1990].

The resulting feature models, cf. Section B.1, are accompanied with a survey that was conducted together with the research partners of the AMARSi project[1], in which the survey was conducted, covering state of the art motion architectures based on motion primitives [Nordmann and Wrede, 2012]. The domain analysis covered the following architectures and their respective experiments and demonstrators:

Quadruped walking over unperceived rough terrain [Ajallooeian et al., 2013] A control architecture for walking and re-balancing the overall pose of the robot. The architecture is hierarchical and modular and couples together four basic mechanisms:

[1] AMARSi is a large-scale European integration project; see: https://www.amarsi-project.eu/

i) a network of coupled central pattern generators (CPGs) (one per end-effector) to generate a periodic gait, ii) a reflex mechanism that modulates the shape of the target trajectories emitted by the CPG system if a leg hits an obstacle, iii) a proportional feedback controller for making the 12 joint angles track the output of the reflex-modulated CPG system, and iv) a higher-level, model-based control loop that stabilizes this pose when perturbed.

Catching objects in flight [Shukla and Billard, 2011] Catching objects with a catching point not located at the center of mass and highly non-linear dynamics, e.g., a tennis racket or a half-filled water bottle. This requires coordination between the arm reaching motion toward the predicted catching location and the hand/finger pose preparation for the actual catching, which is done by coupling two dynamical systems that have been trained individually using the coupled dynamical systems method.

Mixture of controllers to learn inverse kinematics [Waegeman et al., 2013] Control architecture inspired by the MOSAIC control architecture [Haruno et al., 2001], but different controllers specialize not on different tasks, but on different regions in joint space.

Redundancy learning [Nordmann et al., 2012a] An approach utilizing the physical interaction capabilities of compliant robots with data-driven and model-free machine learning to allow fast (re)configuration of redundant robots in kinesthetic teaching. This approach facilitates a hybrid controller to join machine learning capabilities with analytical control.

Humanoid upper body control [Reinhart and Steil, 2012] Control architecture for the iCub for three different bi-manual motion skills trained in physical human-robot interaction. Convergence of a discrete primitive triggers the execution of subsequent primitives.

The analysis covered a wide range of point-to-point [Ajallooeian et al., 2013, Shukla and Billard, 2011, Reinhart and Steil, 2012] and periodic movements [Ajallooeian et al., 2013, Reinhart and Steil, 2012] modeled by autonomous dynamical systems and assessed functional and non-functional properties of their diverse implementations. Functional and non-functional properties were surveyed to assess the motion primitives itself as well as their availability for the targeted development process. *Functional* properties in this survey covered parametrization, data representation, learning algorithms (e.g., online vs. offline learning, supervised vs. unsupervised learning), the required sensor feedback, etc. *Non-functional* properties covered implementation language, technical data representation, timing, software dependencies and availability, etc. An example of the survey results for the DMP is shown in the appendix in Table B.1.

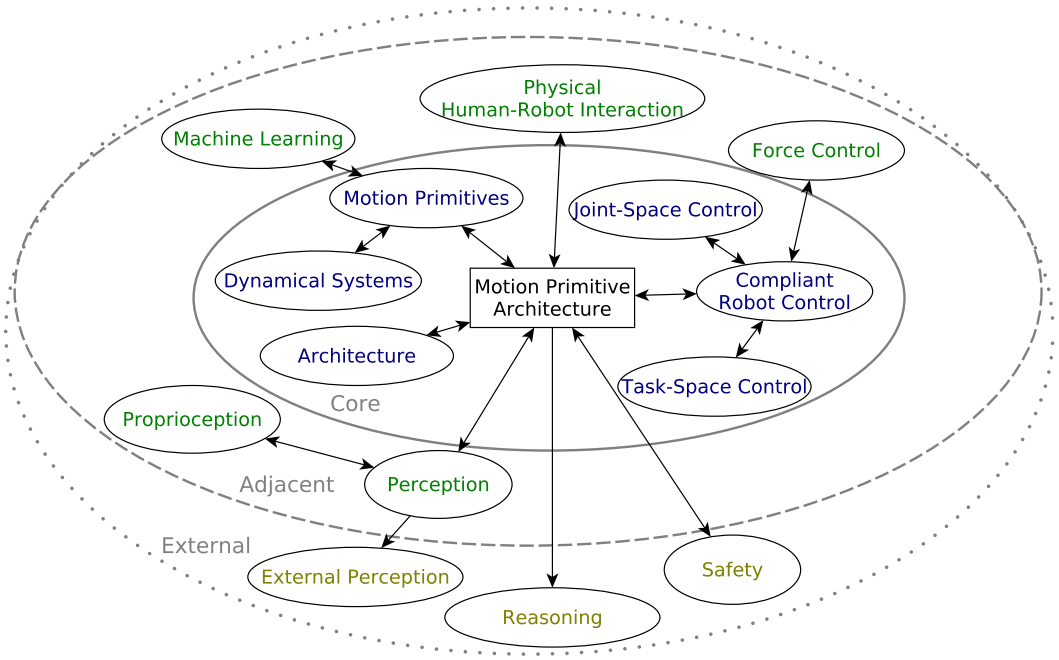


Figure 3.2.: Context diagram of motion architectures. Subdomains (ellipses) are divided into core (blue), adjacent (green), and external (yellow) subdomains.

Related work for the domain of motion primitive architectures encompasses several different sub-domains that all must play together in order to create an entire executable robotics system as targeted by this thesis. Fig. 3.2 gives an overview on the subdomains deemed relevant for motion primitive based movement architectures. This is restricted to the motor domain and does not aim at a general cognitive architecture. For the conceptual architecture, this restriction translates into excluding a large number of important aspects of a cognitive architecture like, e.g., visual perception, attention, language, reasoning, and other higher-level processes. In Fig. 3.2 subdomains are divided into subdomains that are considered belonging to the core of the domain (blue, solid), adjacent subdomains (green, dashed), and external subdomains (yellow, dotted).

In the following the subdomains relevant for motion primitive architectures are discussed, each with related MDE and DSL approaches, if available Nordmann et al. [2014]. The corresponding chapters in the handbook of robotics Siciliano and Khatib [2008] are annotated to the subdomains to relate them to generally acknowledged reference.

3.3.1. Core Subdomains

Core subdomains are the main concerns, to which this work is dedicated. This means that the approach of this thesis intends to allow concise and easy specification of these

concerns.

Architecture

(cf. Part A Chapter 8, *Robotic Systems Architectures and Programming* [Siciliano and Khatib, 2008])

The main concerns of this subdomain are architectural building blocks and abstractions, connection patterns, hierarchies, coordination and composition. Well-covered by roughly every second DSLs surveyed by [Nordmann et al., 2014] is the subdomain of task-level coordination [Biggs et al., 2010]. DSLs of this subdomain are often used for code generation and generate state chart or state machine like software artifacts. Angerer et al. [2012], Nordmann et al. [2015], Thomas et al. [2013], Klotzbücher et al. [2011] for example generate code to realize motion tasks as state hierarchies, state transitions, and import or extension of existing states. Steck and Schlegel [2010] generate constraints for a generic constraint solver from domain-specific models.

As the approach proposed in this work targets support for executable, runnable systems, its mapping to software is also a main core concern that Siciliano and Khatib [2008] assign to the architecture and programming subdomain. Software architectures are necessary to execute motion primitives based systems and run them in a reproducible way. Ringert et al. [2015] discuss advantages of Component-Port-Connector (CPC) based architecture description languages (ADLs) that combine MDE and component-based software engineering (CBSE) to reduce the conceptual gap between models and software. They already proved useful in the automotive, avionics, and robotics domain. Existing and established ADLs like the Architecture Analysis and Design Language (AADL), the Systems Modeling Language (SysML), the Unified Modeling Language (UML), and Modeling and Analysis of Real Time and Embedded systems (MARTE) allow specification of technical architectures and related aspects, e.g., in terms of (software and hardware) building blocks, CPC, composition, communication and middleware concerns. They specify architectures on a generic level and provide means for domain-specific extensions, such as UML profiles to extend UML.

Several recent robotics DSL approaches provide explicit support for software architecture concerns in robotics, e.g., by mapping their functional models to concrete technical artifacts [Baillie et al., 2010, Romero-Garcés et al., 2013, Nordmann et al., 2012a, Kilgo et al., 2012, Dhouib et al., 2012, Brugali et al., 2012].

The architecture subdomain is related to the safety subdomain, e.g., in terms of real-time capabilities and quality of service.

Motion primitives

(cf. Part G Chapter 59, *Robot Programming by Demonstration* [Siciliano and Khatib, 2008])

This subdomain focuses on representation of small building blocks of motion. Its abstractions are motion building blocks with adaptability, flexibility and therefore have a strong link to machine learning subdomain. However, no DSLs targeted to motion primitives in terms of dynamical system with machine learning extensions were surveyed by Nordmann et al. [2014] or are known to the author to this date. Close

to the concepts is the *MiRPA* approach that uses so-called manipulation primitives, which are small primitive tasks that can be combined to complex tasks, yet, without machine learning capabilities. They are combined to more complex tasks (skills) in the form of primitive nets [Kröger et al., 2004, Thomas et al., 2003, Finkemeyer et al., 2007].

Dynamical Systems This thesis focuses on motion primitives represented by dynamical systems, which generate the dynamics of the motion. The author is not aware of DSLs directly targeting this subdomain. A reason for this might be that notations and expressions of this subdomain are already well covered by mathematical expressions, therefore by several programming and modeling environments, e.g., MATLAB and Mathematica, two proprietary state of the art computing environments.

Compliant Robot Control

(cf. Part A Chapter 6, *Motion Control* [Siciliano and Khatib, 2008])

Compliant robots are recognized to generate more natural motions than classical rigid robots [Moro et al., 2011]. In order to execute motion primitives on a compliant robot, control of the robots limbs or manipulators is required. This subdomain is covered with roughly every second of the DSLs surveyed by [Nordmann et al., 2014], however, especially kinematics, dynamics, and classical motion control.

Examples of this subdomain are contributions by Frigerio et al. [2013, 2012a] and Laet et al. [2012c] that target kinematics and dynamics controllers that can be embedded in motion control systems. Artifacts generated from DSLs of the kinematics, dynamics, and motion control subdomains are usually controllers as well as robot simulation and visualization support Bordignon et al. [2010].

3.3.2. Adjacent Subdomains

Adjacent subdomains are considered part of the domain, but not primarily targeted by this work. This means that their main abstractions need to be covered to be included and used in the development, e.g., aspects that need to be supported on an architectural level.

Force Control

(cf. Part A Chapter 7, *Force Control* [Siciliano and Khatib, 2008])

The force control subdomain targets robust and dynamic behavior of robotic systems in compliant interaction with the environment. It includes different control aspects, e.g., torque, stiffness, and impedance control. The subdomain is not very well covered by MDE approaches and DSLs yet [Nordmann et al., 2014]. While several DSLs deal with the dynamics of robots [Frigerio et al., 2013, 2012b], which is a pre-condition for force control, only Klotzbücher et al. [2011] make force control operations an explicit concept of their language.

Perception

(cf. Part C, *Sensing and Perception* [Siciliano and Khatib, 2008])

Although it is not a direct part of the motion control system of a robot, perception is usually necessary in robotic systems and often strongly tied to motion. This is especially true for proprioception, sensing of the robot limbs, e.g., for obstacle avoidance in fast sensorimotor loops. Therefore, external perception is considered an external subdomain and the paragraph focuses on proprioception. Exemplary related MDE and DSL are early publications by Henderson and Shilcrat [1984], Gordillo [1991] to specify sensor systems and recent work by Hochgeschwender et al. [2013, 2014] for specification of perception architectures.

Proprioception Proprioception in robotics is the sense of the relative positions, forces, and torques of the robot's limbs. Typical sensor values in robotics are joint angles, joint velocity, joint acceleration, joint torques, but also its current Cartesian pose, velocity and acceleration when gyroscopes and accelerometers are used.

Machine Learning

(cf. Part C Chapter 9.4, *Robot Learning* [Siciliano and Khatib, 2008])

Machine learning adds flexibility and adaptability to the motion primitives targeted in this thesis. Nordmann et al. [2014] did not survey any DSLs dedicated to machine learning. There are, however, machine learning DSLs outside the robotics context, such as the work by Sujeeth et al. [2011] for description, analysis, and code generation of machine learning approaches. Machine learning in this work has to be supported not in detail, but to on an abstraction level that allows the introduced learning capabilities of motion primitives, e.g., programming-by-demonstration, on an architectural level.

Physical Human-Robot Interaction

(cf. Part G Chapter 59, *Robot Programming by Demonstration* [Siciliano and Khatib, 2008])

Physical human-robot interaction is often a goal of natural movement generation with motion primitives. Its abstractions and essential concepts are, among others, interaction forces, interaction triggers, and kinesthetic teaching. This subdomain has links to the machine learning subdomain in terms of programming-by-demonstration and kinesthetic teaching, as well as to safety.

3.3.3. External Subdomains

External subdomains are the ones that regularly occur together with motion control architectures, but are not covered by this work. Extension points to these are indicated in the course of this thesis. The following sections therefore just indicate the links to the core and adjacent subdomains.

External Perception

(cf. Part C, *Sensing and Perception* [Siciliano and Khatib, 2008])

External perception (sometimes also referred to as "exteroception") is the sensitivity to stimuli originating outside of the body, as opposed to proprioception. It is regularly used together with motion architectures, as it helps recognizing the environment, e.g., to detect goals, objects, and obstacles to be avoided. This subdomain is considered as an adjacent subdomain, e.g., to generate high-level goals for the motion generation.

Safety

(cf. Part G Chapter 57, *Safety for Physical Human-Robot Interaction* [Siciliano and Khatib, 2008])

Safety is especially relevant in the context of pHRI. Robots targeted to pHRI must fulfill several requirements to be suitable for collaboration. Requirements on the cycle time or accuracy, aspects concerning reliability and dependability become more importance as the robot works in contact with humans [Matthias et al., 2011, Haddadin et al., 2009]. Only a few DSLs explicitly target safety concerns, an example being the work by Adam et al. [2014] who present a DSL to specify safety-related rules that a system must obey as well as corresponding actions when these rules are violated.

Reasoning / Cognition

(cf. Part A Chapter 9, *AI Reasoning Methods for Robotics* [Siciliano and Khatib, 2008])

For autonomy, the robot has to be able to reason about itself and its environment, which can be supported by model-driven methods when those models are used at run time. Robotics DSL approaches, however, still use their models and languages mainly at design time [Steck and Schlegel, 2011, Nordmann et al., 2014]. Only a few of the surveyed approaches use the models to exploit the represented knowledge also at runtime, e.g., to model runtime variation points in the task at design time and use them at runtime [Steck and Schlegel, 2010, 2011], or to synthesize DSL programs while learning from demonstration as done by Feniello et al. [2014]. This subdomain is considered as an external subdomain, e.g., to perform high-level planning and generate goals for the motion generation.

3.4. Discussion

The domain of motion primitive architectures is a broad domain composed of several subdomains and adjacent to even more, as shown above. This is recurring typical problem in robotics since most of these domains need to be handled to create entire running systems. This calls for integration support for users that are not and (obviously) cannot be experts in all of these subdomains.

The main goal of the domain is the creation of rich motions of human-like or animal-like complexity. Since the coordination and composition of several motion primitive

into one architecture is the main hypothesis in order to reach this goal, a model-driven engineering approach in this domain needs to support specification, development, and composition of motion primitive architectures. Research on this hypothesis with robotics systems is a big challenge due to two primary reasons:

1. Motion primitives are usually not explicit, but hidden in general-purpose language (GPL) source code, intertwined with functional components from adjacent subdomains, such as perception and motion planning, so that their properties are hard to identify which makes the different motion primitive approaches hard to combine with each other.

2. The same reason, single handcrafted motion primitive experiments, also leads to a large corpus of valuable work basically not capable of being integrated on a technical level, due to different programming languages, architectural decisions, and platform dependencies (software and hardware).

This imposes several requirements for the targeted model-driven engineering approach as elaborated in Section 3.3, including i) specification of the motion primitive itself in terms if dynamical systems and machine learning capabilities, ii) support of periodic and goal-directed movements, iii) explicit handling of dependencies to the robot in terms of perception and actuation, and iv) composition of motion primitive. These challenges motivate the research questions introduced in Chapter 1 to ease and support development and research in this domain.

While some of the required aspects are already covered by existing domain-specific language approaches, as shown in Section 3.3, there is no model-driven engineering approach available to solve this task. However, reuse of existing and available models and solutions will be discussed in the course of this thesis.

The domain analysis and observations made in this chapter, together with the observations made in Chapter 2, motivate the objectives, functional and non-functional requirements, and eventually the approach proposed in the following chapter.

Chapter 4.

Model-Driven Engineering for Motion Primitive Architectures

With the basic concepts of model-driven engineering (MDE) introduced in Chapter 2 as well as the main concepts, challenges and requirements of the motion primitive architecture domain introduced in Chapter 3, this chapter proposes and instantiates a clear systematic design process to deploy the discussed MDE methods in the motion primitive architecture domain. To answer the research questions introduced in Chapter 1, it follows a process of i) extracting domain knowledge, ii) separating the domain knowledge by concerns iii) formalizing the domain knowledge, iv) providing development support, and thereby providing a development environment for the domain expert that allows easy formulation of domain problems, solutions and experiments and to make them executable on robots.

The proposed design process employs and adapts existing patterns for software architecture development [ORMSC, 2001, OMG, 2014, Völter, 2005], as introduced in Section 2.4, which seem to be suitable for use in the targeted inter-disciplinary motion control architecture domain. It aims for early integration and involvement of important stakeholders and to establish an approach to ground the conceptual ideas found in the domain analysis in a resulting *architectural process* to link theoretical research on motor skills with software engineering science and robot hardware, eventually resulting in executable and reproducible robotics experiments.

Section 4.1 discusses the research project context where the proposed MDE approach is employed and discusses the specific challenges and requirements imposed in this context. Section 4.2 translates these challenges in a set of more concrete objectives, functional and non-functional requirements that need to be met by the proposed design process and its resulting domain-specific language (DSL) based MDE process. The concrete design process that is proposed in this work and exemplified in the domain of motion primitive architectures is detailed in Section 4.3.

4.1. Project and Domain Context

Employing an MDE approach in the motion primitive architecture domain in a research context, as proposed in this thesis, introduces additional requirements apart from the general requirements of the model-driven methodology and the design of the domain-specific languages itself, as discussed in Chapter 2, and the requirements from the

motion primitive architecture domain, as discussed in Chapter 3.

A first challenge is to consider the developer background and bias to ensure easy introduction of the proposed method. Although the idea of motion primitives is driven from biology, their developers, in this case researchers creating prototypes to test their motion primitive (MP) hypotheses as the ones presented in Section 3.3, typically have a strong bias to the development tools and environments they are used to [Ritter et al., 2008]. In robotics, a domain typically strongly tied to software development, this bias is usually directed to component-based software engineering (CBSE). CBSE systems are best practice in software development and in principle allow composition of systems from re-usable components. This also needs to be considered when designing a MDE process, since it is vitally important, in research projects as well as in almost any project, to keep the entry threshold low to continuously being able to create and maintain executable, run-capable systems. In an industrial project, the customer expects run capable proof-of-concepts and intermediate versions, in research projects the project executing organization, e.g., the European commission, expects deliverables, demonstrators and runnable experiments to verify project progress.

In a MDE approach, this is not possible in a pure forward process, as vertical prototypes need to be developed first, software libraries need to be implemented, and the core concepts have to be agreed upon and might have to be challenged and updated regularly during the project due to new scientific results. This is not necessarily the case in stable (as in: already stabilized) domains, but this is certainly true in a research project like the European AMARSi project, where this process is employed and concepts are developed and agreed upon during (almost) the entire lifespan of the project.

When concepts are clear, or at least a first iteration, the MDE needs to be instantiated and the resulting toolchain needs to be implemented and rolled-out to the developers and integrated into their development process. However, even when a first iteration of the concepts and toolchain is ready and developers are asked to start adapting the model-driven development process, it is usually not realistic to cover systems completely right from the start. This calls for a way to integrate existing software artifacts, e.g., parts of the systems surveyed in Section 3.3, into the MDE process.

A toolchain also should target covering technological needs from all involved developers. The conducted survey for example already surveyed four different languages and platforms: C, C++, Python, and MATLAB. Even when it is not feasible to provide model-driven support for all these languages and environment, e.g., by providing a technology mapping and code generation toolchain for all of these, developers should not be completely locked out of the development process. Therefore, the targeted approach should be open for integration of systems and components from different environment and languages. This is not only a question of mitigating risk but also of leveraging the corpus of previous (usually significant) development efforts.

From a practical point of view, this also opens the overall process to developers that do not want to subscribe to a MDE approach and rather develop in the classical way, e.g., with general-purpose languages, as they can integrate their hand-crafted compo-

nents and systems. Although this is often not desired, as developments outside the MDE process lose the corresponding advantages such as correctness by construction, model validation, etc., this is based on project experience an important requirement for adoption of a MDE approach.[Selic, 2003]

4.2. Objectives and Requirements

The objectives for the design process are driven by the research questions formulated in Chapter 1 as well as MDE best practices as discussed in Chapter 2 and the challenges of combining motion primitives to rich motor skills in a research context, as discussed in Chapter 3. The targeted domain-specific languages and their respective development environment and process need to be expressive enough to represent the examples found in the domain analysis in Section 3.3, yet address the challenges of the research context as discussed in the previous section. In addition to expressing the (isolated) experiments and examples found in the domain analysis, it additionally has to support combination of motion primitives in a unified architecture to allow validation of the base hypothesis of the domain, i.e. that the combination of motion primitives yields rich motor skills.

Ease Specification of Domain Problems and Examples Problems and their solution can be specified on different levels of complexity. However, the more complex the specification gets, the more difficult it is to understand and to manage, thus increasing the possibility of errors. One of the key techniques to deal with this problem of complexity is through abstraction. By allowing different levels of abstractions, it is possible to expose the complexity that is necessary in a particular context even of complex problems. The design process therefore shall target an abstraction level that allows coping with the complexity of rich motion skills, with motion primitives already being a biologically motivated idea on how these abstractions could look like.

Platform-Neutrality Executing experiments on different robot platforms is one of the natural requirements in robotics, but also explicitly stated in the European research project AMARSi where this approach was deployed. Platform-neutrality has a strong relation to abstraction, since abstracting problem and solution specification from technological aspects not only helps coping with their complexity but also is a key to platform-neutrality.

Restriction The main hypothesis behind motion primitives is that their composition allows generation of complex motions. Yet, many motion primitives are bound to specific platforms, i.e., robotics platforms and software frameworks, implemented in different programming languages and against different application-programming interfaces (APIs), mixed up with other components or application code, and not explicitly represented [Nordmann and Wrede, 2012]. As this threatens the idea of validating the

hypothesis of motion primitives composition on robotics, the design process shall provide means to easily combine motion primitives, e.g., by restricting and homogenizing the architectural choices in favor of compatibility.

Completeness The approach proposed in this work is targeted to motion primitive experts and explicitly wants to avoid that they have to also become software engineers in order to try their research hypotheses on robots. To achieve that programs do modeled with the approach proposed in this work require additional configuration files or code written in a general-purpose language (GPL) to make it executable, the models and languages have to be *complete*. Completeness in this sense refers to the *"degree to which a DSL can express programs that contain all necessary aspects."* [Völter et al., 2013] In cases where several viewpoints represent various concerns of a domain, the set of fragments written for these concerns must be enough for complete generation in order to fulfill this criterion.

Openness In the research context where this design process was deployed it is vital to deal with the fact that a large corpus of legacy work is already available, both in terms of previous classical development approaches as well as in terms of alreadyexisting software artifacts. Introducing a model-driven and DSL based development process into such a context needs to incorporate this legacy work to have a chance of being adopted. This means not only that it should be technically compatible with legacy software, but also the development process and development environment itself should already be integrated into the legacy process and legacy development environment [Selic, 2003].

This is helpful to mitigating risk, to leverage previous significant work, and to ease adoption of the approach by project partners and developers.

4.2.1. Requirements

The above objectives can be formulated as a number of concrete functional requirements (**FR**) and non-functional requirements (**NFR**) that the design process, the resulting DSL development process and its software artifacts have to match. The requirements are enumerated and referenced throughout this thesis to motivate and discuss design decisions.

4.2.1.1. Functional Requirements

The functional requirements are mainly driven by the domain analysis.

FR1 Allow representation of motion primitives in terms of dynamical systems, which is the main functional building block of the domain.

FR2 Allow adaptation of motion primitives through machine learning to allow for the flexibility targeted by motion primitives.

FR3 Allow combination of motion primitive in an architecture, as this allows testing the main hypothesis of the domain.

FR4 Allow dependencies to proprioceptive feedback such as joint angles, joint velocities, joint acceleration, and joint torques, as well as Cartesian poses, velocities and accelerations for reflexes and reaction to the unstructured environments.

FR5 Interface to higher cognitive layers and external perception, e.g., to receive goals.

4.2.1.2. Non-Functional Requirements

The non-functional requirements are largely driven by the research and project context.

NFR1 Use standards and open tools whenever possible and reasonable to increase openness and foster reuse.

NFR2 Consider and respect developers bias, e.g., typical structure of systems and software of the domain.

NFR3 Integrate existing software artifacts and development processes to leverage legacy work and ease transition from classical development processes and environments.

NFR4 Allow iteration and constant refinement to keep up with new research results.

NFR5 Ease expressing of domain problems and solutions to support the domain expert and help prevent errors.

NFR6 Allow formulation of motion primitive architectures in a technology-independent and platform-neutral way so that they survive platform change and can be used across different hardware and software platforms.

NFR7 Allow execution of systems on different platforms, i.e. robot platforms and software platforms.

NFR8 Generate complete executable programs, so that the domain expert does not have to learn GPL languages to experiment with motion primitives.

NFR9 Allow execution in simulation to decrease turn-around time of experimentation and to respect potentially high costs and effort of hardware experiments in robotics.

4.3. Process

The objectives and requirements from the previous sections have to be considered for the design process as a whole as well as for its several design steps and their resulting artifacts, e.g., the resulting model-driven engineering process, software architecture,

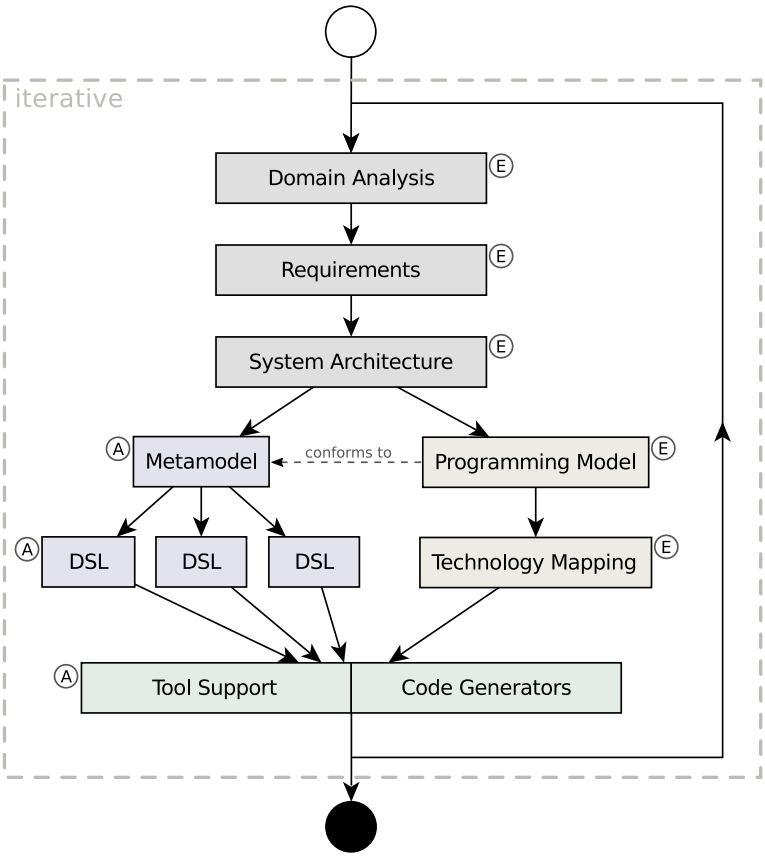


Figure 4.1.: Proposed iterative design process. Activities of the elaboration phase are labeled with Ⓔ; activities of the automation phase are labeled with Ⓐ.

and toolchain. This section motivates the design process and explains its process steps while pointing out the specific requirements that influence them. The proposed design process, cf. Fig. 4.1, adapts patterns for software architecture development, which seem to be suitable for use in the targeted inter-disciplinary motion primitive architecture context (**FR1**).

Therefore, the proposed architectural process shown in Fig. 4.1 adapts Model Driven Architecture (MDA) [ORMSC, 2001] based on the ideas of Völter [2005]. It aims for early integration and involvement of important stakeholders and to establish an approach to ground the conceptual ideas found in the domain analysis in a resulting *architectural process*, linking theoretical research on motor skills with software engineering science and robot hardware, eventually resulting in executable and reproducible robotics experiments. It uses DSLs to formalize the domain knowledge and make it available for automation and creating of executable robotics systems.

The design process is iterative to account for the circumstances and needs of a research project, where concepts change and evolve or have to be iteratively tested in experiments (**NFR4**). The iteration starts with a *Domain Analysis* to extract the

relevant abstractions, features, relations, and *Requirements* of the targeted domain. This step is not explicit in [Völter, 2005] but it is in the Model Driven Architecture standard of the Object Management Group (OMG). The domain analysis for the motion primitive domain is already detailed in Chapter 3 and motivated most of the requirements shown in Section 4.2.1. As a next step, based on the requirements and results from the domain analysis, a technology-independent *System Architecture* is created which focuses on the structure and responsibilities of the system parts. It should therefore make implementation of the functional requirements as efficient as possible (**NFR5**).

To evaluate the technology-independent system architecture and therefore base the ground for iteration of the design process, Völter [2005] defines two further steps in the elaboration phase: creating the *Programming Model* and a *Technology Mapping*. The programming model describes the API of the architecture, i.e. how it is used from a developer's perspective. Software abstractions and interfaces in the programming model conform to the technology-independent system architecture. As a next step, this programming model is implemented with concrete technologies, software platforms and programming languages, i.e., *mapped* to an exemplary technology mapping. This is intended for evaluation of the architecture and can be done in two ways. It can be first evaluated on a *Mock Platform*, e.g., a robot simulator, for fast and easy evaluation that spares the typically great effort of experimenting with hardware. As a second important step when feedback of the first evaluation is already incorporated in iterations of the process, the technology mapping is tested on a *Vertical Prototype* including all aspects of the architecture [Völter, 2005]. In robotics, this usually includes testing on robotics hardware. Up to this point the activities belong to the elaboration phase as defined by Völter [2005], indicated with an Ⓔ in Fig. 4.1.

When the system architecture is evaluated, the automation phase starts, indicated with an Ⓐ in Fig. 4.1. This process targets automation of software development based on DSLs. Design of domain-specific languages is usually done *"metamodel first"* [Völter, 2010], i.e., first a schema is defined, then the editor and/or grammar is added, and then the additional tool support like code generators are added. Therefore as a first step, the technology-independent system architecture is formalized in a *Metamodel*. The metamodel (or "architecture metamodel") formally defines the concepts of the system architecture to make it machine-readable and accessible for later automation. In the proposed process, the metamodel is realized as a set of DSLs, for which it provides the abstract syntax. The DSLs add a concrete (textual or graphical) syntax to make it accessible for programming and editor support. The DSLs use the natural notation of the domain experts so that they are easy to write, read, and understand by the domain expert, who is supposed to be supported by this automation.

Reality in the development of complex systems shows that different experts and stakeholders develop their concerns with their respective tools and environment. The language modularization, extension, and composition approach provides the possibility to design different DSLs for specific concerns and their respective experts/stakeholders, which can be specifically targeted to the respective concern (**NFR5**). At the same time fosters reuse on a language level (**NFR1**) as discussed in Section 2.2.1.

After the DSLs are created, both, the DSLs and the technology mapping are integrated to provide a development environment that can support the developer through automation, e.g., code generation by transforming the formal DSL models to the chosen technology mapping. Once this is established, domain experts can formulate their scientific hypothesis in a language that is natural to them and restricted to their problem domain. Domain-specific model validation and verification based on the formal models can then prevent errors already during specification and code generation allows generation of executable systems without domain experts needing to write GPL code.

4.4. Discussion

This chapter proposes a clear systematic design process for domain-specific languages in robotics that is applied in the domain of motion primitive architectures. It is a first contribution of this work to answer the research question **RQ1**, suggesting a method to exploit and support the complex domain of motion primitive architectures by means of model-driven engineering methods.

The MDA approach is standardized, well known, and well supported in several domains outside robotics, and is therefore a strong candidate for applying model-driven engineering in robotics. However, as this thesis aims at using domain-specific languages as motivated in Section 2.2, the focus on Unified Modeling Language (UML) introduces too strong restriction on their development and expressiveness.

The process proposed in this chapter conforms to existing patterns for software architecture development [ORMSC, 2001, OMG, 2014, Völter, 2005] and adapts them to the motion primitive architecture research context according to the functional and non-functional requirements discussed in this chapter. The iterative character, inspired by Völter [2005] addresses the research context (**NFR4**). Systematic design of a metamodel and DSLs address eased and platform-independent formulation of motion primitive architectures by domain experts (**NFR6, NFR5**). The proposed code generation targets execution of these motion primitive architectures on different platforms (**NFR7**) without the need of manual programming (**NFR8**).

Part III.

Developer Perspective

Chapter 5.

Technology-independent Architecture and Metamodel

Following the approach proposed in Chapter 4, this chapter introduces a technology independent architecture and metamodel that covers the domain introduced in Chapter 3. The technology-independent architecture is intended to cover a domain, yet be simple enough to be *"explainable on a beer mat"* [Völter, 2005], i.e. to be reasonably simple to be understood by all stakeholders and developers. It should make it possible to express the topic efficiently, independent of specific technologies and implementation strategies. It clearly defines the concepts, constraints, and relationships of the architectural building blocks, which constitute the domain of adaptive rich motor skills. The presented architectural metamodel is a product of this work. Its essential concepts and respective consistent terminology are detailed in the following.

The metamodel is a conceptual model of the various entities, their attributes, roles, and relationships, plus the constraints that govern the problem domain. It does not describe solutions of the problem domain, but *allows* describing them. The metamodel is basis for the DSLs discussed later in Chapter 6 and therefore basis for the domain models that describe a domain application or solution, as discussed in Section 2.1. As foreseen by the process introduced in Chapter 4, the concepts of the metamodel were constantly validated and incrementally refined / extended by applying them in vertical prototypes of the domain together with domain experts, e.g., practically involving partners of the European research project AMARSi in testing and evaluation.

When designing a model it is important to know the motivations of the model, and therefore which qualities or relationships are of particular use or should be reasoned about. The model presented in this chapter has two main motivations: to make motion primitive explicit, and allow their easy composition into an architecture. The architectural metamodel shown in Fig. 5.2 depicts the basic models of the domain and its relations. Similar to several approaches discussed in Section 2.3, such as the V^3CMM approach [Alonso et al., 2010], it shows three viewpoints of the metamodel: the static structural architectural aspects (black, solid), dynamical behavioral aspects (yellow, dashed) and algorithmic models (green, dotted).

In the course of this chapter, these models are detailed and discussed along these three viewpoints. `Domain concepts` and their `instances` are highlighted in typewriter font. A running example is introduced and used in the course of this chapter to

Example: Quadruped walking and foot placement
Running example comprising of the quadruped robot Oncilla [Spröwitz et al., 2011] and the combination of a periodic motion primitive for walking as well as a goal-directed motion primitive for foot placement, showing the combination of motion primitives in an architecture to perform a motion skill.

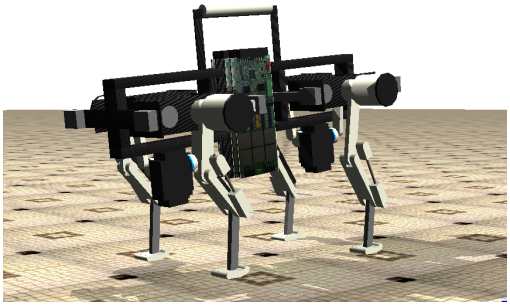


Figure 5.1.: Quadruped robot Oncilla in simulation.

illustrate various – otherwise abstract – concepts.[1] The example comprises of the Oncilla quadruped robot (Oncilla) [Spröwitz et al., 2011], shown in simulation in Fig. 5.1, and the combination of a periodic motion primitive for walking as well as a goal-directed motion primitive for foot placement (a quadruped robot's version of a reaching motion). The example is not meant to show the entire complexity of the domain, but rather serve as a simplified example to ground the introduced, rather abstract concepts, yet showing the combination of motion primitives in an architecture to perform a motion skill. For more complex examples, see Chapter 10.

Concepts introduced in this chapter are linked to findings of the domain analysis for better traceability.

5.1. Structural Models

The architectural building block proposed in this work to represent a motion primitive as a combination of dynamical systems and machine learning for adaptation is termed *Adaptive Module*. They resemble computational building blocks to generate primitive, combinable motions that can generalize to new situations or environments and are robust to perturbations. The main concept of `Adaptive Modules` is accompanied by further structural concepts to organize and combine them in an architecture. Indicated in black with solid lines in Fig. 5.2, they form the main abstractions to express static aspects of motion primitive based architectures. The following section details these concepts, their properties, and relations.

[1] *Running* example in the sense that this example is used throughout this and the following chapters.

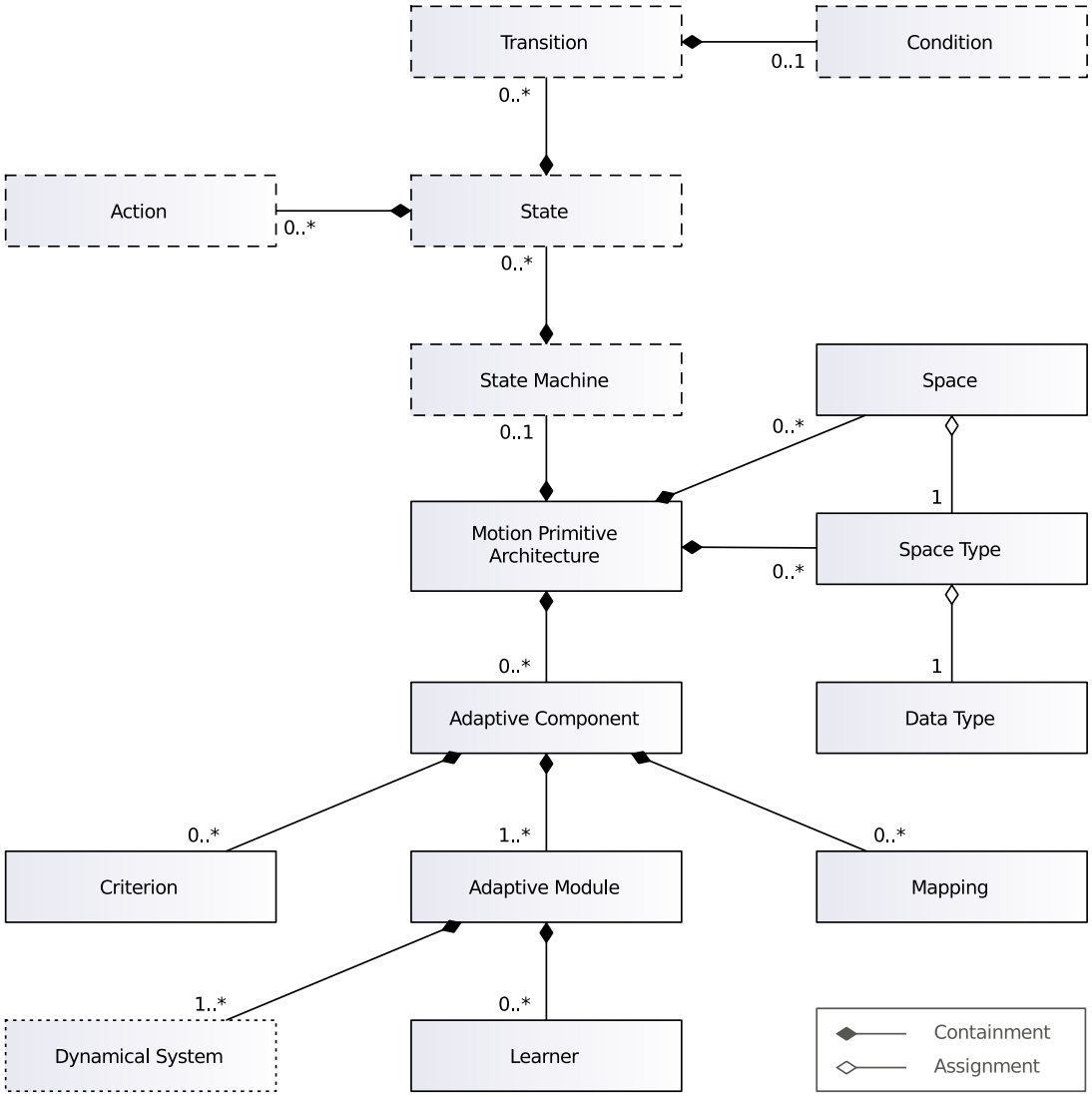


Figure 5.2.: Class diagram of the metamodel for the main abstractions and relations of the motion primitive architecture and its three separate architectural concerns: structural (solid), behavioral (dashed), and algorithmic (dotted).

Example: Spaces and Space Types
In the running example introduced above, two `Space Types` are defined: i) `Leg Joints`, a 2-dimensional `Joint Angles` space that represents joint angles of a single Oncilla leg, and ii) `Foot Position`, a 3-dimensional task-space of `Translation` type to represent the 3-dimensional Cartesian position of a single Oncilla foot. For each leg, the example defines four spaces of each of these two `Space Types`, the status and command of the leg joints of `Leg Joints` type, as well as the status and command of the foot of `Foot Position` type. Fig. 5.3 shows exemplary `Space Type` and `Space` instanced of the running example.

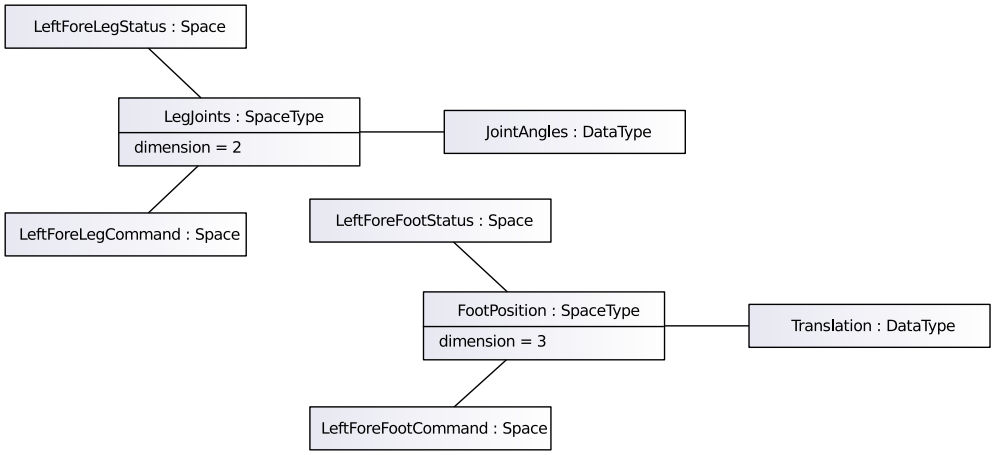


Figure 5.3.: Object diagram of `Spaces`, `Space Types`, and `Data Types` in the running example.

5.1.1. Spaces and Space Types

The concept of `Spaces` abstracts the communication between system components of the motion primitive architecture. It is defined as a number of explicit variables that appear to be jointly manipulated or sensed somewhere in the motion control architecture, e.g., joint angles of a certain robot limb. This usually occurs on all levels of an architecture, but is often very implicitly defined by the functional architecture, as observed in the domain analysis.

`Spaces` are determined by their `Space Type`, which consists of a `Data Type` and a `Dimension`. The base `Data Types` of `Spaces` are the ones found in the domain analysis, such as `Joint Angles`, `Impedance`, or end-effector `Pose`. The domain analysis, especially the control feature models (cf. Section B.1.1) and the input and output specification of the adaptive modules survey (cf. Section B.2), showed that one cannot assume a fixed, limited and a-priori known set of `Spaces` and `Space Types`. Instead, their explicit specification depending on chosen application and chosen robot platform needs to be facilitated.

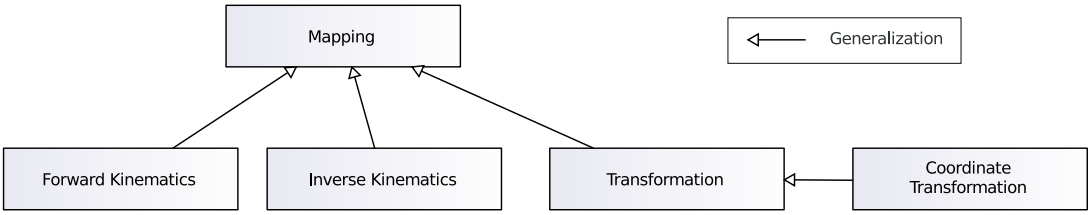


Figure 5.4.: Specializations of Mappings and Transformations.

Spaces can describe status values as well as target values and setpoints. In addition to the status and measurements of the robot itself, they can encompass values of the external, perceived world and setpoints generated by a higher cognitive layer. They therefore address the functional requirements of i) allowing interfacing with higher cognitive layers and external perception (**FR5**) and ii) making dependencies to proprioceptive feedback, e.g., for reflexes and reaction to the environment, explicit (**FR4**). Since they make this dependency explicit, they additionally help with decoupling experiments and methods from robot platforms (**NFR6**, **NFR7**), as they represent defined semantic interfaces between them.

Making Spaces explicit also helps dealing with their technical constraints in an explicit manner. For instance, resource management and arbitration, or data with distinct update rates that need to be temporally aligned first through, e.g., sub-sampling or temporal interpolation.

5.1.2. Mappings and Transformations

In the motor control domain, forward and inverse models are commonly used for the mapping between Spaces, e.g., task and joint variables. These models are represented by Mappings and Transformation: Mappings map data between Spaces of different Space Type, Transformations transform data between Spaces of the same Space Type, see Fig. 5.4. Typical Mappings found in the domain are Forward Kinematics and Inverse Kinematics, found in the domain analysis as well as in the running example. Typical Space Types are coordinate transformations between, e.g., coordinate systems of different sensors or different robots.

How such Mappings and Transformations are exactly implemented is not important in order to identify the crucial role of forward and inverse models in a motion architecture. In fact, Mappings and Transformations themselves may be implemented or learned. Note, that Mappings and Transformations are currently treated as black boxes regarding their implementation. However, this is a potential extension point for more detailed modeling efforts, e.g., work by Laet et al. [2012b,a], which is already prototypically implemented as DSL, detailed by Laet et al. [2012c].

If they are not implemented or explicitly modeled, every two Spaces of a system can be investigated if a forward and inverse model exists and can be learned. The concept of an Adaptive Mapping or Transition that can be learned is termed Adaptive Mapping and has input and output semantics for learning similar to the Adaptive Mod-

Example: Mappings and Transformations
The running example requires an inverse kinematics `Mapping`, to map between the `Foot Position` commands and the `Leg Joints` space for commands sent to the robot.

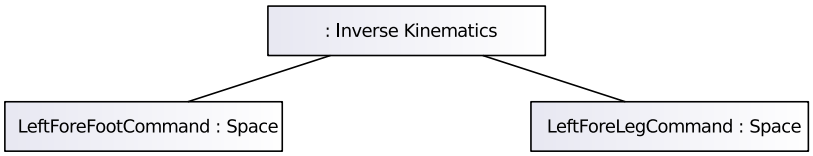


Figure 5.5.: Object diagram of `Inverse Kinematics Mapping` between `Foot Position` commands and the respective `Leg Joints` commands.

ule. An example from the domain analysis for this concept is the learned redundancy resolution by Wrede et al. [2013], Emmerich et al. [2013], Nordmann et al. [2012a].

5.1.3. Adaptive Modules

An `Adaptive Module` is the main functional building block of the motion control architecture and represents a motion primitive. The domain analysis showed that an `Adaptive Module` consists of one or more `Dynamical Systems` together with its relevant input and output `Spaces` and a machine learning mechanism for adaption. An `Adaptive Module` contains one or more `Dynamical Systems`, which generate the output dynamics and can have periodic or non-periodic (goal-directed) dynamics.

To allow adaptation to new situations, new environments and additional learning input, `Adaptive Module`s optionally contain one or more `Learners` that adapt the `Dynamical Systems`. A common case for adaptation is to *shape* the dynamics as done with kinesthetic teaching in the catching [Shukla and Billard, 2011] and upper body control [Reinhart and Steil, 2012] examples. A classic example that can be described with the concept of an `Adaptive Module` is the Dynamical Movement Primitive (DMP) as introduced in Chapter 3, such as the gait pattern generators and reflexes used for the quadruped walking [Ajallooeian et al., 2013]. The distinction between `Dynamical Systems` and the adaptation mechanism, the `Learner`, is in most examples too fine-grained as most of the mechanisms found in the domain analysis generically include adaptation mechanisms. However, the distinction is nevertheless useful in cases where a `Dynamical System` comes without an adaptation mechanism, e.g., is given by a fixed expression, or is already pre-trained.

Also along the findings of the domain analysis and survey (cf. Section B.2), the `Adaptive Module` concept defines a set of dedicated `Inputs`, e.g., to receive goals during execution and sensor values, `Inputs` for configuration as well as `Inputs` for learning data. It also defines `Outputs` for its control output and status, see Fig. 5.6.

The `Adaptive Module` concept has a specific lifecycle with different states, cf. Fig. 5.7, e.g., for execution, online and offline learning. The distinction between on-

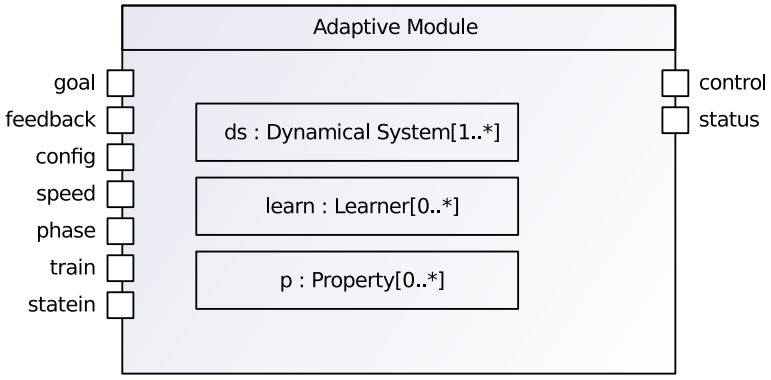


Figure 5.6.: Composite structure diagram of an Adaptive Module with dedicated Inputs/Outputs, Dynamical Systems, Learners, and Properties.

line learning and offline learning is especially relevant from an architectural viewpoint. Both cases need to be treated differently, since responsiveness and execution of the Adaptive Module might differ in the online learning case and the offline learning case. While an Adaptive Module might be able to perform a movement during online learning, in the offline learning case further parts of the architecture may need to collect feedback first, parts of the system may have to be frozen, and the Adaptive Module might be unresponsive during learning. It is therefore useful to distinguish online learning and offline learning on the conceptual level, while the designers of a concrete Adaptive Module still needs large freedom to implement different adaptation and learning schemes that shall not be strongly restricted.

The distinction between the different lifecycle states of the Adaptive Module is therefore reflected in the Adaptive Module Status.

Adaptive Module Status The Adaptive Module Status makes the status of an Adaptive Module visible to surrounding system and coordinating levels. It is a Data Type so that it can be used to specify a certain Space Type as well as respective Spaces and can have one of the following three values:

1. **executing**: The Adaptive Module is not learning, but executing a movement or ready for execution,
2. **online learning**: The Learner is training the Dynamical System(s) in online learning mode, allowing the Adaptive Module to be executed in parallel, or
3. **offline learning**: The Learner is training the Dynamical System(s) in offline learning mode and the Adaptive Module is currently not available to perform a movement.

Dynamical System Status Similar to the Adaptive Module itself, the Dynamical System concept has an internal status that allows the Adaptive Module to behave

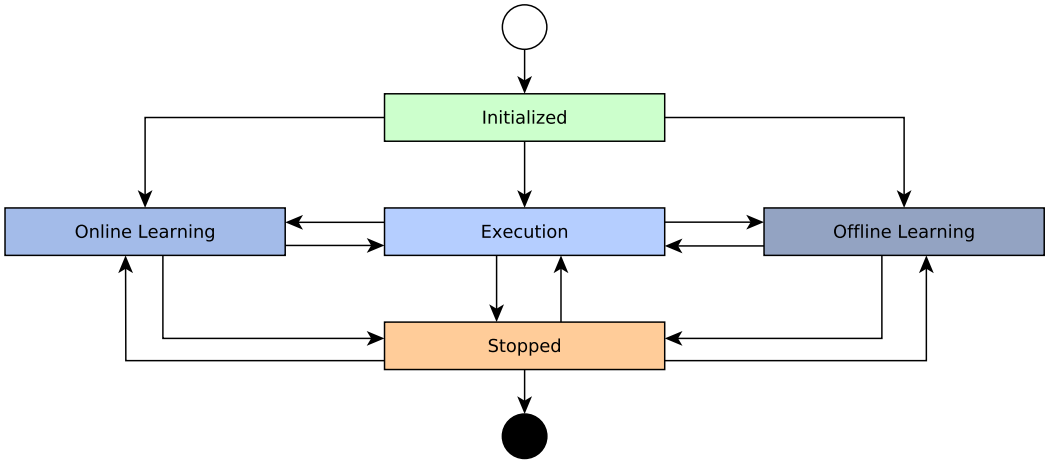


Figure 5.7.: Adaptive Modules lifecycle; the Adaptive Module can be initialized, stopped, and switched between execution, online learning, and offline learning.

based on the state of its contained Dynamical Systems. The domain analysis showed primarily three different states, which provide a clue to the Adaptive Module on the state of its movement:

1. **transient**: the Dynamical System is potentially in untrained regions and therefore uncertain whether it is converging and producing intended or meaningful output.

2. **converging**: the Dynamical System is potentially in a trained region, approaches its desired behavior in a controlled way, but did not reach its goal or attractor yet.

3. **reached**: the Dynamical System is in its desired behavior, i.e., it reached its goal, is following its attractor state, or is stably producing its intended pattern.

5.1.4. Adaptive Components

The domain analysis revealed that Adaptive Modules come with a limited set of architectural structures in their vicinity that determine the semantics of the Adaptive Modules and the represented motion primitives. This finding yielded the concept of Adaptive Components that allows a compact description of these architectural structures. It is defined as an Adaptive Module together with its input and output Spaces and the control logic inside the component. It provides the logical structure around an Adaptive Module, connecting it with the entire system in its different lifecycle states, e.g., learning and execution.

Example: Adaptive Module
The pattern for walking of the quadruped robot is generated by a contained periodic Dynamical System.
This specific Adaptive Module does not have any goal or feedback Input, but only configuration Inputs for its frequency and phase property, as well as the lifecycle. It has Outputs to generate joint angles that perform the walking pattern with the robot and to report its current Adaptive Module Status.
The frequency property of the Adaptive Module changes the frequency of the walking pattern, which should be close to the natural frequencies of the compliant robot and its limbs to produce a forward movement. The phase Input adds an offset to the phase to allow it to start and continue at different points in the stance/swing phase.

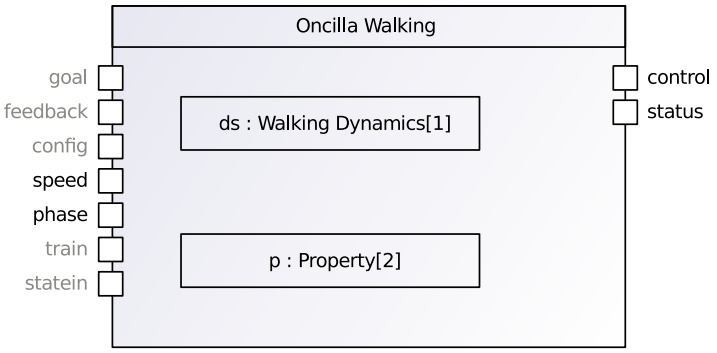


Figure 5.8.: Composite structure diagram of the Adaptive Module in the running example for generating the walking dynamics.

Additionally, Adaptive Components can be configured to have Mappings or Transformations on the goal Input, feedback Input or control Output of the Adaptive Module. This allows integrating Adaptive Module into different systems and platforms by configuring their containing Adaptive Components accordingly, if they were otherwise incompatible Spaces. An Adaptive Module operating in joint space can for example be integrated in a system where the goal is generated in task space by adding the according inverse kinematics Mapping, as it is shown in the running example. It is thereby a further answer to the challenge of executing motion primitive experiments to different robot platforms (**NFR7**), as the concept of Adaptive Components explicitly allow integrating Adaptive Modules in different contexts by adapting the surrounding Adaptive Component without having to change the Adaptive Module.

Adaptive Components can optionally specify a Criterion that monitors convergence of the motion, detailed in the course of this section. Four specialized subtypes of Adaptive Components were identified during the domain analysis as shown in Fig. 5.9:

- Reaching Controllers for goal directed movements get targets and converge

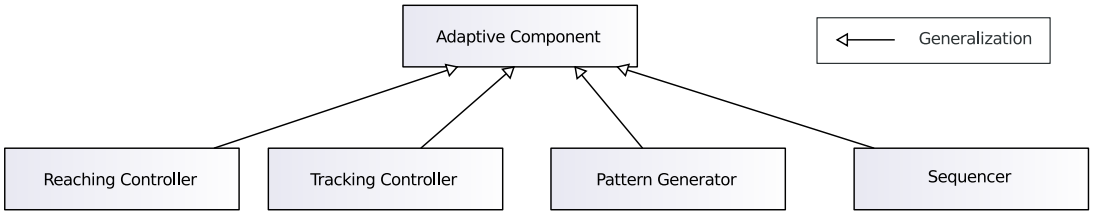


Figure 5.9.: Specializations of the Adaptive Component concept.

towards this target. It has a Criterion that analyses the reaching progress based on feedback and determines the Adaptive Component Status.

- Tracking Controllers converge towards a reference trajectory or a cyclic attractor. The convergence dynamics to the target trajectory is a mixture of the Adaptive Module's dynamics and the input dynamics. Similar to the Reaching Controller it has a Criterion that analyses its convergence based on feedback and determines the Adaptive Component Status. An example of a Tracking Controller is the mixture of controllers example by Waegeman et al. [2013].
- Pattern Generators output a pattern and are not necessarily driven by a target or other input. An example of a Pattern Generator is given in the the mixture of controllers example by Waegeman et al. [2013].
- Sequencers can be understood as special case of Pattern Generators with conditioned iteration of a Dynamical System. It can be based on the Adaptive Component Status of other Adaptive Components, e.g., to wait for the finalization of sequences of a motion.

5.1.5. Criterion and Adaptive Component Status

If an Adaptive Component contains a Criterion, it can report its status similar to Adaptive Modules and Dynamical Systems. The semantics of the Adaptive Component Status relate to the status of the actual motion and can be in task space or configuration space. The concept of the Criterion is similar to the exit conditions formulated for the manipulation primitives in [Kröger et al., 2004]. The Adaptive Component Status reported by a Criterion is:

1. **not converged**: the motion is not finished yet, the task space or joint space parameters are outside the threshold given by the Criterion. This happens for example when the status of the contained Adaptive Module is "*transient*" or "*converging*".
2. **converged**: the motion is finished, the task space or joint space parameters are within the threshold given by the Criterion. This is often similar to the contained Adaptive Module being in its *reached* status, but can differ depending on the actual Criterion.

Example: Reaching Controller
The Reaching Controller to control the placement of the left fore foot consists of an Adaptive Module and a Criterion to check for convergence. The Adaptive Module operates in the joint space, yet, goals are provided as task space positions. Therefore the Reaching Controller requires an Inverse Kinematics Mapping to connect the Adaptive Module to its architectural context.
The Criterion also operates in the joint space, which requires the same Mapping. In the concrete example, the Criterion has properties determining the threshold on the Joint Angles determining when the motion is considered converged.

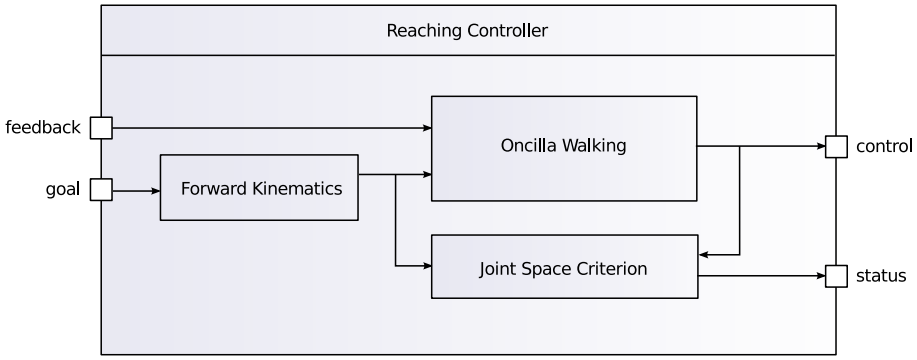


Figure 5.10.: Internal block diagram of the Reaching Controller to control the placement of the left fore foot.

5.2. Behavioral Models

Expressing the domain examples in the presented concepts is a first qualitative evaluation and shows that the concepts are indeed capable to cover the functional variety of the surveyed motion architectures regarding their structural aspects [Nordmann et al., 2015]. Nordmann et al. [2015] discuss, however, that behavioral aspects are often missing in the system representations surveyed in the domain analysis. There, behavior is largely not explicitly represented, but often hidden in the source code of sequencing components. Behavioral aspects, however, are equally important and necessary to specify the coordination and combination of different motion primitives (**FR3**) and therefore to formalize entire motion primitive architectures and generate entire executable experiments.

This section therefore introduces further models dedicated to system-level coordination [Biggs et al., 2010], which is already covered by several well-known models and languages [Nordmann et al., 2014]. Widely used in robotics, are models based on Harel state charts [Harel and Politi, 1998], such as the UML Statechart [OMG, 2010] and the State Chart XML (SCXML) [Barnett et al., 2013] from the World Wide Web Consortium (W3C). These models incorporate Harel's notions of hierarchical and parallel states and are thus suitable for general-purpose state machines. Since a large corpus of

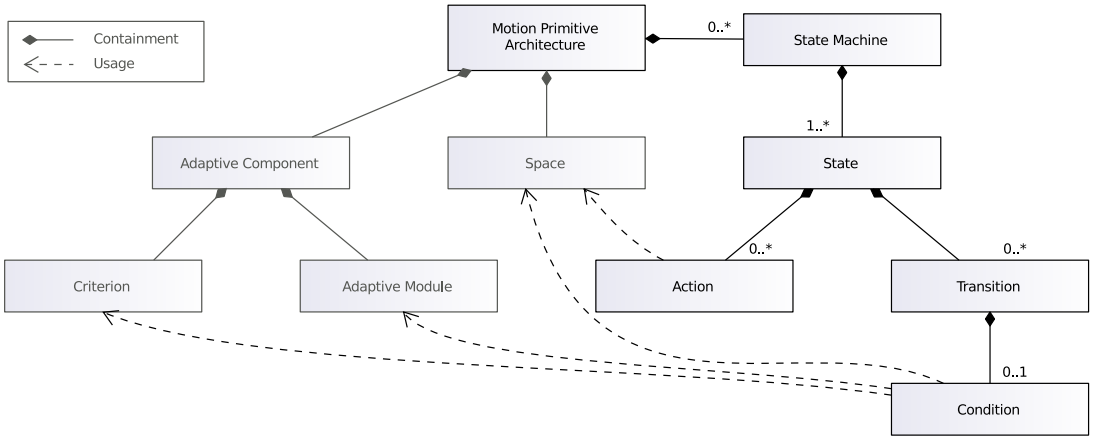


Figure 5.11.: Metamodel, focused on the behavioral aspects for system-level coordination. The metamodel shows the concepts of the behavioral model (black) and their relations to concepts of the structural model (gray).

work exists for these models, they serve as a base for the behavioral model introduced in this work and are extended by domain-specific (motion primitive-specific) extensions to express the variety of behaviors covered by the domain analysis, cf. Fig. 5.11.

The top-level abstraction of the Harel state chart and therefore basis for the behavioral model in this work is the `State Machine` (sometimes also "state chart"). It contains a number of `States` and the transitions between them as well as an initial `State` that is transitioned to when starting the `State Machine`. `States` define entry and exit `Actions` and a set of `Transitions` to other `States` that define how it reacts to events. Special cases of `States` are `Parallel States` and `Composite States`. `Parallel States` encapsulate a set of child states, which are simultaneously active when the parent element is active. `Composite States` contain further substates and, like the `State Machine`, have an initial `State` that is transitioned to at when entering the `Composite State`. `Transitions` are triggered by events and can be conditionalized via guard `Conditions`. They may contain `Actions`, which are executed when the transition is taken. The `Condition` defines weather or not a certain `Transition` to another `State` is followed. For a more detailed and complete description of the models please refer to [Harel and Politi, 1998, Barnett et al., 2013, OMG, 2010].

Although the state chart model serves as a good and well-known basis for the behavioral model of the targeted domain, further motion primitive extensions and models are introduced in the following to express the variety of behaviors covered by our domain analysis.

5.2.1. Actions

`Actions` define what is happening inside a state. This is a first domain-specific extension point to the rather generic state chart concepts. Apart from generic actions like log messages, domain-specific actions like changing the lifecycle state of an `Adaptive`

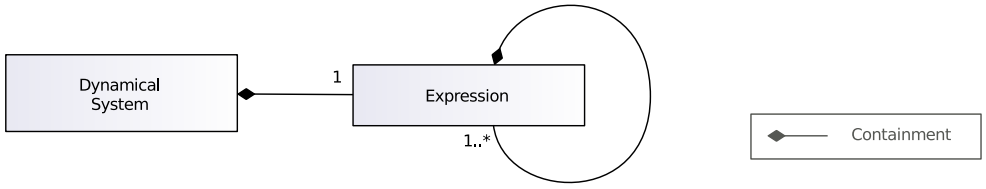


Figure 5.12.: Metamodel, focused on the algorithmic aspects.

Module or triggering execution of a motion primitive.

Another required Action is to provide data for a certain Space, which can be used for example to provide goals for a motion primitive (resp. its corresponding Adaptive Module.

5.2.2. Conditions

The domain analysis revealed several motion-specific synchronization mechanisms to trigger composition and sequencing of motion primitives: The Adaptive Component Status introduced in Section 5.1.5 represents the status of the performed movement: i) *converged*: The reference of the movement is reached (in case of a goal-directed movement) or tracked (in case of a periodic movement) and ii) *not converged*: Movement is still ongoing.

A further domain-specific condition is based on the robot and especially relevant in the human-robot interaction and programming-by-demonstration (PbD) examples, cf. [Nordmann et al., 2012a] It allows system coordination based on the movement and learning states of motion primitives, as well as triggering the execution of motions and learning steps.

Coordination between the state machine and components, often hidden in code, implemented as state machine events and event-handlers, is now explicit in the DSL specification.

5.3. Algorithmic Models

The concepts to model the algorithmic part of the targeted domain are based on Dynamical Systems as motivated in Chapter 3. Dynamical Systems are based on a strict and well-defined mathematical framework and can be expressed by a mathematical expression as depicted in Fig. 5.12.

Fig. 5.13 details further how these Expressions look like. Expressions can be unary or binary. Examples for Unary Operators are Parentheses, Negation or trigonometric functions. Examples for Binary Operators are Addition, Subtraction, Multiplication, Division, and Exponentiation. An Expression can also be an Assignment that assigns the result of an expression to a Variable.

To be usable within the overall framework, possible Variables of the Expressions are the Inputs and Outputs of the surrounding Adaptive Module with the restriction

that only Outputs, not Inputs, can be the left side of an Assignment. Fig. 5.14 shows an example expression from a Dynamical System of the running example that calculates the Output of an Adaptive Module based on a calculation with its Inputs.

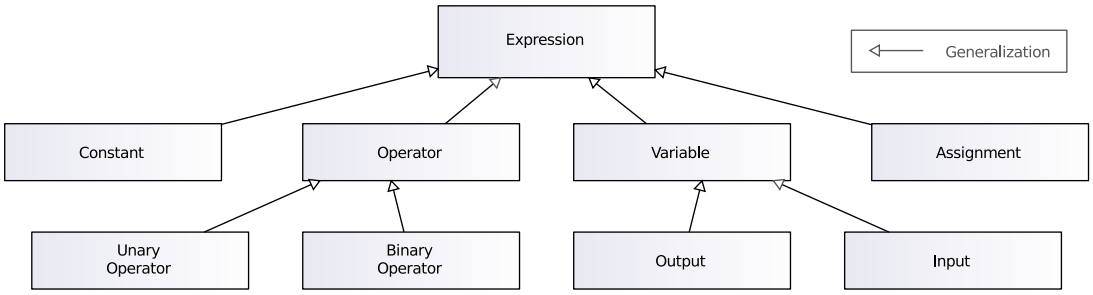


Figure 5.13.: Specializations of the Expression concept.

5.4. Discussion

Atkinson and Tunjic [2014] argue that there is a *"growing consensus on the need to move to comprehensive, view-based approaches"* to cover the different aspects of complex systems. The number of viewpoints should be minimal and viewpoints should be orthogonal [Atkinson and Tunjic, 2014]. The introduced metamodel comprises models to describe motion primitive architectures from three orthogonal viewpoints: The *structural* models revolve around the main concept of Adaptive Modules that represent motion primitives. The *behavioral* models use the widely used state chart model as a base and add further motion primitive specific extensions to allow easy specification of certain common actions and synchronization mechanisms found in the domain. The *algorithmic* models are based on mathematical Expressions and Operators and therefore open up to their full expressiveness. The algorithmic model references Inputs and Outputs of Adaptive Modules to integrate with the overall motion primitive architecture. The choice of the three viewpoints is conforming to the state of the art and other approaches in the field, e.g., V^3CMM [Alonso et al., 2010] where this partitioning is quite explicit. The different views have only minimal overlap in conformance with Atkinson and Tunjic [2014] as they describe different concerns of the same system.

The models introduced in this chapter are largely platform independent models (PIMs) to allow formulation of motion primitive architectures independent from a target technology (**NFR6**). Targeting these models to executable artifacts is introduced in Chapter 6 and Chapter 8.

The functional models introduced in this chapter address several of the requirements formulated in Section 4.2. The abstractions of Adaptive Module with its specific lifecycle, inputs, outputs, and status make the representation of motion primitives with its learning and execution phases explicit and compact (**FR1**, **FR2**). The behavioral models, especially with their domain-specific extensions allow dynamic coordination

and combination of motion primitives (**FR3**). The introduced concepts of `Spaces` and `Mappings` allow explicit formulation of dependencies to external perception (**FR4**) and higher cognitive layers (**FR5**). The concept of `Adaptive Components` with the potential input and output `Mappings` allows connecting these even when their `Space Types` are not agreeing, under the condition that forward and/or inverse `Mappings` exist.

Example: Reaching Dynamics
A simple Dynamical System from the running example expresses the dynamics of the leg joints to reach a certain goal in the joint space. Simple exemplary dynamics that move the joint angles from its current position 10% of the distance to its goal configuration could be written with the equation $x_t = x_{t-1} + 0.1 * (g - x_{t-1})$, with x_t being the current state, x_{t-1} the previous state and g being the goal configuration.
Fig. 5.14 shows how this equation translates to the introduced algorithmic model in the form of a tree of expressions. A Constant, an Assignment, and several Unary Operators and Binary Operators express the above formula. The new state x_t is assigned to the Adaptive Module's control Output "ctrl", the previous state x_{t-1} is read from the Adaptive Module's feedback Input "fdb", and the goal configuration g is read from the Adaptive Module's goal Input "goal". Note, how the Adaptive Module operating in joint space can be used, even when the foot goal is (typically) given in task space coordinates, by simple adding an Inverse Kinematics Mapping to the goal Input in the surrounding Reaching Controller, as exemplified later.

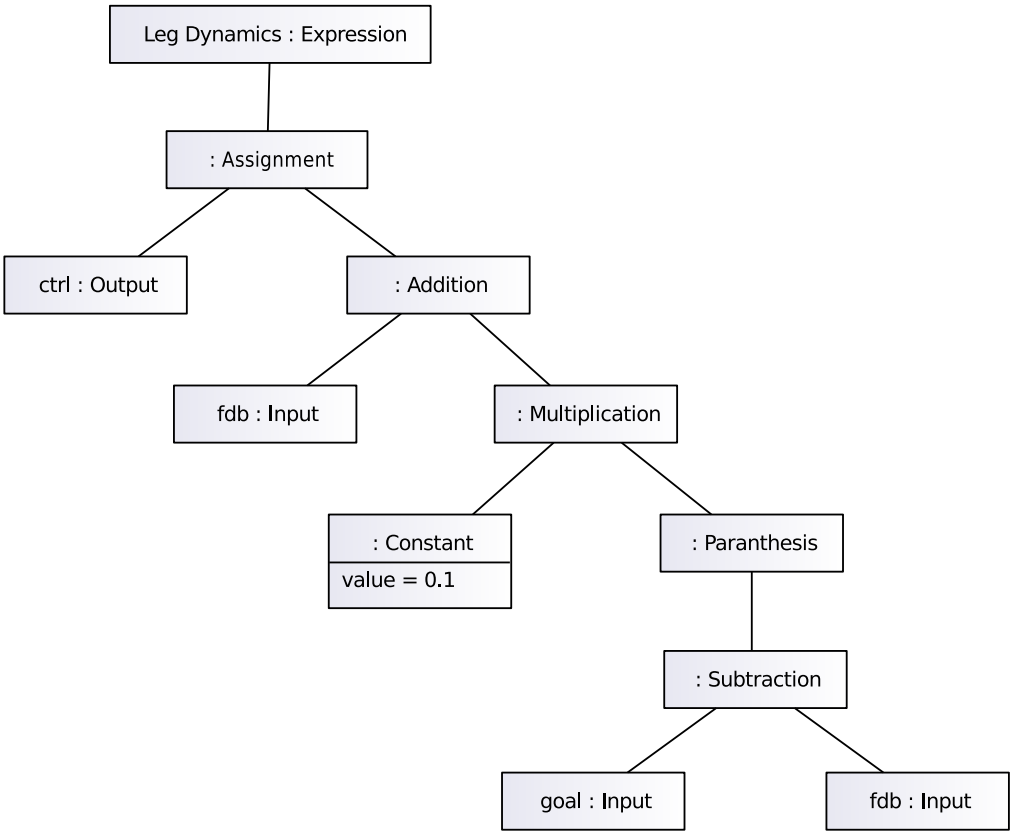


Figure 5.14.: Object diagram of the expression $ctrl = fdb + 0.1 * (goal - fdb)$ to express the leg dynamics in the running example.

Chapter 6.

Language Modularization and Design

> "Beware of the Turing tarpit in which everything is possible but nothing of interest is easy."
> – Alan J. Perlis

While the models introduced in Chapter 5 provide a clear semantic framework to specify motion primitive architectures, they need a concrete, e.g., textual or graphical, syntax to be accessible for a developer. Apart from practical consideration, this is a valuable process since *"the process of defining the language actually helps the architecture/development team to better understand, clarify, and refine the architectural abstractions, as the language serves as a (formalized) ubiquitous language that lets you reason about and discuss the architecture."* [Völter et al., 2013]

This chapter details modularization and design of the domain-specific languages (DSLs) for the introduced metamodel. A formal DSL design consists of a specification written using a semantic definition method. The most widely used formal notations for several years include regular expressions and grammars for syntax specifications, and attribute grammars, rewrite systems, and abstract state machines for semantic specification [Mernik et al., 2005]. In recent years, however, a class of powerful tools became available that allow convenient definition, reuse and composition of DSLs together with their integrated development environments (IDEs). This class of tools is called *language workbench*, a term introduced by Fowler [2005], and is used in this work. A language workbench explicitly subscribing to the idea of language modularization, extension, and composition (LMEC) as targeted by this work is Jetbrains Meta-Programming System (MPS) [Jetbrains.com, 2003] that was used for modularization and design of the DSLs introduced in this chapter.

Following the idea of LMEC, the metamodel is decomposed into a set of DSLs with the different concerns of the motion primitive architecture domain realized as a separate DSL. The DSLs are composed with the composition methods introduced in Section 2.2.1.

Section 6.1 further introduces the concept of language workbenches, and discusses the tool of choice in this work, MPS, since it not only provides the meta-metamodel (M3) for the metamodel and DSLs of this work, but also has influence on the design of the languages and the toolchain discussed in the course of this thesis. Section 6.2 introduces the concrete language modularization of the introduced metamodel with different modularization and composition methods. Section 6.3 provides an overview

on design dimensions of DSLs, e.g., their concrete syntax and constraints, and discusses the concrete design of the modularized languages introduced in Section 6.3. Section 6.4 shows the model-to-model transformations (M2Ms) between the modularized languages to map the motion primitive architecture models to more platform specific models (PSMs).

6.1. Language Workbenches

"Language workbench" is a term introduced and popularized by Fowler [2005]. It describes a class of software development tools designed to define, reuse, and compose domain-specific languages together with their integrated development environment. Language workbenches therefore follow the idea of language-oriented programming and usually support concerted specification of the metamodel, editing environments for the DSL, and its execution semantics, e.g. through interpretation and code generation.

Several language workbenches have been developed over the last years, such as MetaEdit+ [Tolvanen and Rossi, 2003], Monticore [Krahn et al., 2001], Jetbrains Meta-Programming System [Jetbrains.com, 2003], and Spoofax [Kats and Visser, 2010]. One of the most popular language workbenches is Eclipse EMF Xtext (Xtext) [Bettini, 2013], which has a large community and is accompanied by a wide range of tools for textual and graphical editors, transformations, code generation and visualization under the umbrella of the Eclipse Modeling Project (EMP). EMP is well adopted in the robotics DSL community and its tools, e.g. Xtext, are used by most of the robotics DSL approaches surveyed by Nordmann et al. [2014].

The language workbench that is used for development of the motion primitive DSLs in this work is Jetbrains Meta-Programming System [Jetbrains.com, 2003]. It has strong support for composable language definition, DSLs can be extended and embedded, which supports the LMEC approach motivated in the design process proposed in Chapter 4. MPS solves potential grammar ambiguity issues by working with a projectional editing approach. This means that the main definition of a system is held in a model, not in source code, and it is edited through textual or graphical projections of this model, as shown in Fig. 6.1. Classical parser-based approaches treat the language code as the primary artifact and the abstract syntax tree (AST) is created by parsing the language code. When working with projectional editing, the AST is directly manipulated, and only its text-like or graphical projection is only used for looking at the AST. For projectional editors it is relatively easy to define several notations for the same language concept by changing the projection rules. Therefore, different notation for different concerns can be used for different stakeholders [Völter et al., 2013].

In addition to its support for LMEC and projectional editing, its wide support for different well-integrated language design aspects was a reason to choose MPS for this work.

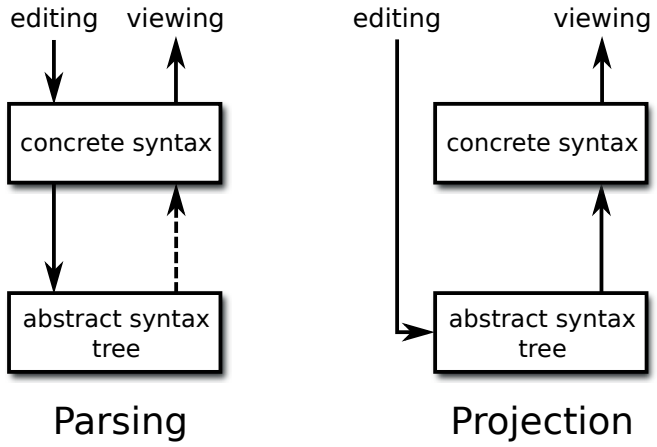


Figure 6.1.: Classical parser-based approach (left) and projectional editing (right). In the parser-based approach, the user interacts only with the concrete syntax. In projectional editing, the user manipulates the AST directly and observes its projection [Völter et al., 2013].

6.2. Language Modularization

The three main concerns/viewpoints of the domain are structural, behavioral and algorithmic, as introduced and discussed in Chapter 5. This is therefore a natural way of decomposing the metamodel into languages. However, an important motivation for LMEC is to foster reuse of the languages, which is one of the non-functional requirement for this work (**NFR1**). Not to jeopardize composability and potential reuse of the languages, a composition approach is necessary that does not introduce unnecessary dependencies between these languages, i.e. languages of the different concerns should be reusable independent of the other concerns.

Fig. 6.2 shows how the metamodel proposed in Chapter 5 is decomposed into mainly five DSLs developed in this work, and an additional DSL for `Data Types`.

Motion Primitive DSL The Motion Primitive DSL is the main DSL to express the structural aspects of motion primitive architectures. It is a textual DSL and comprises most of the concepts defined in Section 5.1. It is therefore an architecture DSL (ADSL) in the terms that it *"expresses a system's architecture directly.* Directly *means that the language's abstract syntax contains constructs for all the ingredients of the conceptual architecture. The language can hence be used to describe a system on the architectural level without using low-level implementation code, but still in an unambiguous way."* [Völter et al., 2013] This is true for the structural aspects of the architecture.

The main functional aspects expressed with Motion Primitive DSL are motion primitive and the necessary `Spaces` to connect them with each other, to external perception, to external higher cognitive layers and to the robot. The Motion Primitive DSL

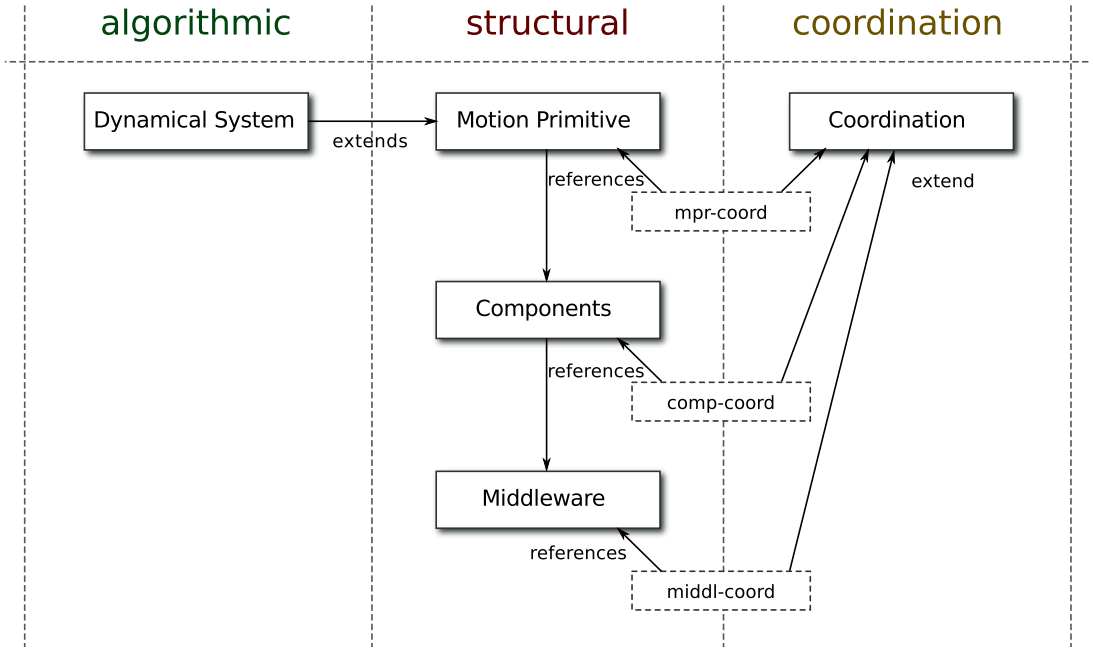


Figure 6.2.: The language modularization proposed in this work and their dependencies.

is supposed to be technology and platform independent; therefore, all platform and technology dependent features are omitted in this language.

Dynamical System DSL The Dynamical System DSL constitutes the "business logic DSL" [Völter et al., 2013] and encompasses the algorithmic models, i.e. `Dynamical Systems`. It introduces one new `Dynamical System` concept that specialized the original `Dynamical System` and contains a single `Assignment Expression` in conformance with the algorithmic model discussed in Section 5.3, as well as validation and type system rules for convenient integration.

Component DSL The Component DSL describes structural aspects of component-based software engineering (CBSE), namely `Components`, `Ports`, `Classes`, and `Methods`. It therefore models the software architectural aspects of a CBSE system and allows arbitrary computational content inside the components. The core metamodel of the language conforms to the main abstractions of common architecture description languages (ADLs) as discussed in Section 3.3.1. The main concepts of the Component DSL are depicted in Fig. 6.3.

`Ports` of `Components` are strongly typed and communicate with each other when `Output Port` and `Input Port` use the same `Scope`, an identifier for a common communication channel. `InputPorts` of the components can be configured in terms of their input buffer size and buffering strategy, e.g., to keep always the latest received item

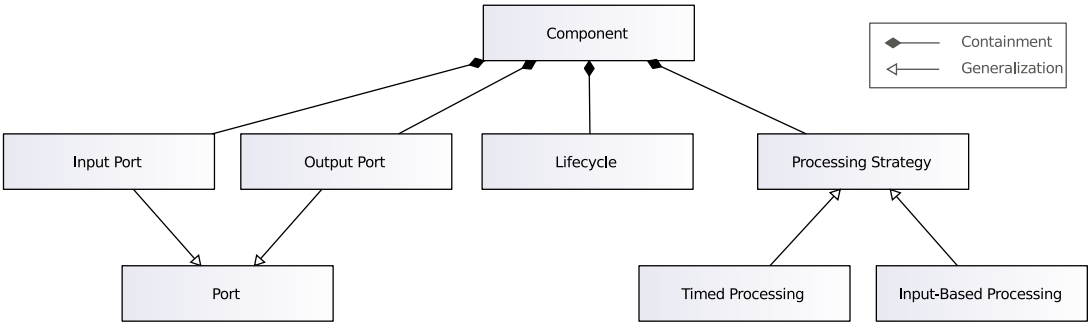


Figure 6.3.: Metamodel of the Component DSL, focused on the `Component`.

buffered.

So-called `Processing Strategies` allow specifying input-based or timing-based activation semantics of the `Components` to model the different dynamic execution and interaction behaviors found in the domain analysis.

Middleware DSL The Middleware DSL describes structural aspects of systems in terms of middleware concepts, more precisely systems based on the event-based middleware Robotics Service Bus (RSB), that organizes the communication of system parts via publish/subscribe and remote procedure calls (RPCs) [Wienke and Wrede, 2011]. The main concepts of the Middleware DSL are `Processes`, bus `Participants` and its specializations, `Publisher` and `Listener` as well as `RPC Server` and `RPC methods`.

Coordination DSL The Coordination DSL covers the behavioral aspects of the motion primitive architecture metamodel, coordination of the motion primitives. The main structural element of the language, and its root concept, is the `State Machine` that is a container for all the `States`, `Actions`, and `Transitions` that define the system-level behavior.

Primitive Coordination DSL The pure Coordination DSL does not simplify specification of behavioral motion primitive architecture aspects for the domain user or raise the expressiveness compared to general purpose modeling languages. For this reason, Primitive Coordination DSL is introduced that *extends* the Coordination DSL by domain-specific coordination aspects introduced in Section 5.2.

The Primitive Coordination DSL extends the Coordination DSL by several motion primitive specific `Actions` and `Conditions`. Motion primitive specific `Actions` are primarily `ChangeState`, to change the state of an `Adaptive Module`, and `PublishToSpace`, to publish a certain data item to a `Space`, e.g., to provide a goal for a movement. Motion primitive specific `Conditions` added with the Primitive Coordination DSL are primarily `StateChanged` to react to lifecycle change of a `Adaptive Module`, `ConvergedCondition` to react to a `Adaptive Component` being converged,

and `RobotAffected` to react to the robot being touched, e.g., in a physical human-robot interaction (pHRI) scenario.

Types DSL The Types DSL provides a collection of over 150 `Data Types` that are typically found in robotics systems, e.g., `Joint Angles`, `Joint Torques`, `Pose`, `Translation`, `Rotation`, and `Forces`. Its types are originally based on the domain analysis discussed in Section 3.3, taking input especially from the control feature models (cf. Section B.1.1) as well as the input and output specification of the adaptive modules survey (cf. Section B.2).

6.2.1. Language Composition

Integrating the motion primitive specific coordination aspects introduced in Section 5.2 into the Coordination DSL would mean *referencing* the Motion Primitive DSL (in terms of Section 2.2.1), therefore imposing a direct dependency which hinders easy reuse and thus does not comply with **NFR1**. Instead, Coordination DSL *reuses* Motion Primitive DSL (in terms of Section 2.2.1) for which an adapter language Primitive Coordination DSL is introduced.

To allow this it *extends* the Coordination DSL and *references* the Motion Primitive DSL, see Fig. 6.4. A `State Machine` with motion primitive extensions is expressed as a `State Machine` and an `Adaptive Module` Circuit. `State Machines` and `Adaptive Module` Circuits are expressed with the two different languages Coordination DSL and Motion Primitive DSL respectively. The referencing Primitive Coordination DSL depends on the referenced Coordination DSL and Motion Primitive DSL because concepts in the Primitive Coordination DSL references extend concepts from the Coordination DSL and reference concepts from the Motion Primitive DSL. In this case, Primitive Coordination DSL is the *referencing language*, and Coordination DSL and Motion Primitive DSL are the *referenced languages*. In this case, Primitive Coordination DSL serves as an *adapter language* for language *reuse* in terms of Völter et al. [2013] as it allows using the independent languages Coordination DSL and Motion Primitive DSL to be used together without introducing an explicit dependency between them. This also allows both languages to be used alone as well as in different contexts.

Similarly, the Component Coordination DSL provides extensions to the Coordination DSL to coordinate systems modeled with the Component DSL and the Middleware Coordination DSL provides extensions to the Coordination DSL that integrate with the Middleware DSL.

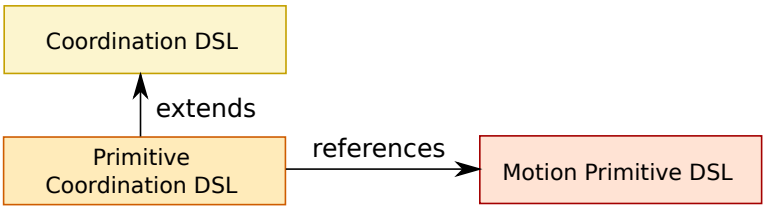


Figure 6.4.: Language composition of the Primitive Coordination DSL.

All of these languages need to be combined with the Coordination DSL to be able to express the behavioral aspects of motion primitive architectures as proposed in Chapter 5. The Coordination DSL encompasses the behavioral models, i.e. `State Machines`. However, `State Machines` are a rather generic and widely used concept, also useful outside the motion primitive domain. This section exemplifies the language modularization, extension, and composition proposed in this work for some of the involved languages.

The Dynamical System DSL in its current implementation extends the Motion Primitive DSL, introducing a specialization of the `Dynamical System` concept. This currently makes the Dynamical System DSL not usable independent of the Motion Primitive DSL. This could easily be changed, though, with the language modularization concepts discussed in Section 2.2.1.

The Motion Primitive DSL has a dependency to the Types DSL so that all its `Data Types` can be used for specification of the `Space Types`. The Component DSL and the Middleware DSL also have a dependency to the Types DSL for specification of their `Port` types and `Participant` types respectively.

A further means of language composition is annotations, which technically provide a separate viewpoint, which are often used for technical or transformation controlling aspects and can be added to the model by different stakeholders. In this work, the Component DSL and Middleware DSL provide annotations for platform-specific aspects. These annotations are used in the code generation, discussed during transformations, detailed in Section 6.4. They provide hints for the code generators regarding the mapping of the specified motion primitive architecture to a certain platform, as exemplified in Section 9.1.2.

With this modularization, different concerns of motion primitive architectures are separated in a way that generic concerns can be used without introducing dependencies to motion primitive language and aspects, e.g., outside the motion primitive domain. By using the adapter languages, generic languages and motion primitive specific languages can be integrated to realize the domain-specific models and aspects of the metamodel specified in Chapter 5.

6.3. Language Design

Fig. 6.5 shows language design aspects that are supported by MPS to specify a DSL. The *structure* aspect defines the abstract syntax of the language with its concepts, properties, and relations. This corresponds to the metamodel. The *editor* aspect adds a concrete, e.g., textual or graphical, syntax to the language by defining the projection rules for the elements specified in the language structure aspect. The *type system* and *constraint* aspect define the static semantics of the language by means of scoping, value restrictions, typing rules and type checks. Finally, the *transformations* aspect defines the execution semantics of the language through model-to-model transformations (M2Ms) and model-to-text transformations (M2Ts). Each of these aspects is specified in MPS by dedicated, built-in DSLs, such as the *Structure Language*, the

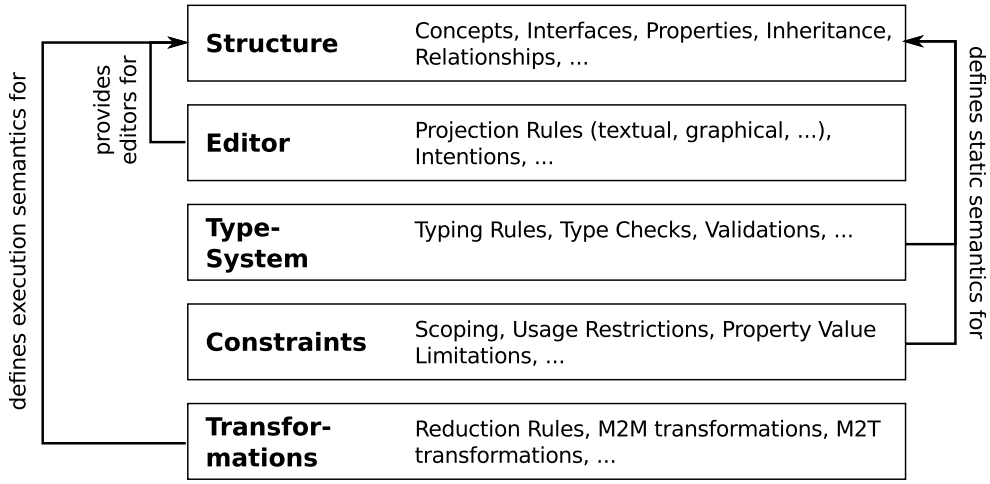


Figure 6.5.: Language design aspects in MPS [Völter et al., 2014].

Typesystem Language, etc.

After modularization of the metamodel into DSLs the abstract syntax provided by the metamodel, resp. the MPS language structure, has to be extended by a concrete syntax. Karsai et al. [2009], Mernik et al. [2005], Völter et al. [2013], Nguyen et al. [2014] distinguish the following essential design dimensions of DSL development, which are the basis for discussing the DSLs in the following sections:

Expressiveness Expressiveness (sometimes also "expressivity") is often a main concerned of DSLs and one of their fundamental advantages: their increased expressiveness over general-purpose languages (GPLs). This typically means that programs are shorter and their semantics are easier to access by processing tools. Programs expressed using a DSL can and should be significantly more concise than expressed in GPLs.

Coverage A DSL can cover its intended domain entirely, i.e. for each program relevant to the domain a program can be written in this DSL. This often has the risk of the DSL to become too general and therefore resulting in bigger programs and losing the advantage of increased expressiveness. In practice, many languages do not fully cover their respective domain [Völter et al., 2013].

Semantics and Execution Völter et al. [2013] distinguishes semantics between static semantics and execution semantics. *Static semantics* are often established by means of language constraints and type-checking rules. Constraints check properties of a model and can be validated to check if a model is structurally and syntactically correct, e.g. basic constraints and typing rules. *Execution semantics* denote the semantics of a DSL either when transformed in another DSL, or when interpreted/executed.

Transformations (M2M or M2T) define the execution semantics of a DSL by mapping it either to other DSLs or eventually to the actual execution infrastructure, often the targeted GPLs. The M2M transformations are discussed in this chapter in Section 6.4; M2T transformations are discussed later in Chapter 8 in terms of code generation.

Support Platforms / Integration Once the executable artifacts are generated, the support for target platforms and integration into existing environments, tools, and platforms becomes relevant. "Platform" in this context means the technical execution context, so the software framework, and all additional tools or libraries necessary to use the generated artifacts. The DSL needs to be able to integrate with other parts of the development process, and ideally supports integration with other languages and platforms without a lot of effort.

Concrete Syntax Similar to using the natural abstractions of the stakeholders, (re)using the natural or existing, established notations in a DSL is a key point for it's efficiency and adaption by the users. A good notation makes expression of common concerns simple and concise and provides sensible defaults. It is acceptable for less common concerns to require a little more verbosity in the notation. Concerns to be addressed when designing a syntax are good *writability*, *readability*, *learnability*, and *effectiveness*. I.e. the syntax can be written efficiently, it is concise, can be read and understood easily to effectively express typical domain problems and produce good solutions, and provides the necessary context to understand the code.

Some exemplary classes of concrete syntax are (purely) *textual*, well suited for detailed descriptions, *graphical*, good for describing relationships effectively, *symbolic*, e.g., mathematical expressions using a symbolic notation, or *tables and matrices* that can be useful for collections of data items, or two independent dimensions of data.[Völter et al., 2013]

The following section discusses the introduced language design aspects along the five MPS language aspects shown in Fig. 6.5. For the sake of brevity, all aspects are discussed with selected examples from the Motion Primitive DSL, the Dynamical System DSL, and the Coordination DSL, as well as the Primitive Coordination DSL.

6.3.1. Structure

In MPS, DSLs are designed "metamodel-first", by first defining the language structure and then adding editors, type system, constraints, and transformations. Therefore, for designing the languages, the different concerns of the metamodel were converted to the language structure of the different DSLs according to the modularization discussed in the previous section. Since the structure of the proposed DSLs is defined with the *Structure Language* of MPS, this gives a full picture of the levels of modeling shown in Fig. 2.1: The structure base language of MPS constitutes the meta-metamodel (M3) level in this work by providing the models the DSLs have to conform to.

```
interface concept AdaptiveModuleIF extends INamedConcept
                                            AdaptiveIF
                                            Triggered
                                            HasStateIF
                                            HasProperties

  properties:
  << ... >>

  children:
  ds : DynamicalSystemIF[0..n]
  goal : Input[0..1]
  feedback : Input[0..1]
  cfg : Input[0..1]
  speed : Input[0..1]
  phase : Input[0..1]
  train : Input[0..1]
  statein : Input[0..1]
  control : Output[0..1]
  stateout : Output[0..1]

  references:
  << ... >>
```

Figure 6.6.: Exemplary language structure definition in MPS for the Adaptive Module interface, defining its relation to further interfaces (upper right) and the cardinality and roles of contained concepts (lower left).

Definition of the language structure is exemplified mainly with the Motion Primitive DSL. The main root concept of the language is the Circuit that is a container for the Adaptive Components, Adaptive Modules that represent the motion primitives as well as all Spaces necessary for the desired experiment or application. A further root concept is the Space Type that specifies which Spaces are valid within the Circuit.

Fig. 6.6 shows how these concepts and the metamodel look like when implemented with MPS. It shows a screenshot of the MPS IDE with the language tree and all its aspects on the left side and the editor view on the right side. The language tree is unfolded showing some of the main concepts of the language, e.g., the Adaptive Module, the specialized Adaptive Components, Spaces, on left side. The right side shows the structural definition of an interface for the Adaptive Module concept, defining child relation to Inputs, Outputs, and Dynamical Systems, as defined in Section 5.1. All other concepts are specified in a similar fashion. The main structural and modularization units of this language are Adaptive Components and Adaptive Modules. They are partitioned in Circuits and specify the main structure of the motion primitive architecture.

Further important structural elements of the language are Space Types and Spaces to model the relation and communication between the Adaptive Components and Adaptive Modules. Space Type is a further root concept of the language to not bind it to a specific Circuit, but make it available for usage in several Circuits. This allows for example to define and share certain robot-specific Space Types, e.g., Oncilla Leg

```
<default> editor for concept AdaptiveModule
  node cell layout:
    [-
    primitive { name }
      strategy : % strategy %
      in : % goal % , % feedback % , % cfg % , % speed % , % phase % , % statein %
      out : % control % , % stateout %
      ds : (- % ds % /empty cell: <default> -)
      # properties #
      <constant>
    /folded cell: [- primitive { name } -]
    -]
```

Figure 6.7.: Language editor definition in MPS, showing the definition of a textual editor for the Adaptive Module with its child concepts and properties.

Angles, KUKA LWR torques, etc., each with the robot specific types and dimensions, which allows to offer robot-specific packages for convenience and reuse. Spaces can be added and specified by a Space Type. Fig. 6.8b shows specification of two different Spaces of the same type.

6.3.2. Editor

After creating the structure of the language, the concrete syntax is added to the language with MPS' *Editor Language*. All of the DSLs are textual or at least initially developed as textual DSLs. This is because good textual DSLs can be very effective. Purely textual DSLs usually integrate very well with existing development infrastructures and are usually well suited for detailed specification of anything that is algorithmic or generally resembles traditional GPL like code.[Völter et al., 2013] *"Textual representations for example usually have the advantage of faster development and are platform and tool independent whereas graphical models provide a better overview and ease the understanding of models."* [Karsai et al., 2009] However, since the introduced DSLs are developed within the projectional editing workbench MPS, introducing additional graphical projections to allow graphical editing of the DSLs is explicitly supported. This is not detailed in this work but has been prototypically implemented for the Coordination DSL where the graphical notation is one of the natural natotions.

Fig. 6.7 shows how a textual editor for a concept is designed, again with the example of the Adaptive Module. The "node cell layout" specifies the textual layout of the concept editor, with the bold phrases ("primitive", "strategy", etc.) being fixed strings, strings in braces ("name") being wildcards for concept properties, strings enclosed with the percent sign ("%strategy%", "%goal%", etc.) being wildcards for other concept editors, and strings enclosed with the number sign ("#properties#") referring to other editors. MPS also allows defining the layout of the "folded cell", i.e. a limited and compact representation of the concept so that it can be folded in a more complex system specification for the sake of clarity.

Fig. 6.8 shows how this looks like for in the running example, showing a snippet of the DSL code to edit the Reaching Controller with an Adaptive Module and a Dyna-

mical System inside. The snippet also shows the feature of the folded cell. The goal input mapping, the Oncilla Inverse Kinematics, and the Criterion, the Translatory Criterion, are not fully visible with all its properties, but folded and just represented with their concept name ("Oncilla Inverse Kinematics") and instance name ("invkin") (the folded cell).

A main aspect of the language is to raise its effectiveness of expressing structural motion primitive architectures compared to expressing them with GPLs. One of the language aspects to achieve this is by allowing compact specification of Adaptive Components and Adaptive Modules, e.g. by simple selecting the processing strategy and thereby configuring its execution scheme. Fig. 6.8a shows a snippet that specifies a motion primitive, it's embedding in the architectural context by surrounding it with a Reaching Controller (an Adaptive Component), and defining its inner dynamics (the Dynamical System). The motion primitive is specified to be computed and to generate its next output every time the overall motion primitive Circuit is stepped.

The design of the textual editor, the concrete textual syntax of the language, should follow the natural notation of the domain and the domain experts. The domain analysis, however, did not show a natural notation in the domain, since most of the motion primitives are defined in GPL code. If any, there is the mathematical notation of expressing a motion primitive by its dynamics as realized with the Dynamical System DSL. For the Motion Primitive DSL the natural notation is therefore mainly based on using the terms that occur in the domain, e.g., primitives, inputs, outputs, mappings, criterions, etc.

Another means to raise the effectiveness is editor support. The skeleton of a motion primitive, shown in Fig. 6.8a, for example is created every time a new primitive is created in the language, so that the user only has to fill out the remaining blanks and variation points. Fig. 6.8 shows a DSL example with several of the introduced concepts involved. The snippet shows an Adaptive Component (red) with a motion primitive resp. an Adaptive Module, Inverse Kinematics Mapping, a Criterion, together forming a Reaching Controller to reach a certain pose and check convergence.

The expressiveness of the Dynamical System DSL is largely driven by the easy integration of the formula expressing the dynamics of a motion primitive with its Inputs and Outputs. Tying data streams of a complex system to algorithmic code is usually a tedious task in GPL code, but very compact in the Dynamical System DSL while still being formulated on an abstract level and therefore platform and technology independent. Its notation is the natural notation of the domain expert, mathematical expressions, who can easily specify its connection to the surrounding system, yet do not need to be concerned with the technical details that realize its proper execution and technical integration.

The editor additionally provides context help, as shown in Fig. 6.9, and can validate the expressions based on type system and inference rules. Fig. 6.9 shows a DSL example that specifies the internal dynamics of an Adaptive Module by specifying its Dynamical System with the Dynamical System DSL. The Dynamical System DSL is a textual DSL to match the natural notation of Dynamical Systems: mathematical expressions. The snippet shown in Fig. 6.9 expresses the Dynamical System initially

```
Reaching Controller Place Foot {
  primitive joint ctrl
    strategy: timed(samplerate: 1)
    in: goal<leg angles>, fdb<leg angles>, <no cfg>, <no speed>, <no phase>, <no statein>
    out: ctrl<leg angles>, <no stateout>
    ds: <{ctrl} = {fdb} + 0.1 * ({goal} - {fdb})>
    properties: << ... >>
  criterion : TranslatoryCriterion distance from descriptor
  map goal input: Oncilla Inverse Kinematics invkin from descriptor map feedback input: <no map_fdb>
    map control output: <no map_ctrl>}
```

(a) Motion Primitive DSL snippet of a Reaching Controller to reach a certain pose and check convergence.

```
Space left fore leg status <leg angles>
  connection outgoing to fdb of PlaceFoot.jointctrl

Space left fore leg command <leg angles>
  connection ingoing from ctrl of PlaceFoot.jointctrl
```

(b) Definition of two Spaces of the same Space Type for the status and command of the left fore Oncilla quadruped robot (Oncilla) leg.

Figure 6.8.: Examples of the structural motion primitive architecture aspects expressed in the Motion Primitive DSL.

introduced in Fig. 5.14, assigning the return value of a calculation referencing the feedback and goal Inputs to the control Output of the surrounding Adaptive Module.

6.3.3. Type System

The Dynamical System DSL makes use of the type system aspect to covers the algorithmic aspects of the motion primitive architecture metamodel. It describes the Dynamical Systems, the internals of Adaptive Modules that determine its dynamics. The main concept of the language is the Dynamical System, which extends an abstract Dynamical System concept of the Motion Primitive DSL. The intention of this integration with the Motion Primitive DSL is that the mathematical expression to representing the dynamics of the motion primitive can be integrated with the Adaptive Module concept, shown in Fig. 6.9.

Large parts of expressing mathematical expressions with the DSL are already available with MPS' *Expression Language*. This language provides concepts for Expressions, unary and binary, and Assignments. Therefore, allowing expressing Dynamical Systems as mathematical expressions in the Dynamical System DSL and using the Inputs and Outputs of the Adaptive Modules in the expression is rather a matter of integrating those with the MPS language and type system. For this matter, Dynamical System DSL introduced two new concepts, references to Inputs and Outputs: InputReference and OutputReference. Fig. 6.10 shows two rules that incorporate

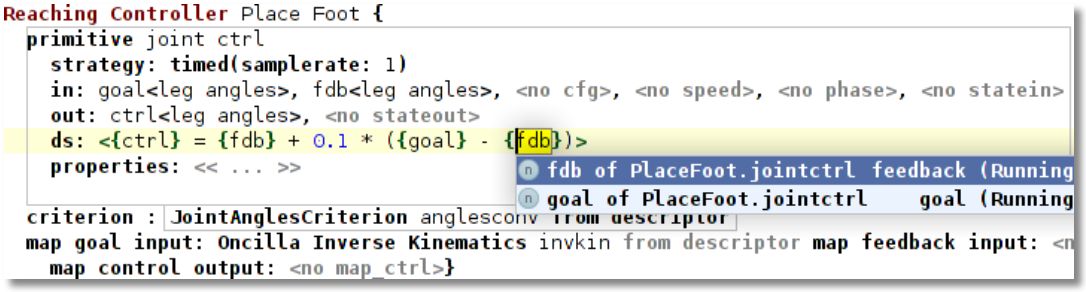


Figure 6.9.: Dynamical System DSL snippet integrated into the Motion Primitive DSL specification of an Adaptive Module. Proper scoping allows only referencing Inputs and Outputs of the surrounding Adaptive Module, as detailed in Section 6.3.4.

these concepts into the MPS language and type system.

The two concepts InputReference and OutputReference *extend* the MPS base concept Expression, so that they can be used at any point in a mathematical expression where any other expression would be valid, e.g., within parenthesis, as term of an addition, factor of a multiplication, etc. This, however, does not allow the inference rule of MPS to check typing of an expression. To allow this, type system rules are used to specify that the type of an InputReference is the same as of an Input, and the type of an OutputReference is the same as of an Output (Input and Output are both strongly typed in the Motion Primitive DSL). In the current version of the language, the types of Inputs and Outputs are specified to be of type double for reasons of simplification, as shown in Fig. 6.10a. This could be extended to be more fine grained based on the Data Type definitions in the Motion Primitive DSL to allow more sophisticated type checks.

Another rule for OutputReference, shown in Fig. 6.10b, specifies that the OutputReference concept is a legal "lvalue", which means that it can be used on the left side of an Assignment operation. This allows a Dynamical System to use Inputs and Outputs in arbitrary expressions on the right side of an Assignment operation, but only a reference to an Output of an Adaptive Module to be used on the left side of an Assignment operation.

The scoping rule for Inputs and Outputs established in the Motion Primitive DSL, only allowing references to Inputs and Outputs *inside* the Adaptive Module they are references from, restrict the usage of Inputs and Outputs (their respective InputReferences and OutputReferences) to only legal ones, as shown by the context help in Fig. 6.9.

6.3.4. Constraints

A further important advantage in terms of support when using domain-specific languages is the possibility to validate its underlying models on a *semantic* level, whereas typically GPL programs can just be validated on a *technical* level. Means to do this

```
rule typeof_Input {
  applicable for concept = Input as input
  overrides false
  do {
    infer typeof(input) :<=: <double>;
  }
}
```

(a) The Input concept of the Motion Primitive DSL is specified to be of double type, so it can be used inside Expressions.

```
concept behavior OutputReference {
  constructor {
    <no statements>
  }
  public static boolean lvalue()
    overrides Expression.lvalue {
    return true;
  }
}
```

(b) OutputReference is integrated into the MPS type system by declaring it a legal "lvalue", which means that it can be used on the left side of an Assignment operation.

Figure 6.10.: Integration of domain concepts with the MPS base language and type system.

are included in the DSLs as constraints and type system as detailed in Section 6.3. These are validated by MPS instantaneously during editing.

An example for constraints in the Motion Primitive DSL is a set of rules that perform checks on ingoing and outgoing connections of Spaces. Since both, Spaces resp. Space Types are strongly typed, rules can check type compatibility of the connections endpoints, the Space and the Input or Output and issue a warning to the user when an incompatibility is detected. Chapter 8 and Chapter 9 exemplify how this looks like from a user's perspective.

A further set of rules checks the semantics of the specialized Adaptive Component concepts, e.g., warns when a Reaching Controller or Tracking Controller doesn't provide a Criterion to check its convergence or does not have a goal Input defined. This is not the case for a Pattern Generator for example that does not necessarily need a goal given and does not necessarily need a Criterion. To not restrict experimenting too much, though, these validation rules issue warnings, not errors, to still allow code generation and testing.

Another important help is scoping, i.e. restricting references to only valid elements of the model. An example where this is realized in the Motion Primitive DSL is referencing of Inputs and Outputs. Input-based Processing Strategies for example specify the Inputs that trigger execution of the Adaptive Modules. This is only valid for Inputs of the same Adaptive Module, not all Input instances of the entire model. The constraint is implemented accordingly in the Motion Primitive DSL, a concrete example is shown above in Fig. 6.9.

The text color of the concepts matches the color-coding of the Motion Primitive DSL, though. This is especially relevant since the Component Coordination DSL and Middleware Coordination DSL introduced in Section 6.2 also add additional Actions and Conditions. They also have their specific color-coding matching the colors of the Component DSL and Middleware DSL respectively, so that it is visually traceable to which domain the specific actions and conditions belong. Fig. 6.11 shows a DSL

```
state Place Foot (final: false)
  actions:
       on entry: publish Translation: z: 0.0 y: 0.4 x: 0.7 to left fore foot goal
       on exit: << ... >>
  transitions:
       -> Backup on PlaceFoot.JointAnglesCriterion converged if <no jexlCondition>
       -> Stop on button {exit} pressed if <no jexlCondition>
```

Figure 6.11.: Coordination DSL snippet with Primitive Coordination DSL extensions to publish a goal to a Space, and transition to the next state on convergence of the corresponding Reaching Controller for this goal.

snippet of a State with motion primitive specific extensions.

Primitive Coordination DSL A more interest case of constraints is realized with the Primitive Coordination DSL. Its constraints and inference rules can be especially powerful as they can operate on the structural as well as the behavioral models, checking correctness of concerns that deal with both models. The Primitive Coordination DSL depends on the structural model through its references to the Motion Primitive DSL and depends on the behavioral model through extending the Coordination DSL. This enables constraints in the Primitive Coordination DSL that perform checks based on both models.

One example implemented with the Primitive Coordination DSL is the Converged-Condition that consists of a mandatory reference to a Criterion, so that it is only possible to check for convergence of Adaptive Components that indeed provide a Criterion and therefore provide an Adaptive Component Status in the first place.

Another example is a constraint defined for the PublishToSpace Action, to publish a certain data item to a Space. The Space and Space Type are part of the Motion Primitive DSL (belonging to the structural model), the data type is part of the Types DSL, and the Action itself part of the Primitive Coordination DSL (the behavioral model). The validation rule is part of the Primitive Coordination DSL, but can follow the reference to the Space in the structural models and check for compatibility between the Space Type and the Data Type of the item to send.

This is an example of a validation step that helps avoiding expensive debugging at run time and therefore repeating costly experiments. This concrete validation prevents an error that would not be detected by a compiler, but would lead to an error during run time that is usually hard to debug. If executed, the Joint Angle would potentially be serialized, sent over the network, and tried to be deserialized on the receiving side while the receiving side would assume a different data type according to the Space Type. This would in most cases result in a runtime error, e.g., in a segmentation fault when mapped to C++, as discussed in Chapter 7.

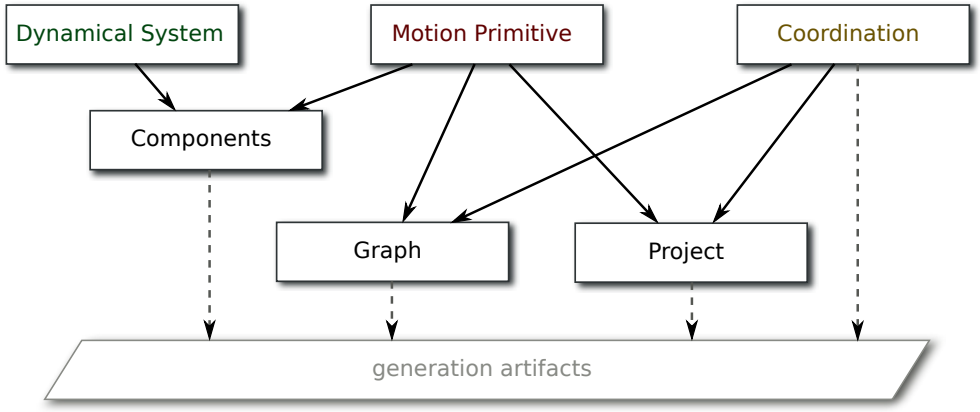


Figure 6.12.: Model-to-model transformations from higher abstraction languages to lower abstraction languages. Solid arrows indicate model-to-model transformations discussed in Section 6.4.2, dashed arrows indicate model-to-text transformations discussed in Section 6.4.3.

6.4. Transformations

A typical goal of model-driven engineering (MDE) approaches in general, and this work, is automation, e.g., through code generation, to bridge the gap between models and implementation, i.e. making the models executable. For external DSLs, as they are used in this work, there are mainly two different usage patterns: *generation* and *interpretation*. Both add execution semantics to the DSLs, but with different advantages of which just a few will be discussed here.

When interpreting models, the execution semantics is given by how the interpreter executes the models. This usually enables faster changes as it does not require extra code generation and deployment steps when the models change, since the models are directly executed. Code *generation* on the other hand defines the execution semantics of a DSL by mapping it either to other DSLs or eventually to the actual execution infrastructure, often the targeted GPLs. There are two main reasons why this work uses code generation for the most parts: firstly, the existing programming model, introduced in Chapter 7, simplifies code generation; and secondly because it fits quite well with the LMEC approach.

Code generation can be done from abstract levels down to pure GPL code, which has its advantages as it lowers platform requirements and dependencies. This is often not feasible though as code generators can become very complex. Existing programming models and domain-specific software libraries usually reduce complexity of code generators, as the difference between models and their corresponding software abstractions is smaller. Code generators then usually only configure the domain-specific software libraries according to the model instead of generating the full code.

In contrast to interpretation, code generation also fits well with the LMEC approach and the iterative process introduced in Chapter 4. While an interpreter is often mono-

lithic and has to be changed when the models change, code generation can be done gradually by first only generating parts of the code and implemented others manually. Later code generators can be extended gradually to generate more parts of the code. This also works by first providing generators for lower level DSLs and later tailoring it more and more to domain experts by raising the level of abstraction.

To add execution semantics to the DSLs, the transformations aspect (also called generator aspect) of MPS adds M2M and M2T transformations to the DSLs. While MPS would allow implementing code generators, i.e. M2T transformations from the languages discussed above to generate GPL code directly, it makes sense to transform the languages discussed above into so-called *intermediate models* that are one step closer to the final software artifacts, yet still models. This section introduces some of the implemented model-to-model transformations that generate these intermediate models, the final model-to-text transformations that eventually generate the actual executable GPL code, depicted in Fig. 6.12.

6.4.1. Intermediate DSLs

Generating intermediate models in a chain of several transformations, sometimes referred to as *chaining*, instead of directly generation source code in a single transformation has several advantages. An obvious reason is to break down the complex task of code generating into smaller, better maintainable pieces, i.e. multiple stages of M2M and M2T transformations, cf. Fig. 6.12. Fig. 6.12 shows the set of M2M transformations developed in this work targeting the generation of system-level visualization, executable source code, and project configuration for source builds. Solid arrows indicate model-to-model transformations, dashed arrows indicate model-to-text transformations that are introduced in Section 8.4.

Another reason is reuse of the intermediate models or languages. Several top-level languages may target the same artifact and can therefore target the same intermediate language with its M2T, instead of each language implementing the M2T in parallel. Both, the Motion Primitive DSL and the Coordination DSL, target generation of graph-like system visualization. Instead of both including M2T transformations for this target and therefore both targeting a concrete technology, i.e. concrete textual output, both implement M2T transformations to the intermediate Graph DSL, that implements the M2T transformations once and so that retargeting to a different target technology, i.e. different concrete textual output, can be done at one single place. Multi-stage transformations can get complex, though, complicating the debugging the overall transformation. MPS solves this problem by (optionally) keeping the intermediate models, i.e. models of the intermediate languages, for debugging purposes. [Völter et al., 2013]

Apart from the introduced DSLs that target the actual modeling of the motion primitive architectures, two further DSLs were developed in this work and serve mainly as intermediate steps in the code generation pipeline.

6.4.1.1. Project DSL

The Project DSL is a DSL with the purpose of modeling software projects, i.e. software artifacts and their library dependencies. It's main abstractions follow cross-platform build process software CMake [Martin and Hoffman, 2010] and are `Projects` and `Dependencies`. `Projects` contain an arbitrary number of `Dependencies`, which specify a software library name and a minimum required version, and if they are mandatory or optional. The language syntax and design is simple and not optimized for any of the design aspects above, since it is mainly used as an intermediate step in the transformation chain detailed in the following section for generating project dependency configurations, not to be written manually.

6.4.1.2. Graph DSL

The Graph DSL is a DSL with the purpose of modeling project visual representations of graphs. Its abstractions describe the topology and visual aspects of graph representations and are mainly inspired by the GraphML. Its main abstractions are therefore `Graphs`, `Nodes`, `Edges` and their `Labels`, as well as their visual attributes.

Similar to the Project DSL, the language syntax and design is simple, since it is mainly used as an intermediate step in the transformation chain for generating visualizations of the structural architecture and the coordination, as exemplified in the course of the thesis.

6.4.2. Model-to-Model Transformations

The M2M transformations shown in Fig. 6.12 map the high-level models of the motion primitive architectures, expressed in the Motion Primitive DSL, Coordination DSL, and Dynamical System DSL to platform specific models, such as the Component DSL and Middleware DSL models.

The models of the Motion Primitive DSL, the structural description of the motion primitive architecture, the `Circuit` with the `Adaptive Components`, `Adaptive Modules` and `Spaces` is mapped to a component `Circuit` and `Components` of the Component DSL. While this transformation yields the `Component` skeletons and their configuration, a mapping from the Dynamical System DSL yields the computational content of the `Components` that has to fit into the skeletons. Both, the models of the Motion Primitive DSL and of the Coordination DSL are transformed to Graph DSL models to yield two separate graph representations of the static architecture (the `Circuit`) and the dynamic aspects (the `State Machine`). Additionally, the Motion Primitive DSL and of the Coordination DSL models are transformed to Project DSL models, i.e. to `Dependencies` that represent their software dependencies.

Fig. 6.13 shows an exemplary transformation rule ("*reduction rule*" in MPS) from the Motion Primitive DSL that transforms (*reduces*) an `Adaptive Module` to a `Component`. These transformations rules, transforming Motion Primitive DSL models to Component DSL models, target generation of executable code.

```
Component $[name] ($[classname])
  Initial State: <no initstate>
  Processing Strategy: $COPY_SRC$ [timed(samplerate: 1)]
  Input Ports:
  $COPY_SRC$ [InputPort<DataType> <no name>(DEFAULT, <no scope>)]
  $COPY_SRC$ [InputPort<DataType> <no name>(DEFAULT, <no scope>)]
  $COPY_SRC$ [InputPort<DataType> <no name>(DEFAULT, <no scope>)]
  $COPY_SRC$ [InputPort<DataType> <no name>(DEFAULT, <no scope>)]
  $COPY_SRC$ [InputPort<DataType> <no name>(DEFAULT, <no scope>)]
  $COPY_SRC$ [InputPort<DataType> <no name>(DEFAULT, <no scope>)]
  Output Ports:
  $COPY_SRC$ [OutputPort<DataType> <no name>(DEFAULT, <no scope>)]
  $COPY_SRC$ [OutputPort<DataType> <no name>(DEFAULT, <no scope>)]
  properties: $COPY_SRCL$ [{ <no name>: false}]
```

Figure 6.13.: Model-to-model transformation rule to transform an Adaptive Module to a Component.

The transformation rule shown in Fig. 6.13 takes an Adaptive Module as input and yields a Component. The transformation rule shows the Component editor of the Component DSL and allows to filling its editable parts with macros based on the properties of the input Adaptive Module. Strings in braces, e.g., “name”, are properties of the Component that are set based on properties of the input, i.e. the Adaptive Module.

The name of the component is based on the Adaptive Module name, but eliminated white spaces as these are forbidden in the name property of the Component concept. “$COPY_SRC$[...]” indicates that another transformation rule is called. The content of the bracket is then replaced by the output of this transformation rule, e.g. the Input Ports of the Component in this example are generated by calling transformation rules on all Inputs of the Adaptive Module that transform these into Input Ports. Configuration of the resulting Ports is done according to the Space that the respective Inputs and Outputs were connected to.

Note, that the Adaptive Components are not directly mapped to single Components, but rather to a set of Components, namely its contained Adaptive Module, the Criterion, and the Mappings, each configured according to internal wiring of the specific Adaptive Component, cf. Section 5.1.4.

6.4.3. Model-to-Text Transformations

In addition to the M2M transformations, M2T transformations are specified to generate the textual artifacts, e.g., executable GPL source code and textual specifications for a graph-like system visualization. The transformations developed in this work to generate the actual executable experiments are discussed in more detail from a user perspective in Section 8.4. This section rather exemplifies M2T transformations in MPS in one example.

While code generators can get quite complex, the code generators in this work are well maintainable due to the chosen language modularization and M2T transforma-

```
<$[state] id="$[state id]">
  $COPY_SRCL$ [invokes ]
  <onentry>
    $COPY_SRCL$ [<actions></actions> ]
    $LOOP$ [$IF$ [<send id="$[wait condition]" eventexpr="$[wait condition]" delayexpr="$[duration]"></send> ]]
  </onentry>
  <onexit>
    $LOOP$ [$IF$ [<cancel sendid="$[wait condition]"></cancel> ]]
    $COPY_SRCL$ [<actions></actions> ]
  </onexit>
  $COPY_SRCL$ [<transitions></transitions> ]
</state>
```

Figure 6.14.: Generator rule to map a Coordination DSL `State` to SCXML elements.

tions that mapped the most domain-specific and abstract concepts already to models closer to implementation, as discussed in Section 6.4.2.

Fig. 6.14 shows an exemplary generator rule of the Coordination DSL generator. The code generator targets State Chart XML (SCXML) [Barnett et al., 2013], a free eXtensible Markup Language (XML)-based file format for graphs. The Graph DSL generator generates XML files in the GraphML format. This is technically a two-step transformation, as an M2M transformation targets the MPS' XML base language, which itself has generation rules to produce the actual text file in XML format as output. This is not explicitly visualized in Fig. 6.12 for the sake of clarity.

The generator rule shown in Fig. 6.14 transforms a Coordination DSL `State` into an according SCXML (XML) element. Similar to the M2M transformation rule shown in Section 6.4.2, the rule consists of several phrases enclosed with a bracket ("$[...]") which indicates wildcards that are replaced with actual model properties during code generation, e.g., the "id" property of the XML node element is replaced based on the "name" property of the Coordination DSL `State` that is transformed by this rule.

The result of this code generation step is an SCXML file, which performs the intended coordination of the motion primitives when executed with the *Apache Commons SCXML* engine, cf. Section 7.2.2.

When code generation targets a format that is, other than XML, not available as base language in MPS, code generators are written in the so-called *TextGen* language aspect to generate text output directly. The *TextGen* aspect contains constructs to print out text, transform nodes into text values as well as layouting the text output. This is exemplified for further code generators in Section 8.4.

6.5. Discussion

This chapter proved the language modularization, extension, and composition approach suitable to implement a set of DSLs to cover the richness of the domain introduced in Chapter 3 and to cover the separate concerns presented in the metamodel in Chapter 5. While the functional requirements introduced in Section 4.2 were mainly tackled by the models presented in Chapter 5, the DSLs introduced in this chapter take several of the non-functional requirements into account. While the use of the language workbench is not in line with the most common choice in robotics, Xtext [Nordmann

et al., 2014], this work makes reuse of MPS, an state of the art open-source language workbench (**NFR1**). The language aspects available with MPS to design DSLs were suitable to design the structure, the concrete syntax, constraints and type system, as well as multi-stage transformations for code generation. The language modularization, foremost the according modularization of the transformations, allows and eases iteration and constant refinement to keep up with new research results, as the transformation and code generation is divided into small, maintainable transformation steps (**NFR4**).

The focus of this chapter and the presented DSLs, though, is to ease expressing of domain problems and solutions for domain experts (**NFR5**) and allow formulation of systems in a technology-independent and platform-neutral way (**NFR6**). The presented DSLs with the concrete syntax, constraints, type systems, and transformations provide the basis for rich editor support like syntax highlighting and context help, validation at design time and code generation. Especially the Primitive Coordination DSL makes the potential of the LMEC approach visible: It operates on the structural models as well as the behavioral models and can therefore provide support on the cross-cutting concerns, which are often hard to incorporate and consider in a classical development process, especially when system grow and become complex. This chapter shows some examples; more will be exemplified and discussed from the user's perspective in Chapter 8 - Chapter 10.

In the same way that the Dynamical System DSL extends the Motion Primitive DSL to allow modeling of the algorithmic models, the `Dynamical Systems`, further languages can be composed with the presented approach to detail further concepts and aspects. The DSL introduced by Klotzbücher et al. [2011] could be incorporated to express `Dynamical Systems`, and the geometric relations DSL by Laet et al. [2012c] could be incorporated to ease specification of `Mappings` and `Transitions`. The language modularization, extension, and composition and the introduced multi-staged transformations, however, allow extending these languages. At the current state of affairs, this requires re-implementing these DSLs in MPS for the sake of meta-metamodel compatibility. This is exemplified in this work with the reuse of SCXML and GraphML.

The Component DSL was designed for an easy mapping of the Motion Primitive DSL concepts to component-based software artifacts inside MPS. Existing ADLs such as Architecture Analysis and Design Language (AADL), the Unified Modeling Language (UML), and the Systems Modeling Language (SysML) can be considered as potential (alternative or additional) transformation targets as they provide rich modeling support for technical architectures, are widely used, and come with huge tool support.

Chapter 7.

Programming Model and Technology Mapping

Target of this work and the design process introduced in Chapter 4 is the generation of executable systems that can be run on robots to test motion primitive architecture hypotheses (**NFR7**). The idea behind model-driven engineering (MDE) is to automate the necessary software development by automatically mapping these models to an executable target technology as discussed in Section 6.4. The target technology of this mapping is termed *technology mapping* in conformance with [Völter, 2005] and denotes the software architecture, software libraries and programming languages, the models are mapped to, e.g., by means of code generation.

Before automation can be set up, the target technology needs to be tested to evaluate if the technology mapping addresses the requirements of the domain and the project context (**NFR2**), e.g., platform requirements, performance, or quality of service requirements. In order to so, developers need to implement exemplary applications against this technology mapping manually at first and, in order to do this consistently, need a programming model. A programming model in this sense is the "architecture API" [Völter, 2005] that applications are implemented against, i.e. it specifies how an architecture is used from a developer's perspective. Völter [2005] emphasizes the importance of mock platforms and vertical prototypes during a design process as proposed in Chapter 4. A mock platform, e.g., a robot simulator, allows developers to fast and easily evaluate their systems locally, sparing the typically great effort of experimenting with hardware. Vertical prototypes on the other hand are proof-of-concepts that implement complex features of a system through all implementation levels. They serve as an early evaluation of the chosen programming model and technology mapping before automation of software development is set up to map the models to this technology.

Section 7.1 introduces the programming model that was developed in this work in compliance with the metamodel presented in Chapter 5. Section 7.2 introduces an exemplary technology mapping that was implemented in the course of the AMARSi project to realize the programming model. Section 7.3 presents two mock platforms and vertical prototypes that were implemented in the AMARSi project and further research projects to evaluate the programming model and technology mapping. Section 7.4 shows how the programming model, once established, serves as a basis to include existing software artifacts into the MDE process (**NFR3**). Software imple-

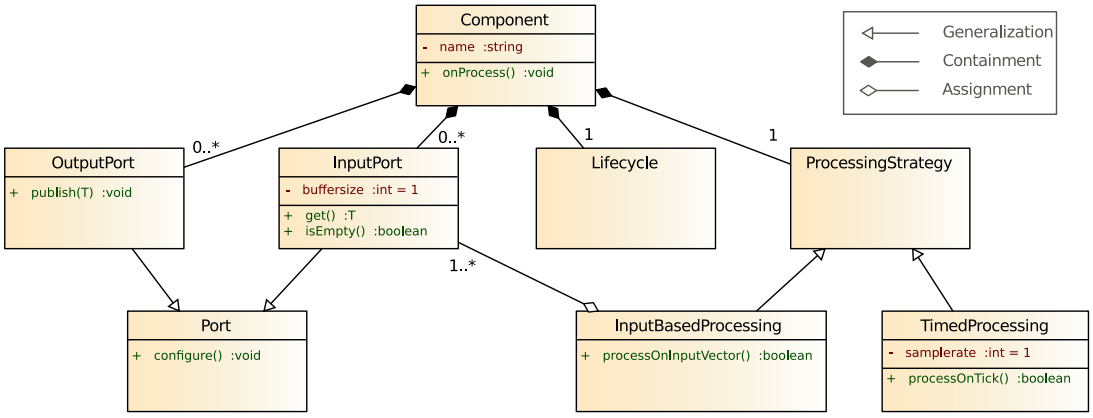


Figure 7.1.: Class diagram of the programming model, focused in the component level.

mented against this programming model can be incorporated into the software development automation, since it thereby complies with the metamodel, as it is detailed in Section 7.4.

7.1. Programming Model

The programming model is a software realization of the metamodel presented in Chapter 5 as it maps the conceptual ideas of the metamodel to technical concepts. It further details the relation between the domain-specific and the more technical parts of the software architecture. The programming model introduced in this work is targeted by the code generators and used when implementing software explicitly for the motion primitive environment. The following sections describe the concepts and abstractions of the programming model in terms of its exposed interfaces, properties, methods, and events. It is a result of iteration within the AMARSi research project together with the project partners and has been extended along the language concepts of the Motion Primitive DSL. The programming model needs to support execution of the domain-specific language (DSL) concepts introduced in Chapter 5, such as `Adaptive Component`, `Adaptive Module`, `Mappings`, `Dynamical Systems`, as well as their interplay.

7.1.1. Structural Model

State of the art in software development in robotics and several other disciplines is component-based software engineering (CBSE), which aims to build complex systems by composing software systems from a combination of reusable off-the-shelf or custom-built components.

A system in this sense is realized as a graph of loosely coupled components communicating via dataflow. One of the key advantages is that more than one instruction can be executed at once. Thus, if several components are ready to process at the same

time, they can be executed in parallel, therefore potentially allowing massive parallel execution. The graph in itself does not precisely specify the order of execution, but instead is data-driven: Whenever input data is available, the respective components will be processed and the result is sent.

Classical dataflow models assume infinite parallelism, which is not realizable. Therefore, other models such as Kahn Process Networks have been suggested that are demand-driven, i.e. components are only activated when their outputs are requested. Other models, such as Synchronous Data Flow use metadata to statically schedule execution as needed. However, these approaches also come with complexity and are primarily necessary for more fine-grained dataflow than targeted within work, e.g., when instruction rather then component granularity is used.[Lütkebohle and Wachsmuth, 2011]

Therefore, the programming model developed for this work uses the classical data-driven execution with one or more threads per component. This approach reduces the complexity and therefore time to develop and test new applications [Brugali and Scandurra, 2009, Brugali and Shakhimardanov, 2010, Bischoff et al., 2010]. The main processing element is the base component that the models of the motion primitive architecture are mapped to, shown in Fig. 7.1. It contains the algorithms of a sub-part of the application, e.g., a motion primitive. In order to communicate, components have an arbitrary number of input and output ports that are connected to a communication channel. Each port is strongly typed and has an identifier of the communication channel it participates in, termed *scope*. Input ports receive data, can queue them and hand them to the computational code inside the component. Components can publish their computation results through their output ports. Ports can be defined to communicate locally (only with components in the same process) or remotely, i.e. over the network with components in other processes and on other machines. Each input port has a buffer of an arbitrary (yet configurable) size to decouple computation and different data consumption rates of connected components. By default, input ports have a buffers size of 1 and always keep the last item.

7.1.1.1. Processing Strategies

When and *if* a component is processed is determined by the `Parallel States` defined in the structural models in Section 5.1. The programming model defines interfaces for timing-based strategies, which process components based on a fixed global clock, and input-based strategies, which processes components based on incoming data. When configured with an input-based strategy, e.g., port-triggered, incoming data at the configured port or a combination of ports immediately triggers computation of the component. By coupling several components in this way, complete effect chains, e.g., calculating a new control command based on latest sensor readings, can be created, all triggered when new data is published to the first component in the chain. Input-based strategies are the default strategies for `Mappings`, `Transformations`, and `Criterions` to compute the output immediately when new data arrives. Fig. 7.2 shows this for the `Forward Kinematics` and the `Adaptive Module` of the running example.

Example: Mapping of the Reaching Controller to the programming model.
An example of the Motion Primitive DSL concepts mapped to software artifacts of the chosen programming model is shown in Fig. 7.2. Both, the Forward Kinematics Mapping, as well as the Adaptive Module are mapped to Components. The default Processing Strategy for Mappings is Port Triggered, which means it will always compute when new data arrives at its Input Port and send the result of the mapping as soon as the calculation is done. The Adaptive Module is mapped to a component with two Input Ports and one Output Port, and is configured with a Timed Processing Strategy by default. The model-to-model transformation (M2M) maps the Output Port of the Mapping and the goal Input Port to the same Scope to establish communication.
With this mapping, the Adaptive Module will produce a stable control signal on its Output Port with the fixed timing given by the Timed Processing Strategy, always with the latest received goal. The goal is updated every time the Mapping receives data, immediately computing the output due to the Port Triggered Processing Strategy and passing it on to the goal Input Ports of the Adaptive Module.

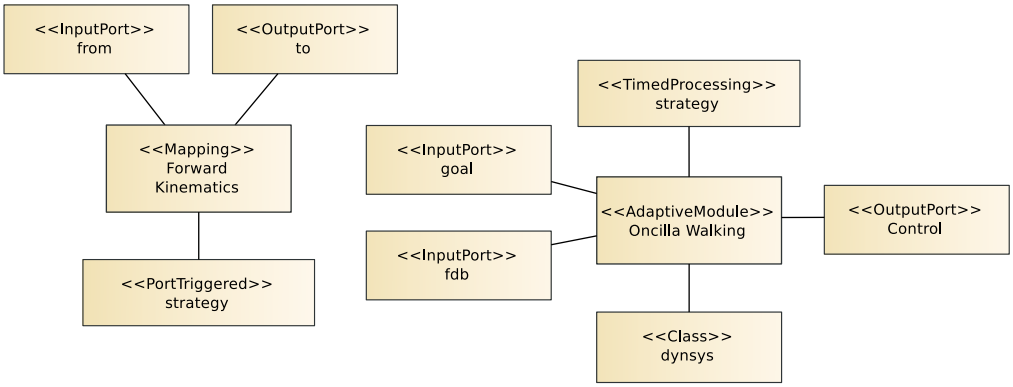


Figure 7.2.: Object diagram, showing the technology mapping of the Reaching Controller of the running example.

Timing-based strategies subscribe to a global timing signal of the component circuit, called the *heartbeat*. It can either be provided by a clock component, which produces a timing signal with a fixed rate, or arbitrary other components, e.g., a component representing the robot platform. In this way, the timing either has a fixed rate, is based on the cycle time of a robot, or even based on the virtual time provided by a simulation component. Timing-based strategies are the default for Adaptive Modules to produce the control output with a stable fixed rate, which is often required for robotics control. Fig. 7.2 shows how the the Reaching Controller from the running example maps to the programming model. When configured with a timed Process-

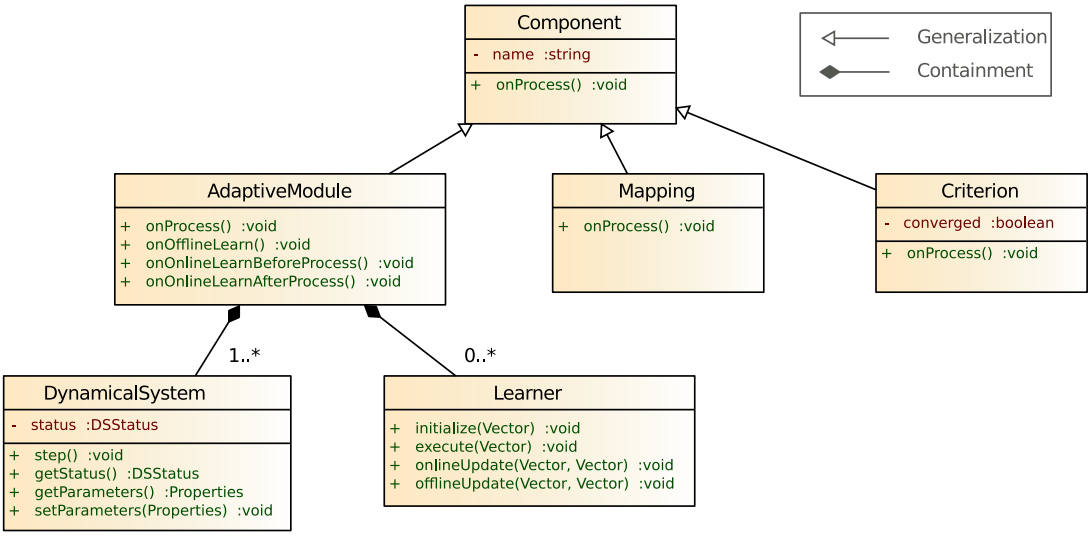


Figure 7.3.: Class diagram of the domain-specific programming model, realizing the domain-specific models introduced in Chapter 5.

ing Strategy, Adaptive Modules may miss the latest sensor reading when they start processing right before the sensor reading arrives. However, Dynamical Systems are robust against these kind of perturbations, so that a fixed timing on the Output Ports is often favorable over guaranteeing not to miss a sensor reading. This can even be a requirement given by the robot's interface as discussed for the KUKA Lightweight Robot IV (KUKA LBR IV) in Section 7.3.2.

7.1.1.2. Component Lifecycle

The base component has a lifecycle that allows transitions between different component states such as execution, online learning, offline learning, and stopped. Each state provides programming hooks that execute user-defined code on entering the state, on exit, and when the component is periodically triggered by the processing strategy. Additional generic lifecycle states defined by the base component are states for loading, persisting, and resetting. Resetting includes cleaning all caches and input queues of the component and calling hooks that allows the implementation of component-specific reset functionality, e.g., resetting of the internal state.

7.1.1.3. Domain-Specific Components

Apart from the basic component, the programming model defines domain-specific components according to the structural model introduced in Section 5.1. An excerpt of how this programming model applies to the concepts defined in the Motion Primitive DSL is shown in Fig. 7.3. The Adaptive Module, as one of the core concepts of the Motion Primitive DSL, is realized in this programming model as a specialized component, with a set of input and output ports with fixed roles. These ports correspond

to the Inputs and Outputs defined in the Adaptive Modules concept. The Adaptive Module has an extended domain-specific lifecycle, providing hooks for the specific learning states of an adaptive module. In extension to the rather generic states and transitions, the programming model defines domain-specific states and transitions that are necessary in order to perform learning in the Adaptive Modules. For this purpose, the actual processing is divided into three different states: *Execution*, *OfflineLearning*, and *OnlineLearning*, cf. Fig. 5.7.

- In the **Execution** phase, the component is computing on every processing step, i.e. the component-specific implementation of the processing method.
- In the **OfflineLearning** phase, the component performs an offline learning step, e.g., a batch update, on every processing step, i.e. the component-specific implementation of the offline learning hook.
- In the **OnlineLearning** phase, both, execution and online learning, is performed on every processing step. In order to support different learning methods, three hooks are called on every processing step that can have component-specific implementations: an online learning step *before* execution, the execution step, and an online learning step *after* execution.c

This allows a component developer to define a certain behavior before, during and after the component is processed in its online or offline learning state. In online learning mode, hooks are arranged to allow learning steps before and after the processing step, for instance to allow comparison of the Dynamical System before and after processes, which is convenient or even necessary for some of the learning approaches surveyed in the domain analysis.

The Dynamical System in the proposed the programming model has a step method to advance it by one step, a Dynamical System Status, and parameters for tuning.

Note, that an Adaptive Component, part of the structural model introduced in Section 5.1, does not have a specific counterpart in the programming model, as it expresses a *structural* aspect and is therefore mapped to a set of components with their specific port configuration to realize the internal logic of the Adaptive Component.

7.1.2. Behavioral Model

The programming model for the core behavioral models detailed in Section 5.2 is chosen in conformance with Harel state charts [Harel and Politi, 1998].

It is an event-driven model that consists of actions, which are invoked in response to events. Actions are organized in states that can be transitioned to based on the occurrence of events, usually managed through a so-called dispatcher with an event queue.

To specify a program, the developer implements custom actions and assigns those actions to states. The program control remains with the dispatcher and is passed to the actions upon occurrence of an event.

An advantage of this programming model is that a large number of technology mappings already exist, e.g., in the form of open-source execution environments for State Chart XML (SCXML) with support for a large-number of programming languages, including C/C++, C#, Java and Python, or as support for Unified Modeling Language (UML) State Machine Diagrams that can be mapped to Harel state charts.

7.1.3. Algorithmic Model

The algorithmic model detailed in Section 5.3, i.e. the `Dynamical System` specified with a mathematical expression, does not require a dedicated programming model. Its model elements such as variables, constants, and algorithmic operations are core parts of almost all general-purpose languages (GPLs) and are therefore already available for developers.

7.2. Technology Mapping

A concrete technology mapping, i.e. an exemplary implementation of the programming model for a specific software platform and in a specific programming language, serves as an early validation of the concepts before being targeted by code generators. It can checked for the quality of service which matches the requirements of the application domain and addresses the non-functional requirements, cf. Section 4.2.

The metamodel and the DSLs introduced in Chapter 5 and Chapter 6 respectively are designed to be technology-independent (**NFR6**) so that a technology mapping can be chosen independently and even multiple different technology mappings can be designed that the models will be mapped to. This is useful in robotics where different robot platforms and applications may require completely different technologies, e.g., real-time capable software on embedded systems, or large distributed systems for computationally heavy machine learning applications.

This section introduces an exemplary technology mapping that was initially created and iterated in the four years European research project AMARSi and used for the use-cases and evaluation presented in Chapter 10. It is implemented mainly in C++, as this is a common choice for the motion control stack of robots, but was replaced by Java for the behavioral part for usability reasons as part of the iteration process proposed in Chapter 4.

7.2.1. Technology Mapping for Structural Aspects

The structural and algorithmic aspects are implemented based on a general-purpose middleware and a robotics data type library, a rather generic component framework and a domain-specific library that provides interfaces for the domain-specific abstractions of the metamodel presented in Chapter 5.

Many of the non-functional requirements outlined before can typically be delegated to a (robotics) middleware. For instance, the ability to distribute or co-locate components in the same process should be supported by a middleware framework and not be

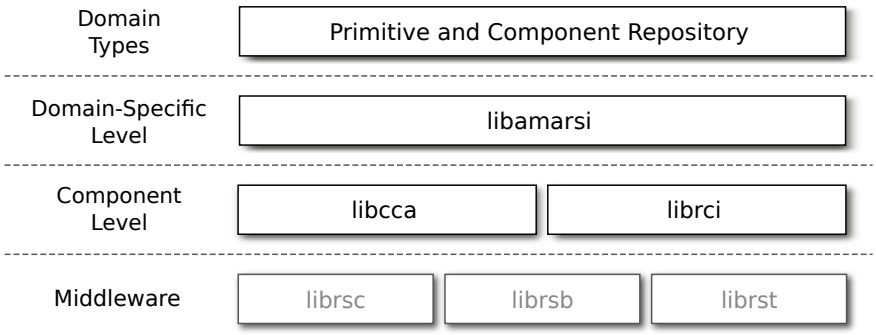


Figure 7.4.: Technology mapping for the structural aspects with *libamarsi* realizing the higher-level domain-specific aspects, *libcca* and *librci* realizing the component level, and *librsc, librsb, librst* realizing the middleware level. Libraries in light gray boxes were not implemented as part of this work.

part of the domain-specific architecture. The proposed technology mapping adopts the event-based middleware Robotics Service Bus (RSB) [Wienke and Wrede, 2011] due to its small footprint, extensive configurability, and openness. Additionally, RSB provides the required tools for experimental research and integration with other relevant frameworks such as the Robot Operating System (ROS) [Quigley et al., 2009] or Yet Another Robot Platform (YARP) [Fitzpatrick et al., 2008]; cf. [Wienke et al., 2012, Moringen et al., 2013]. It therefore allows the technical integration of high-level components implementing cognitive models such as cognitive levels or external perception (**FR5**).

The Compliant Component Architecture (CCA) and the Robot Control Interface (RCI) provide component-based software abstractions for experimental robot platforms [Nordmann et al., 2012b]. RCI is available as a C++ library called `librci`, providing interfaces and base implementations for robot control and sensing (mainly proprioception). CCA, available as C++ library `libcca`, realizes the component-based programming model introduced in Section 7.1.1 and allows implementation of functional components.

In addition to these rather generic libraries, the C++ library `libamarsi` realizes the domain-specific abstractions and interfaces, i.e. base classes and interfaces for `Adaptive Modules`, `Dynamical Systems`, etc., according to the programming model introduced in Section 7.1.1. It therefore is the application-programming interface (API) level that developers use to implement software for the proposed software architecture, e.g., their `Adaptive Modules` or `Dynamical Systems`.

7.2.2. Technology Mapping for Behavioral Aspects

The technology mapping for the behavioral parts of the programming model is based on the RSB middleware as well as the open-source SCXML execution engine *Apache Commons SCXML*. The proposed technology mapping uses an SCXML engine that

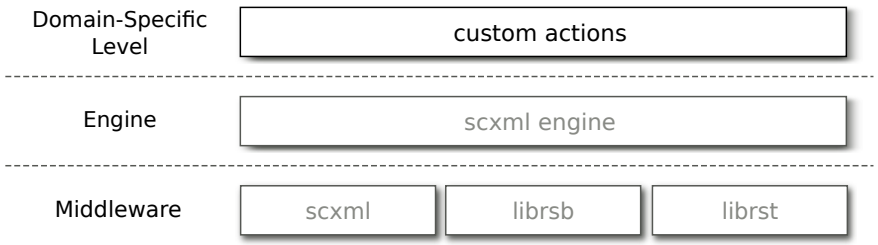


Figure 7.5.: Technology mapping for the behavioral aspects with the custom actions for motion primitive architectures realized based on an SCXML engine that extends the Apache SCXML engine with connections to the *librsb* middleware and the *librst* type library. Libraries in light gray boxes were not implemented as part of this work.

extends the base *Apache Commons SCXML* engine by connections to the RSB middleware and the *librst* type library. The rather generic engine that can be used for coordination in any RSB based system is extended by custom actions and filters implemented in Java to add the domain-specific `Adaptive Components` and `Conditions` proposed in Section 5.2.

This technology mapping of the behavioral aspects is an example of the iteration foreseen by the design process introduced in Chapter 4. In the first iterations of the technology mapping, the SCXML code specifying the system-level coordination was executing using the open-source Qt library *scc*, an SCXML compiler for the Qt state machine framework. This was a natural choice as *scc* generates C++ code from SCXML code, so mapping to the same programming language as the technology mapping for the structural aspects. This choice, however, turned out to be disadvantageous, as it requires recompilation every time the SCXML code for coordination changes.

The recent iteration using the *Apache Commons SCXML* execution engine directly executes SCXML code without a compilation step. The *Apache Commons SCXML* abstractions for custom actions, cf. Section 7.1.2, can be implemented in Java and can be easily reused across different SCXML states.

7.3. Mock Platforms and Vertical Prototypes

Mainly two mock platforms and vertical prototypes were used during the design process proposed in Chapter 4 to develop and evaluate the programming model and technology mapping described in this chapter. They were iteratively developed and refined in the course of the AMARSi project based on a quadruped robot simulator mainly serving as mock platform, as well as an industrial robot simulator and hardware that together with machine learning components served as full vertical prototype for motion primitive architectures.

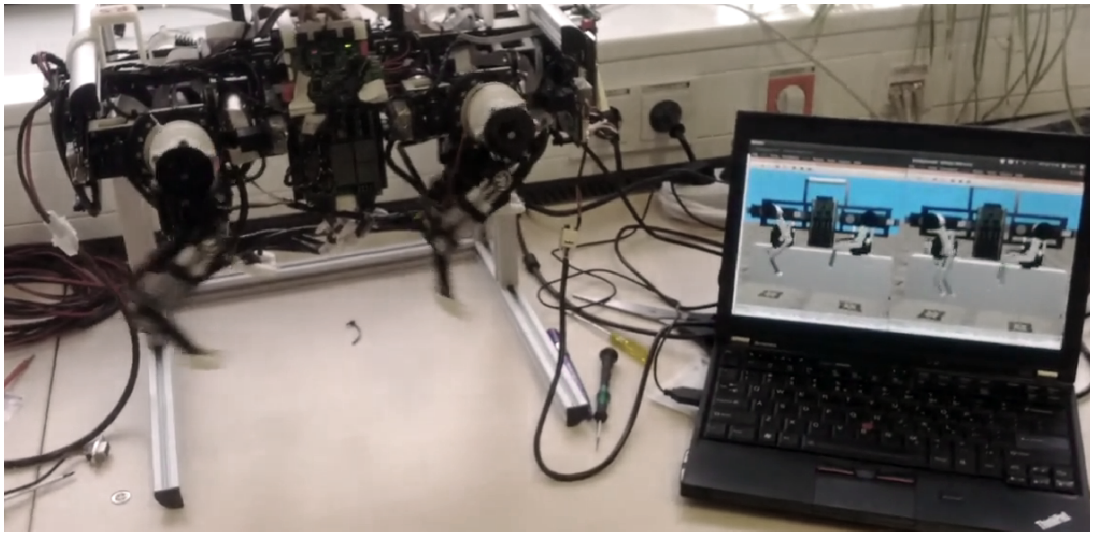


Figure 7.6.: Oncilla roundtrip.

Having a *mock platform* (in terms of [Völter, 2005]) is vital for early iteration of the technology mapping and evaluation of the chosen abstractions in the metamodel and programming model. In robotics, this can spare a lot of time-consuming and costly experimentation with a real robot, and additionally reduce unnecessary wear and tear. Testing with a mock platform, e.g., a simulator, however, does not replace experimenting on the real robot, which should always follow successful simulated experiments to incorporate all factors of the real world.

Both platforms are configured so that they produce and publish their sensor values with a fixed timing and are continuously listening for new control commands.

7.3.1. Oncilla Mock Platform

The mock platform used during development of the abstractions, programming model, and technology mapping introduced with this thesis is the simulator of the Oncilla quadruped robot (Oncilla), initially shown in Chapter 5, Fig. 5.1. It is particularly well suited as platform exploration of the motion primitive domain, as it is a relatively low-cost robot, yet, equipped with biologically leg mechanism with passive compliance [Spröwitz et al., 2011]. Fig. 7.6 shows the robot hardware (mounted on a stance) on the left and the simulated robot running on a laptop at the bottom right. The Oncilla features three actuated degree of freedom per leg; hip angle in the for-aft direction, leg length, and hip rotations in the transverse plane. It was designed for easy access to blueprints and documentation to provide a convenient research platform for quadruped robotics.

A robot simulator has been created with a precise description of the robot kinematics and dynamics, including the particular parallel compliance and asymmetric actuation of the robot's leg mechanism, cf. [Nordmann et al., 2013a]. Fig. 7.6 shows the model

inside the Webots simulation front end [Michel, 2004]. On top of the Webots model, a CCA/RCI-based interface with a taxonomic representation of the robot joints has been designed. One of its main design goal was a common abstraction between hardware and simulation, with binary compatibility. This interface is locally accessible through a C++ interface, using multiple inheritance to expose the node taxonomy. It is also remotely available through the CCA component framework in C++, and through the RSB middleware with C++, Java, Python, and Common Lisp bindings. The simulator and the real robot share the same (local and remote) interface to allow easy and fast transition from simulation to real-world experiments, which was an explicit design decision [Nordmann et al., 2013a]. Both publish the sensor values with a fixed timing so that software components of a connected component circuit can run synchronous with the robot when being configured with an `Input-Based Processing Strategy`. Coupling several components in this way can create entire effect chains to calculate new control commands based on the latest sensor readings. All components of the effect chain are triggered when new sensor readings are published from the robot, and therefore executed synchronous with the robot.

7.3.2. KUKA LWR Vertical Prototype

The full vertical prototype in terms of a complete elaboration of a system that was used during development of this work is an industrial robot system with machine learning components for physical human-robot interaction (pHRI), termed *FlexIRob*, cf. [Nordmann et al., 2012a, Emmerich et al., 2013, Wrede et al., 2013, Nordmann et al., 2015]. The system uses a KUKA LBR IV, which is a redundant manipulator with seven joints, allowing a manifold of configurations in joint space for a single end effector position. The KUKA LBR IV is well suited as a vertical prototype for the motion primitive domain as it comes with rich sensor feedback, and an impedance-based control scheme, resulting in active compliance of the manipulator [Bischoff et al., 2010]. A hierarchical controller [Nordmann et al., 2012a] controls the end effector position as well as the joint configuration according to a given redundancy resolution. The redundancy resolution is trained in pHRI and encoded in an artificial neural network (ANN). This can also be used for an assisted gravity compensation mode [Emmerich et al., 2013] where the end effector is freely movable and the trained ANN continuously generates an appropriate redundancy resolution. To enable these control and interaction modes, the system facilitates several motion primitives trained in online learning and offline learning mode, as well as analytical motion primitives for control [Nordmann et al., 2015].

The system is also available as a mock platform for fast and easy prototyping and experimentation, facilitating an OpenRave [Diankov and Kuffner, 2008] based simulator. The simulator has the same (local and remote) interface as the real robot platform to allow easy and fast transition from simulation to real-world experiments as well as replaying real experiments in simulation. Both publish their sensor values with a fixed timing so that software components of a circuit moving the KUKA LBR IV can run synchronous with the robot when being configured with a `Input-Based Processing`

`Strategy`. The KUKA LBR IV interface additionally produces a global clock signal for the component circuit so that even components with timing-based `Processing Strategies` are also synchronous with the robot. The KUKA LBR IV simulator produces the timing signal based on its (virtual) simulation time so that the entire circuit can also be executed in simulation time.

An industrial application further detailing this vertical prototype is featured in Chapter 10.

7.4. Deployment Descriptors

Leveraging existing software components and including it in the proposed MDE approach is a requirement of this work (**NFR3**). It does not only help to incorporate a large corpus of valuable legacy work and components that are hard to reverse engineer in the DSLs, but it also provides flexibility to researchers and developers. When concepts under research are not yet stabilized enough to be incorporated into the metamodel and DSLs, the programming model can serve as a basis for compatibility with the metamodel, but at the same time leaving the full flexibility the general-purpose language has to offer to the developer to implement the inner workings of the subject of research. At the same time, this can eases and therefore speeds up adoption of the model-driven and DSL based development process as it provides the means for gradual transition from classical development processes. This is an important requirement in research projects, where project-executing organizations expect frequent deliverables, demonstrators, and runnable experiments to verify the project progress.

As a means to do so, descriptions of software components, termed *deployment descriptors*, are proposed and developed in this work. They describe the metadata, the interface, and the deployment of software artifacts, exemplified in Listing 7.1. The metadata aspects of the deployment descriptor, comprising of name, description, and author(s), are mainly useful to build a repository of software components compatible with the programming model and to provide an overview to the user.

The interface aspects of the deployment descriptor hold the necessary information to represent the described software artifact as concept in the domain-specific language, as exemplified in the following example box. They describe the concept as *"black-box software building blocks with explicated properties"* [Schlegel et al., 2015], describing it to the level necessary to integrate it into the language context and include it into basic validation, e.g., type checking as discussed in Chapter 6. The third part of the deployment descriptor describes the deployment information of the software artifact to the level that is necessary to incorporate it into the code generation detailed in Section 8.4. It describes the necessary headers that need to be included and the path to the CMake configuration file, which holds more information on software dependencies, necessary compiler flags, etc. The deployment descriptors are currently tied to the technology mapping introduced in Section 7.2 and rely on C++ and CMake [Martin and Hoffman, 2010] for configuration.

Listing 7.1: Deployment Descriptor for the MVITE controller

```
<?xml version="1.0" encoding="UTF-8" ?>
<DynamicalSystem>
  <name>MVITE</name>
  <author>EPFL</author>
  <description>
    Modified Vector-Integration-To-Endpoint (VITE)...
  </description>
  <classname>MVITEDS</classname>
  <headers>
    <header>MVITEDS.h</header>
  </headers>
  <cmakeconfigfile>
    /homes/anordman/prefix/share/MVITEDS/MVITEDSConfig.cmake
  </cmakeconfigfile>
  <cmakeprefix>MVITEDS</cmakeprefix>
</DynamicalSystem>
```

Software artifacts implemented against the introduced programming model, and thereby compatible with the metamodel introduced in Chapter 5, can thereby be described to reflect their counterpart in the metamodel. This paves the way to integrate them into the DSL toolchain. Chapter 8 discusses this from a user's perspective in more detail. The following example box shows this with the running example, in which the `Dynamical System`, formerly specified with a simple mathematical expression, cf. Section 6.3.2, is replaced by the MVITE controller loaded via its deployment descriptor.

7.5. Discussion

The introduced programming model describes the API of the software architecture, i.e. how it is used from a developer's perspective. It provides software abstractions for the concepts of the metamodel introduced in Chapter 5. Based on a the Oncilla based mock platform and the KUKA LBR IV based vertical prototype introduced in Section 7.3 an exemplary technology mapping is presented. The proposed technology mapping realizes the structural and behavioral aspects of the domain based on a number of open-source C++ libraries and the widely used SCXML execution engine *Apache Commons SCXML*. The technology mapping underwent an iteration to raise flexibility and ease prototyping and testing of the `State Machines`.

The programming model conforms to the metamodel described in Chapter 5 and is therefore the basis for developing compatible software. An exemplary technology mapping is provided for implementing motion primitive systems and validating them on mock platforms and vertical prototypes. The mock platform used during the development of this work is the Oncilla with convenient local and remote interfaces. A

Example: Integrating a software artifact into the DSL toolchain
As a basic example, how the programming model can facilitate integration of an existing software artifact, the `Dynamical System` in the running example is switched with a `Dynamical System` implemented against the programming model and described with the deployment descriptor shown in Listing 7.1. The `Dynamical System` is a modification of the popular VITE controller that produces a bell-shaped velocity profile [Bullock and Grossberg, 1989].
An implementation of the MVITE controller was implemented against the introduced programming model by an AMARSi project partner and described by a deployment descriptor. Fig. 7.7 shows a DSL snippet from the running example, where the `Dynamical System` is no longer specified by its mathematical expression, cf. Section 6.3.2, but instead the MVITE controller loaded via its deployment descriptor.

```
primitive joint ctrl
  strategy: timed(samplerate: 1)
  in: goal<leg angles>, fdb<leg angles>, <no cfg>, <no speed>, <no phase>, <no statein>
  out: ctrl<leg angles>, <no stateout>
  ds: MVITE vite from descriptor (::amarsi::MVITEDS)
  Modified Vector-Integration-To-Endpoint (VITE) as AMARSi dynamical system.

  properties: << ... >>
```

Figure 7.7.: The MVITE `Dynamical System` that was implemented against the proposed programming model and described with the deployment descriptor shown in Listing 7.1, replaces the formerly used `Dynamical System` given by a mathematical expression.

KUKA LBR IV based system with motion primitives and machine learning components for physical human-robot interaction is used as vertical prototype and will be further detailed and discussed in Section 10.1.

The deployment descriptors allow describing existing software artifacts or new software artifacts for research and prototyping, to be included into the domain-specific language models. The deployment descriptors describe software artifacts on a level that allows basic validation on a model level as well as inclusion into the targeted code generation. This addresses an important non-functional requirement of this work, i.e. integration of existing software artifacts into the model-driven engineering processes (**NFR3**).

The final view on the levels of modeling in the proposed development process is depicted in Fig. 7.8. The right side of the illustration shows the modeling, i.e. specification of applications in the DSLs, as discussed before. The left side shows the software artifacts where (existing or currently prototyped) software components that conform to the programming model can be integrated into the DSL models and therefore incorporated in the automated development process through code generation, as further detailed in the following chapter in Section 8.4. This helps answering the research

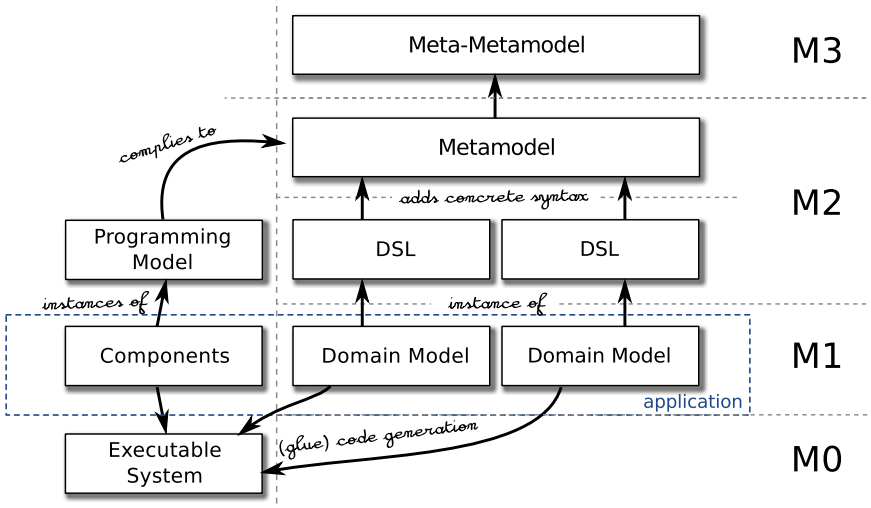


Figure 7.8.: Levels of modeling in relation to the deployment descriptors.

questions **RQ3** and **RQ4** introduced in Chapter 1 if the DSL based approach can be organized in a fashion that is open and flexible enough to allow continued research in the domain and at the same time incorporating legacy work to lower the risk of introducing model-driven engineering methods into robotics research.

The loose coupling of components and asynchronous communication, chosen for the introduced technology mapping, permits fast prototyping of component-based systems and is common practice in several robotics frameworks such as the ROS. While this proved to be useful for practical validation of the chosen domain concepts on compliant robots, it may, however, result in indeterministic behavior in the presence of delays. For use-cases where this is harmful, e.g., rigid robots and strict requirements for precision and timing, a different technology mapping that allows static scheduling should be preferred [Lotz et al., 2015]. Kahn Process Networks and Synchronous Data Flow for example exhibit deterministic behavior that does not depend on computation or communication delays.

The language modularization and multi-staged transformations introduced in Chapter 6, however, explicitly allow extending the proposed technology mapping and providing additional mappings.

Part IV.

User Perspective

Chapter 8.

Toolchain

So far, the previous chapters examined the proposed design process from a conceptual perspective (Chapter 2 - Chapter 4) and a developer's perspective (Chapter 5 - Chapter 7). However, while a metamodel and homogenization of the domain is a value by itself, this work targets at supporting the development for the *user*, i.e. the domain expert who wants to describe and evaluate a motion primitive architecture or experiment. Formulation and execution of architectural hypotheses should be easy and fast for a domain expert without the need to become an expert in software engineering or general-purpose languages (GPLs) like C++, Java, or Python.

Based on the introduced metamodel and the domain-specific languages (DSLs) detailed in Chapter 6, the formulation of these architectures can be done on a much higher level of abstraction with languages that make the concern of the domain much more explicit. To evaluate these architecture hypotheses on robot platforms, either in simulation or in hardware, their specification has to be made executable. This chapter introduces an integrated development environment (IDE) that allows specifying an architecture hypothesis on a high level of abstraction and making it executable by means of code generation. The IDE together with the DSLs provide features that are typical for IDEs, such as rich-text editing, syntax highlighting, code completion, but additionally leverage the semantics DSLs specification for model validation, code generation, a component repository, and domain-specific visualization.

The IDE-typical features are briefly discussed in the context of this work in Section 8.1. The DSL specific features of the IDE are discussed in more detail in terms of validation in Section 8.2, a domain-specific component repository in Section 8.3, and generation of system visualization and code generation in Section 8.4. Section 8.5 concludes how all of this is deployed and shipped together with the software architecture introduced in Chapter 7 to provide a domain-specific development environment for specification and execution of motion primitive architectures.

8.1. Integrated Development Environment

One means to provide support during the development process is to supply a rich IDE for the introduced DSLs. IDEs offer comprehensive help to programmers for software development, usually consisting of editors, code completion and syntax highlighting, and build automation tools. The language workbench Jetbrains Meta-Programming System (MPS) [Jetbrains.com, 2003] used in this work supports many of these aspects

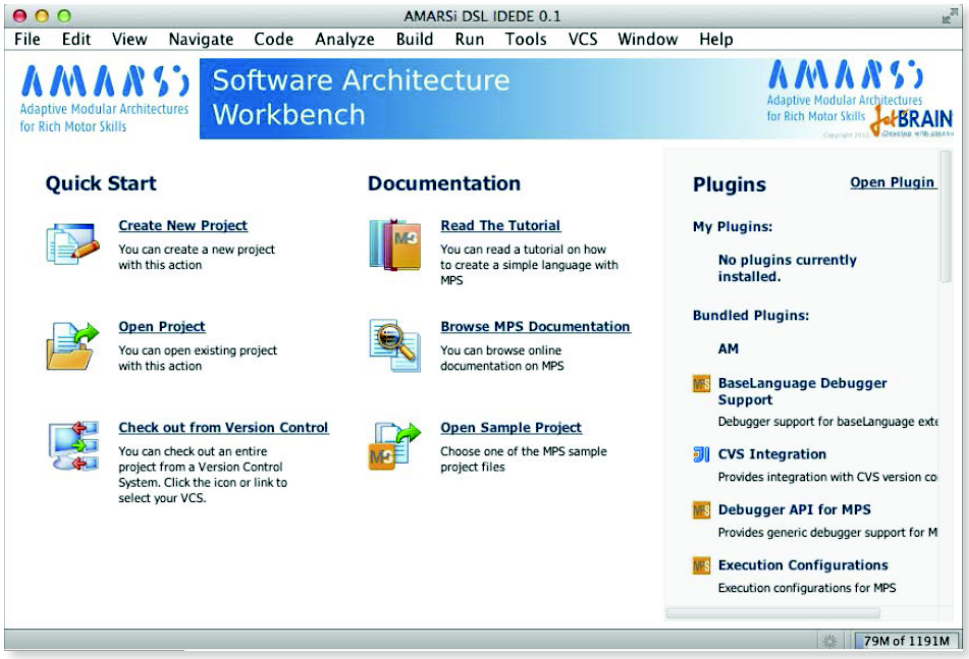


Figure 8.1.: Customized, MPS-based IDE themed for the AMARSi project bundled with the languages introduced in Chapter 6.

and allows bundling and deploying a customized DSL IDE together with language plug-ins.

This customized DSL IDE can be bundled and shipped to the domain expert to explore the design space of the motion primitive architecture domain, while at the same time *restricting them* to the consolidated abstraction and therefore maintaining compatibility and composability of the motion primitive architectures. Fig. 8.1 shows a screenshot of the "AMARSi Motion Primitive Workbench", a customized IDE as it was bundled and customized during the AMARSi project. The IDE was not just themed in the project context, but was bundled with the languages introduced in Chapter 6. As it is based on the free Jetbrains Meta-Programming System, it can be distributed as free software and is platform-independent, i.e. available for Linux, Windows and Mac. Fig. 8.1 shows a screenshot on a Mac computer.

Fig. 8.2 shows the editing view of a recent version of the DSL IDE. The upper left part shows the logical view of the project, which contains one solution[1] with the language fragments. The upper right depicts the projectional editor, showing the textual projection of a Coordination DSL fragment. While it looks like a textual editor, editing actually operates on the abstract syntax tree (AST), as explained in Section 6.1. The lower left part of the screenshot shows the AST of the current language fragment in the *Node Explorer*.

[1] "Solution" is MPS' term for the collection of language fragments that describe an application.

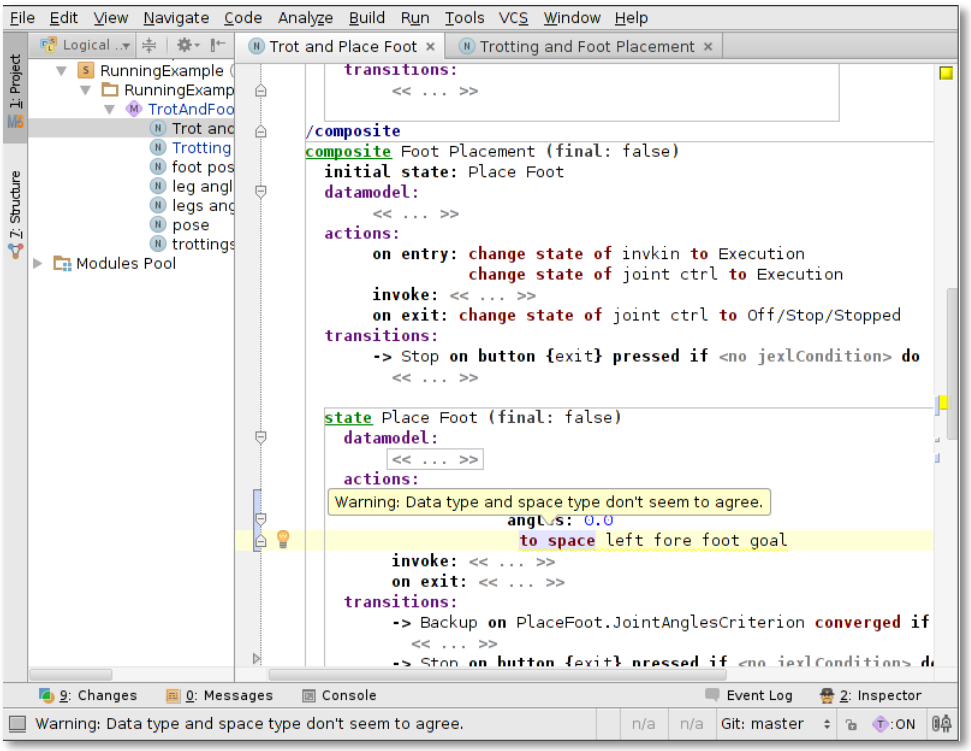


Figure 8.2.: Screenshot of the DSL IDE editor view. The upper left part shows the logical view of the project; the upper right depicts the projectional editor.

Since the editor is just manipulating the AST and the concrete syntax is just a projection, the editor actually enforces syntactical correctness by not allowing to manipulate text in a way that does not comply with the textual projection of the AST.

8.2. Validation

One important advantage of the semantic abstraction when using domain-specific languages to express applications is the possibility to validate its underlying models on the semantic model-level, where typically GPL programs can just be validated on a syntactical level. Validation is based on the constraints and the type system included in the DSLs as discussed in Section 6.3. MPS validates these constraints dynamically and instantaneously during specification.

Fig. 8.2 shows a basic example where the user is warned about a type incompatibility in a Primitive Coordination DSL action. The `PublishToSpace` action that is part of the Primitive Coordination DSL validates the data type to be published, `JointAngles` in the example, and the data type of the `Space` that it is published to. This validation is across language borders and across different viewpoints. The `Space` and `Space Type` are part of the Motion Primitive DSL, the data type is part of the Types DSL, and

both are defined in the structural viewpoint of the system. The validation rule is part of the Primitive Coordination DSL and therefore part of the behavioral viewpoint, but can follow the reference to the `Space` in the structural viewpoint.

This is an example of a validation step that helps avoiding expensive debugging at run time and therefore repeating costly experiments. This concrete validation prevents an error that would not be detected by a compiler, but would lead to an error during run time and is hard to pin down and debug. If executed with the technology mapping introduced in Chapter 7, the `JointAngle` would be serialized and sent over the network with the Robotics Service Bus (RSB) middleware and then tried to be deserialized on the receiving side. The receiving process would assume a different data type according to the `Space Type`, which would most probably result in a C++ segmentation fault. The constraints included in the DSLs can detect this error, providing valuable development support for the user.

8.3. Component Repository

The deployment descriptors introduced in Section 7.4 that describe existing software artifacts implemented against the programming model can be used to provide a repository of software components compatible with the metamodel and the DSLs.

While MPS' default format to store language fragments and models is an eXtensible Markup Language (XML) based format, it provides the so-called *Stubs* mechanism to define custom formats to persist and load models. By using this stubs according to the XML based format introduced in Section 7.4, MPS can read these descriptors and instantiate language models according to the software artifacts they represent.

Fig. 8.3 (upper left) shows how this looks like to the user of the DSL IDE. The models instantiated from the deployment descriptors are made available in a repository, available in the file system and loaded from deployment descriptors. Since they are available as language concepts, they are available in the code completion. Fig. 8.3 shows an example, where an `Adaptive Module` is added to a `Reaching Controller` and in addition to the default `Adaptive Module` concept, the specific `Adaptive Modules` from the repository are suggested. Accordingly, `Dynamical Systems` and `Criterions` from the repository are suggested when editing the respective parts of the model.

A similar mechanism was used and implemented outside this work to create all `Data Types` of the Types DSL, by parsing the data type descriptions of the Robotics System Types (RST) library.

8.4. Code Generation

In addition to supporting the user with modeling motion primitive architectures and motion primitive based applications, one of the main goals of this work is to make these models executable on robot platforms, i.e. support the software development through automation. There are different usage patterns to achieve this with DSLs [Mernik

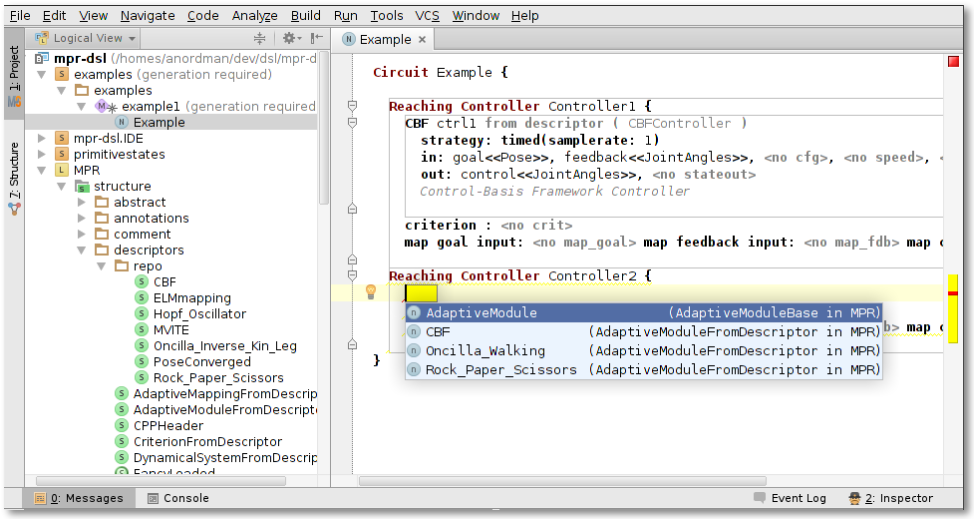


Figure 8.3.: Using models from the repository in the Motion Primitive Workbench. The models are loaded from the repository and available as language elements (left), and thereby available in the editor (right).

et al., 2005, Spinellis, 2001]. The toolchain introduced with this work uses code generation in order to make the models executable for the reasons discussed in Section 6.3. Code generation in this work is realized by chaining model-to-model transformations (M2Ms) and model-to-text transformations (M2Ts). While this the transformations are already discussed from a developer's perspective in Section 6.4, this section now discusses generation of all software artifacts provided to the user of the toolchain.

In MPS, code generators are defined as part of the DSL transformations by either defining M2M transformations to MPS base languages or with the *TextGen* aspect, as discussed in Section 6.4. Fig. 8.4 provides an overview of the M2M and M2T transformations. Gray boxes and dashed arrows refer to the M2M transformations already discussed in Section 6.4.2, black boxes and solid arrows show the M2T transformations. The adapter languages (cf. Section 6.3) are left out for the sake of clarity.

8.4.1. System Visualization

The first type of generated artifacts during development of the presented approach was system-level visualization. This served as an early validation of the introduced models and provided a good overview on existing motion primitive architectures modeled with early iterations of the introduced metamodel and DSLs as demonstrated by Nordmann and Wrede [2012].

Code generation for the system visualization is based on the Graph DSL, to which M2M transformations exist from the Motion Primitive DSL as well as from the Coordination DSL as shown in Fig. 8.4 so that a graph-like system visualization is generated representing the `Circuit` as well as the `State Machine`. The Graph DSL generator

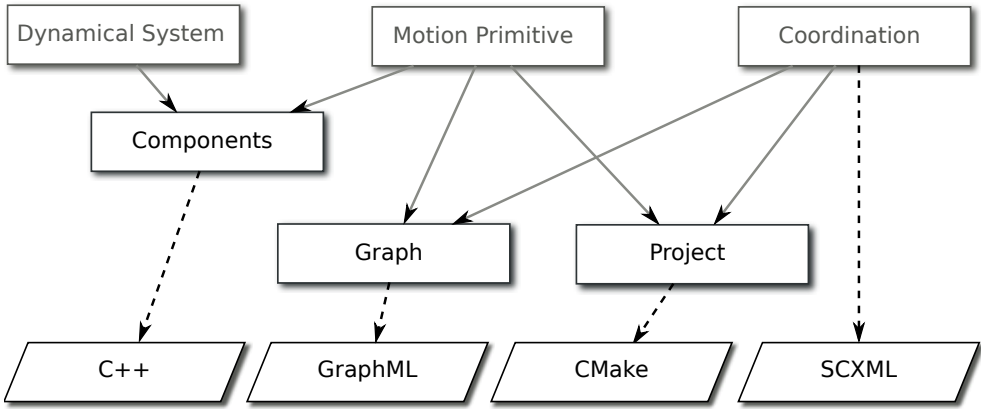


Figure 8.4.: Code generation from higher abstraction languages down to the generation artifacts. Solid arrows indicate the model-to-model transformations; dashed arrows indicate the model-to-text transformations generating executable code and system visualization.

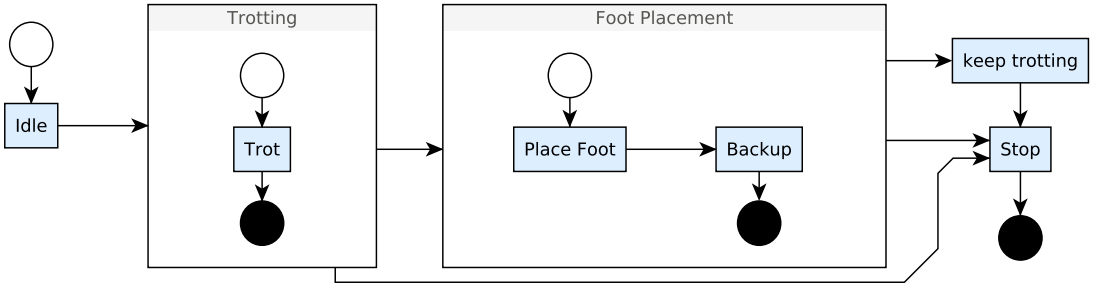


Figure 8.5.: Auto-generated visualization of the coordination of the running example. The white boxes represent the composite states, the blue boxes the states.

therefore targets MPS' XML base language, similar to the generation of State Chart XML (SCXML) as discussed in Section 6.4.3.

Result of this code generation step are two GraphML files, the first one visualizing the structural architecture, the `Adaptive Components`, `Adaptive Modules`, and `Spaces`, the second one visualizing the behavioral aspects, the `State Machine`, its `States` and `Transitions`. As an example, Fig. 8.5 shows the auto-generated visualization of the coordination of the running example. The white boxes represent the composite states, the blue boxes the states. Layouting was done with *yed*, a free-to-use GraphML editor that provides powerful auto-layouting features.

8.4.2. Executable code

The different code generators generating the executable GPL code act as the *implementor* of the components [Völter et al., 2013] so that the motion primitive experts do not have to do the implementation by themselves. For this reason, the code generators

```
<rsb:listener rsb:type="rst.flexirob.devices.RobotState" rsb:scope="$[scope]">
  <rsb:filter rsb:class="mpr.scxml.RobotConvergedFilter">
    <affected>$[affected]</affected>
  </rsb:filter>
</rsb:listener>
```

Figure 8.6.: MSM transformation rule of the Primitive Coordination DSL for the `Robot Converged Condition`.

proposed in this work target the generation of the entire executable code so that no additional manual implementation has to be done (**NFR8**).

8.4.2.1. SCXML Code

Generation of SCXML code from the Coordination DSL is already introduced in Section 6.4. In the presented toolchain, however, the user designs the coordination by additionally using the Primitive Coordination DSL for the motion primitive specific aspects. Fig. 8.6 shows an exemplary transformation rule for one of the Primitive Coordination DSL `Conditions`: `RobotAffected`. When the code generator transforms the Coordination DSL model, the `State Machine` with the contained `States`, concTranss and `Dynamical Systems` to SCXML, it will find and execute the according transformation rules when a Primitive Coordination DSL concept is found.

Result of this code generation step is an SCXML file, which performs the intended coordination of the motion primitives when executed with the SCXML engine introduced in Section 7.2.2. The rule shown in Fig. 8.6 reduced the `Robot Converged Condition` to SCXML code that is configured to receive a particular `Data Type` over a particular `Scope`, and checking it with the RobotConvergedFilter, which is part of the SCXML engine.

8.4.2.2. C++ Code

The generators for the executable C++ code are M2T transformations as part of the Component DSL. These generators generate two different kinds of C++ artifacts: i) C++ components representing `Adaptive Modules`, `Criterions`, `Mappings`, and `Dynamical Systems`, each with a header and an implementation file, and ii) a C++ main file instantiating and configuring the components properly.

The `Circuit` is mapped to the C++ main file instantiating and configuring its contained components properly. Since the Motion Primitive DSL model is already mapped to Component DSL models as introduced in Section 6.4, the concepts are already quite close to the generated code so that the M2T rules are not complex. Fig. 8.7 shows a small snippet from the *TextGen* rules that generate the component instantiation and configuration. Code generation at this point is mainly looping over all components of the Component DSL model, mixing static C++ code and properties from the Component DSL model, e.g., component names, port names and types, and scopes. For the models from the component repository not only the component instantiation and

```
foreach component in node.components {
  append {  } {ComponentPtr } ${component.getComponentName()} { = ComponentPtr(new } ${component.getClassname()}
      {("} ${component.name} {"));} \n;
  append {  } ${component.getComponentName()} {->setProcessingStrategy(} ${component.strategy.getInitializer()}
      {);} \n;

  foreach port in component.inputports {
    append {  } ${component.getComponentName()} {->configureInputPort("} ${port.name} {", PortConfiguration::}
        ${port.config} {("} ${port.getScopeString()} {"));} \n;
  }
  foreach port in component.outputports {
    append {  } ${component.getComponentName()} {->configureOutputPort("} ${port.name} {", PortConfiguration::}
        ${port.config} {("} ${port.getScopeString()} {"));} \n;
  }
```

Figure 8.7.: Snippet of the *TextGen* rules for C++ main file generation. The M2T transformation loops over all components of the Component DSL model, mixing static C++ code and properties from the Component DSL model, e.g., component names, port names and types, and scopes.

configuration is generated, but also preprocessor commands to include the required headers according to the specification inside their deployment descriptors.

Similar M2T transformations transform the Adaptive Modules, Criterions, Mappings, and Dynamical Systems that are specified in the DSL into according C++ headers and implementation files. All models from the component repository are ignored at this point, as their just need to be instantiated and configured.

The Adaptive Modules, Criterions, and Mappings are transformed into components inheriting from according libamarsi abstractions; cf. Section 7.2.1. These files contain the business logic, e.g., the component for the Adaptive Module contains the C++ code to handle state changes, incoming data, forwarding it to the Dynamical System and publishing the output of the Dynamical System. The Dynamical System is transformed to a C++ class. The code generated from the Dynamical System Expression must fit into the surrounding C++ code, which is achieved fitting the generated C++ code into the code skeleton provided by the libamarsi Dynamical System interface.

8.4.2.3. Project Configuration

To be able to easily compile and execute the generated C++ code, according CMake files are generated to check dependencies, build, and install the experiment specified by the motion primitive expert. This is done by code generators from the Project DSL.

The M2M transformations from the Motion Primitive DSL to the Project DSL already include all dependencies from the component repository elements according to their deployment descriptors, which specify their CMake dependencies and file system paths. Proper M2T transformations transform the Project DSL model into CMake files that model all dependencies and include the paths of the software artifacts from the component repository.

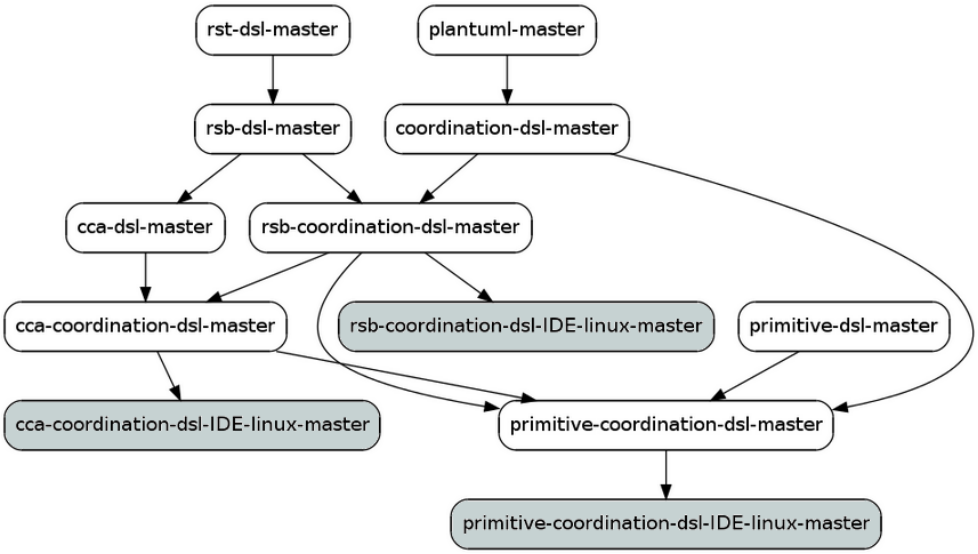


Figure 8.8.: Toolchain and software builds on continuous integration server. The illustration shows the auto-generated dependency graph of the DSL CI jobs. Note that the arrows point into the direction of the build flow, so that the dependency is at the source of the arrow.

8.5. Deployment

The Motion Primitive Workbench introduced in this chapter is bundled and shipped to the domain expert to explore the design space of the motion primitive architecture domain, while at the same time *restricting them* to the consolidated abstraction and therefore maintaining compatibility and composability of the motion primitive architectures.

To ship the IDE, the DSLs and the accompanying software architecture, e.g., the programming model and technology mapping introduced in Chapter 7, in a consistent manner, all three aspects are integrated on a continuous integration (CI) server. The CI server continuously fetches the DSLs from their repositories, build them, integrating them with an MPS installation and providing the entire package for downloading. Fig. 8.8 shows the dependency graph of the language builds and builds of the DSL IDEs:

- The **-dsl-master* jobs build the DSL plug-ins for MPS. When they are built successfully, downstream languages, i.e. languages that depend on the respective language, are triggered automatically so that incompatibilities between languages become visible immediately. This is illustrated in Fig. 8.8 showing a rendering of the dependency graph of the language jobs. The graph was automatically generated by the CI server.

- The **-IDE-linux-master* jobs build the DSL IDEs that include the language

plug-ins and additionally some optional theming as shown in the beginning of this chapter in Fig. 8.1 (manually highlighted in Fig. 8.8 (gray) for the sake of visibility). The IDEs built by these jobs are custom MPS based IDEs in the form of Java applications ready to run on Linux.

The same CI server continuously builds the software libraries of the technology mapping introduced in Chapter 7. The library jobs run static code analysis and unit tests, and trigger package jobs[2] when the tests run without errors. This allows providing binary downloads of the introduced toolchain and software packages of the targeted technology mapping.

Note, that the language modularization allows providing three different DSL IDEs. The *primitive-coordination-IDE-linux-master* builds the Motion Primitive Workbench that is detailed in this chapter and has all DSLs proposed in this work included. The *cca-coordination-IDE-linux-master* builds a DSL IDE with all DSLs included, *except* the Motion Primitive DSL and Primitive Coordination DSL. This IDE provides the same technical features as the Motion Primitive Workbench such as rich text editing of structural models and behavioral models, validation, and code generation, but is targeted at a developer of generic Compliant Component Architecture (CCA) based systems without using the motion primitive models and features. Similarly, the *rsb-coordination-IDE-linux-master* builds a DSL IDE with all DSLs included, *except* the Motion Primitive DSL, Primitive Coordination DSL, Component DSL, and Component Coordination DSL. This provides an IDEs targeted at developers to ease modeling of generic RSB-based systems and their coordination.

While all introduced DSLs are available in the Motion Primitive Workbench, not necessarily all of them have to be used in a modeling project. The proposed language modularization, extension, and composition (LMEC) detailed in Section 6.2 was explicitly designed so that a subset of the languages can be used without dependencies to unused languages.

8.6. Discussion

The toolchain presented in this chapter shows one of the tangible results of the design process proposed in Chapter 4 that is exposed to the user, i.e. the domain expert to be supported in the development of motion primitive architectures. It integrates the DSLs, their editing environment, dynamic constraint checking and model validation, M2M, and M2T transformations for code generation and visualization. The toolchain is available together with the software architecture that is targeted by the code generators that are delivered with the DSLs. It is available for download from a CI server that also provides the DSL IDE itself for download, together with software packages of the targeted technology mapping.

Both together make the model-driven engineering (MDE) process that is targeted by the design process introduced in Chapter 4 available for the domain expert to ease

[2]Currently only Linux packages for two recent *Ubuntu* versions in 32 bit and 64 bit.

and automate development and testing of motion primitive architectures. The author argues that the consistent metamodel available with the DSLs provides the basis for combination of motion primitives while at the same time the programming model and the deployment descriptors provide enough flexibility for research to explore the design space of motion primitive architectures. The entries of this model repository can be used inside the DSL specification and reference software artifacts outside of the MPS environment, which is a non-functional requirement for incorporating and leveraging legacy code (**NFR3**).

The model-driven engineering process enabled by the DSLs and the accompanying Motion Primitive Workbench is exemplified and further discussed in the following chapter.

Chapter 9.

Modeling Motion Primitive Architectures

The toolchain introduced in Chapter 8 allows development and easy editing of motion primitive architectures based on the proposed domain-specific languages (DSLs) and the accompanying software architecture introduced in Chapter 7 makes these models executable. The packaging and deployment described in Section 8.5 makes both easily accessible for the user, i.e. the motion primitive expert. This chapter introduces the intended model-driven engineering (MDE) process for the domain expert, i.e. the workflow to develop, execute, and validate motion primitive architectures based on the proposed languages and toolchain. This workflow is termed *Hypothesis Test Cycle* in the course of this work in accordance with Dittes [2012].

Several of the MDE approaches discussed in Chapter 2 discuss the intended workflow a user follows in their approach. Model-driven engineering workflows targeted to robotics typically support robotics software development process in different stages and levels of abstractions. Typically starting with functional modeling, deployment to the robotics platform, execution, and maintenance. Three recent examples of MDE processes in robotics are now briefly discussed before introducing the workflow proposed in this work: the *Robot Application Development Process* (RAP) [Kraetzschmar et al., 2010] of the BRICS project, the *SafeRobots* framework [Ramaswamy et al., 2014a], and V^{3}CMM [Alonso et al., 2010].

Typical first stages of the development process usually relate to modeling of the problem space, its constraints, and requirements, which the Model Driven Architecture (MDA) refers to as computation independent model (CIM). In BRICS RAP for example, the *scenario-building* phase defines the task of the robot, environment features, constraints and characteristics. An additional *functional design* phase derives hardware requirements and top-level functionalities. In the *SafeRobots* framework [Ramaswamy et al., 2014a], the *problem modeling* phase formally specifies application-specific functional and non-functional requirements, the context or the environment.

A typical next activity is the functional modeling, i.e. knowledge modeling or solution space modeling complying with the functional requirements. This is considered the platform independent model (PIM) in MDA. BRICS RAP distinguishes platform (hardware) modeling and software modeling. The *platform-building* phase determines and configures the robot hardware, the *capability-building* phase designs, specifies and develops the actual software components. The following *system deployment* phase combines these components into a complete application. In the *SafeRobots* framework, this phase is termed *problem-specific knowledge modeling*. It models the solution

space based on the functional requirements from the previous modeling phase. The development process of the V^{3}CMM approach mainly affects this functional modeling phase, comprising the structural modeling, the behavioral modeling, and the algorithmic modeling.

After the functional modeling, the next steps of development phases are usually considering the non-functional requirements and bind the models to the actual (hardware or software) platform. In BRICS RAP, this is part of the *system deployment* phase that maps the application to computational units, allocates resources, and configures the launch management. In the *SafeRobots* framework, this is done in the *problem-specific concrete modeling* phase that reduces the functional model from the previous development phase according to the non-functional requirements and non-functional properties such as timings, confidence, resolution levels, etc.

The next development step to produce executable systems is the code generation or interpretation phase, targeting the runtime of the system. BRICS RAP explicitly mentions additional development phases targeted to runtime, modeling aspects for *benchmarking*, stress testing, and security checks, as well as an explicit *maintenance* phase for system testing, tuning, and extension.

The MDE process proposed in this work is detailed in the course of this chapter and discussed with respect to the three exemplary development process introduced above.

9.1. Hypothesis Test Cycle

The development process proposed in this thesis is termed *Hypothesis Test Cycle* and is targeted to efficiently design and evaluate motion primitive architectures in order to ease and speed-up research for experts of the domain. The basic idea is that a domain expert formulates a hypothesis of a combination of motion primitive to yield rich motion skills, and verifies it on robot platforms.

Similar to the related approaches discussed above, it consists of functional modeling, non-functional modeling and code generation phases, all supported by the toolchain introduced in Chapter 8. Additionally to the related approaches discussed at the beginning of this chapter, the functional modeling phase in this work optionally consists of loading and including existing software artifacts into the domain models from the component repository, as described in Section 8.3.

Fig. 9.1 shows a visualization of the proposed Hypothesis Test Cycle, which is detailed in the following sections. The illustration shows the *circular* process of modeling a motion primitive architecture hypothesis, code generation, experimental verification, and – depending on the result – modifying or refining the initial hypothesis. The rounded rectangles in Fig. 9.1 indicate activities; the regular rectangles indicate models or artifacts. The modeling activities are annotated with the DSLs used in the respective activity.

The Hypothesis Test Cycle starts with a hypothesis of the domain expert concerning a certain motion primitive architecture that combines a set of motion primitives to perform rich motion skills to fulfill a given task. The DSLs introduced in Chap-

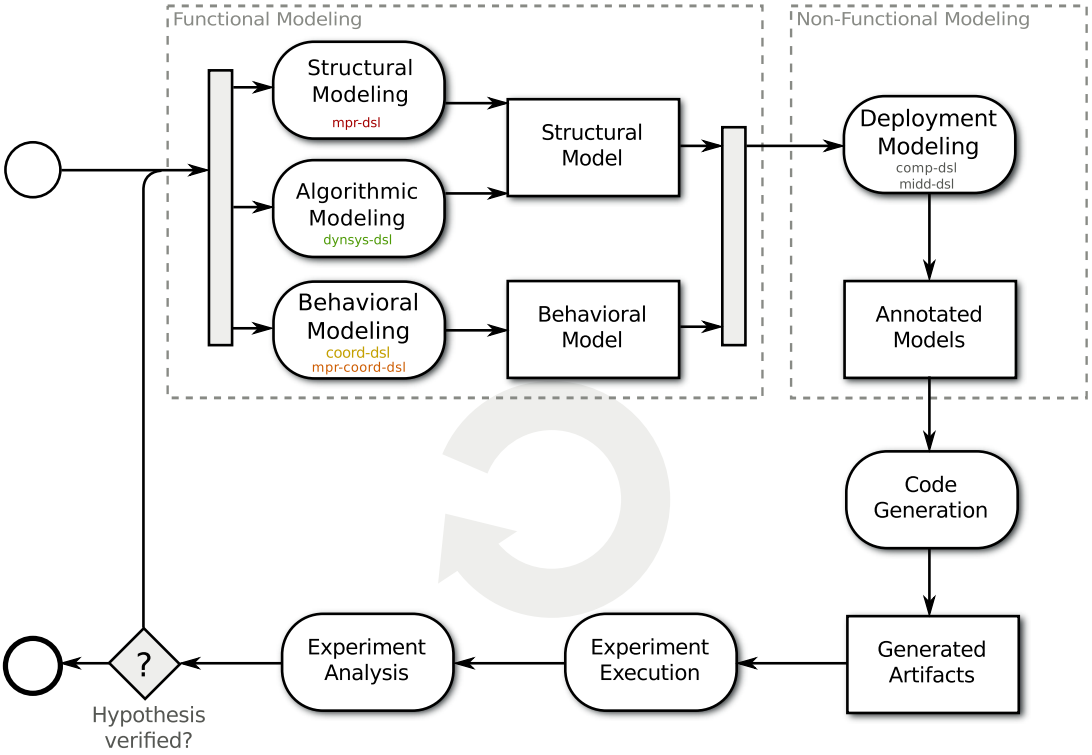


Figure 9.1.: The iterative *Hypothesis Test Cycle* uses the proposed DSLs to model a motion primitive architecture hypothesis for a certain task. Rounded rectangles indicate activities; regular rectangles indicate models and artifacts.

ter 6 allow platform-independent specification of the motion primitive architecture. It usually starts with the modeling of the structural and computational aspects, followed by modeling of the behavioral aspects. This is both detailed in Section 9.1.1. A next step is the specification of the non-functional aspects to bind the experiment to a certain software and robotics platform. This is also supported by DSLs, as indicated in Fig. 9.1 and may be conducted by a different person, filling the role of a robotics or system expert. After the code generation step, performed by the model-to-model transformations (M2Ms) and model-to-text transformations (M2Ts) introduced in Section 6.4 and Section 8.4, the experiment can be executed in simulation or on hardware to validate the initial motion primitive architecture hypothesis.

Experiment results may lead to iterative adaption and refinement of the hypothesis models, which is one of the requirements for this process from the research context (**NFR4**) where hypotheses are usually not correct in the first attempt and need to be adapted or even replaced.

9.1.1. Platform-Independent Modeling

The functional modeling in the domain of this thesis is supposed to be done by the domain expert, in this case the motion primitive expert who wants to design a motion primitive architecture for a certain task and evaluate it on a robot. The motion primitive expert creates a new *solution* in the introduced workbench that provides access to all DSLs and transformations introduced in Chapter 6 and the entire tool support introduced in Chapter 8.

The following order of functional modeling steps is not strict and can be mixed and iterated during the design. It is, however, reasonable to start with the structural and algorithmic models before designing the system coordination. This is done by creating a new `Circuit` that is the root concept of the structural model and a container for all `Adaptive Components`, `Adaptive Modules`, and `Spaces`.

Motion Primitives To start the structural modeling, as a first step the `Adaptive Components` and `Adaptive Modules` to fulfill the intended movements and motion skills are designed with the Motion Primitive DSL in terms of `Adaptive Components` and `Adaptive Modules`. `Adaptive Modules` can be selected from the component repository in Section 8.3 or specified within the models. Selection of the motion primitives is depending on the targeted performance such as robustness to perturbations, input and output dimensionality, or learning capabilities, which lies in the competence of the domain expert.

The toolchain provides the option to either select an existing dynamical system from the repository or specify it with the according mathematical expression. In the same way, when adding a new `Adaptive Module`, the user is provided with context help offering to create a new `Adaptive Module` with the a custom `Dynamical System`, or to load an `Adaptive Module` from the repository with an already built-in `Dynamical System`, as exemplified in Section 8.3.

For instance, in the running example introduced in Chapter 5, dynamics of the foot placement motion were first specified in-line with the algorithmic models and in a second design step replaced by the dynamics of the MVITE [Bullock and Grossberg, 1989] controller from the component repository.

The `Adaptive Modules` are created inside a specific `Adaptive Component` type, e.g., a `Reaching Controllers`, a `Tracking Controllers`, or a `Pattern Generator`, depending on their function inside the intended architecture. Depending on the `Adaptive Component` type, `Criterions` may be added to allow coordination based on the `Adaptive Component Status`.

Dynamical Systems If a `Dynamical System` inside an `Adaptive Module` is not loaded from the repository, it can be specified inline using the Dynamical System DSL. `Inputs` and `Outputs` of the `Adaptive Module` can be used inside the mathematical expression of the `Dynamical System` as exemplified in Section 6.3.2.

Space Types and Spaces As specification of the Adaptive Modules and Adaptive Components, definition of Space Types is required to be created to allow connection of the motion primitive. These are added as root concepts from the Motion Primitive DSL and specified by their Data Type and dimension. The required data types are available through the integrated Types DSL.

Definition of the Space Types is a precondition for the specification of the concrete Spaces to handle communication between the Adaptive Components and Adaptive Modules. Often, several Spaces of the same Space Types are required, e.g., for sensor measurements (the robot status) as well as commands sent to the robot, or different parts and limbs of the robot.

Connect Motion Primitives Once the Spaces are established, the different parts of the architecture are connected through the spaces, e.g., connections from the control Output of an Adaptive Module to a Space, or outgoing connections from Spaces to goal or feedback Inputs of the Adaptive Modules.

At this point of the design process, type incompatibilities between Adaptive Modules and connected components might occur that the Motion Primitive Workbench continuously warns about while designing, as shown in Section 8.2. In this case, it might be necessary to equip the Adaptive Components with additional Mappings to resolve the type incompatibility. An example of this case was introduced in the running example, where the foot goal is given in Cartesian coordinates, but the motion primitive operates in the joint space. Fig. 9.2a shows how the user is warned about this and Fig. 9.2b shows how introduction of an appropriate Mapping (Inverse Kinematics) solves this issue.

Another case requiring additional Mappings added to Adaptive Components is anticipated from the domain analysis. Motion primitives and their machine learning capabilities might be robust enough to work in different contexts and operate in different Space Types, e.g., joint space or Cartesian space, joint angles or joint velocities, which can be achieved by adding the corresponding Mappings.

Design the State Machine When the motion primitive are specified, their combination and coordination needs to be designed. This is done with the Coordination DSL designing the different States and their Transitions. It usually is helpful to introduce Composite States to capsule different high-level phases of the experiment, e.g., training phases and phases in which movements are reproduced bases on the training, as exemplified in Chapter 10.

Link the State Machine to the Adaptive Components and Spaces When the States and their Transitions are designed, the extensions from the Primitive Coordination DSL allow connecting these to the structural model, i.e. the Adaptive Components and Spaces.

Actions inside the States can organize learning and execution phases of the existing motion primitives or publish goals for movement. Conditions for the Transitions

```
Space left fore leg command <leg angles>
  connection ingoing from ctrl of PlaceFoot.jointctrl

Warning: Type of space and input port doesn't seem to agree. (rst.geometry.Translation vs. rst.kinematics.JointAngles)
  connection outgoing to goal of PlaceFoot.jointctrl

// Foot Placement with VITE or DynSys Expression
Reaching Controller Place Foot {
  primitive joint ctrl
    strategy: timed(samplerate: 1)
    in: goal<leg angles>, fdb<leg angles>, <no cfg>, <no speed>, <no phase>, <no statein>
    out: ctrl<leg angles>, <no stateout>
    ds: <{ctrl} = {fdb} + 0.1 * ({goal} - {fdb})>
    properties: << ... >>

  criterion : TranslatoryCriterion distance from descriptor (::cca::component::TranslatoryCriterion)
    strategy: timed(samplerate: 1)
    goal: ingoal<<::rci::Translation>>feedback : infdb<<::rci::Translation>>
    status: status<rst<rst.motioncontrol.MotionStatus>>
    properties: {calculate_x=true} {calculate_y=true} {calculate_z=true} {threshold_x=0.01} {threshold_y=0.01}
        {threshold_z=0.01}
    Cartesian Translatory Criterion
  map goal input: <no map_goal> map feedback input: <no map_fdb> map control output: <no map_ctrl>}
```

(a) In the running example, the Cartesian left fore foot goal Spaces is connected with the goal Input of the Adaptive Module. The Adaptive Module, however, operated in joint space, which results in a warning about incompatible Data Types.

```
Space left fore leg command <leg angles>
  connection ingoing from ctrl of PlaceFoot.jointctrl

Space left fore foot goal <foot position>
  connection outgoing to goal of PlaceFoot.jointctrl

// Foot Placement with VITE or DynSys Expression
Reaching Controller Place Foot {
  primitive joint ctrl
    strategy: timed(samplerate: 1)
    in: goal<leg angles>, fdb<leg angles>, <no cfg>, <no speed>, <no phase>, <no statein>
    out: ctrl<leg angles>, <no stateout>
    ds: <{ctrl} = {fdb} + 0.1 * ({goal} - {fdb})>
    properties: << ... >>

  criterion : TranslatoryCriterion distance from descriptor (::cca::component::TranslatoryCriterion)
    strategy: timed(samplerate: 1)
    goal: ingoal<<::rci::Translation>>feedback : infdb<<::rci::Translation>>
    status: status<rst<rst.motioncontrol.MotionStatus>>
    properties: {calculate_x=true} {calculate_y=true} {calculate_z=true} {threshold_x=0.01} {threshold_y=0.01}
        {threshold_z=0.01}
    Cartesian Translatory Criterion
  map goal input: Oncilla Inverse Kinematics invkin from descriptor (cca::component::OncillaInvKinLeg)
    in: in<rst<rst.geometry.Translation>>
    out: out<rst<rst.kinematics.JointAngles>>
    properties: << ... >>
    Inverse Kinematics for Oncilla quadruped robot leg.
   map feedback input: <no map_fdb> map control output: <no map_ctrl>}
```

(b) Adding an Oncilla Inverse Kinematics Mapping to the Reaching Controller solved the Data Type incompatibility, cf. Fig. 9.2a as the Cartesian goal Input is now automatically mapped to the joint space.

Figure 9.2.: In the running example, the Cartesian left fore foot goal Spaces is incompatible with the Adaptive Module operating in joint space resulting in a warning. Adding an Oncilla Inverse Kinematics Mapping to the Reaching Controller solves the Data Type incompatibility.

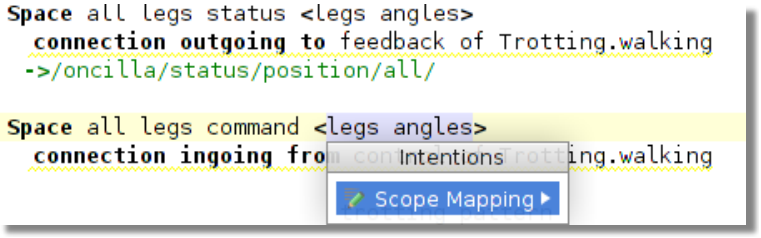


Figure 9.3.: Space annotations of middleware scopes. The annotations are considered by the code generators to configure the middleware accordingly.

between the States can be based on convergence of the motion primitives, i.e. the Adaptive Component Status, on Adaptive Modules changing their lifecycle state, or the robot being interacted with, cf. Section 6.3.4.

9.1.2. Platform-Specifc Annotations

With the functional modeling done, the architecture for a motion primitive experiment is functionally modeled, but is not bound to any platform yet (**NFR6**). In order to do so, non-functional aspects have to be added that influence the M2M transformations to the platform specific model (PSM). The transformations configure the underlying component model and middleware so that the architecture is correctly deployed to the target platform and communication between the motion primitive and the robot is established.

This task is not necessarily done by the domain expert, the motion primitive expert, but intended to be done by a system integrator. As it is implemented as DSL annotations, it is well separated from the functional modeling.

Configure Scopes Several Adaptive Modules rely on extensive proprioceptive feedback from the robot or generate control targets for the robot. Both is organized via Spaces, but these are not yet connected to the robot.

To connect the Spaces to the actual robot platform, the underlying middleware that handles communication of these scopes needs to be configured to use the correct *scopes*[1]. Note that this assumes that the robot is available for our technology mapping, i.e. allowing communication over RSB [Wienke and Wrede, 2011]. This was done for several robot platforms in the course of the AMARSi project, e.g., the Oncilla quadruped robot (Oncilla), the KUKA Lightweight Robot IV (KUKA LBR IV), and the iCub humanoid robot (iCub) within the AMARSi project. Interoperability features of RSB allow connecting various other frameworks such as YARP and the Robot Operating System (ROS) [Wienke et al., 2012].

Fig. 9.3 shows how this annotation looks like. The Spaces of the motion primitive architecture that constitute the interface to the robot, either reading its sensor status

[1] *Scopes* are identifiers of concrete communication channels in the Robotics Service Bus (RSB) [Wienke and Wrede, 2011] that was chosen for the technology mapping.

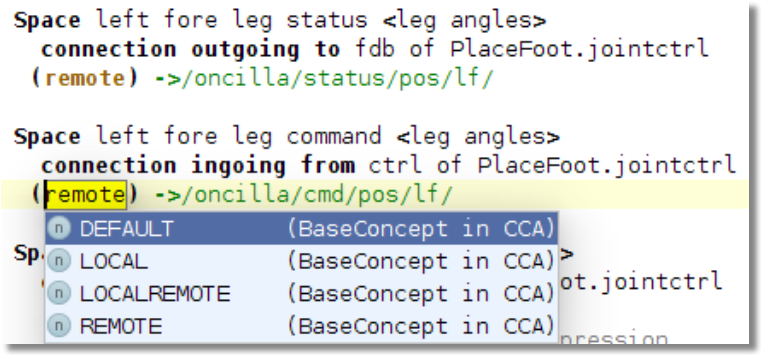


Figure 9.4.: Space annotations of transport configuration. The annotations are considered by the code generators to configure the component framework accordingly.

or setting commands, are annotated with the according scopes of the robot interface. Fig. 9.3 shows a DSL fragment from the running example with one already annotated `Space` and another `Space` currently being edited. The annotation is considered by the code generators to configure the middleware to receive the `JointAngles` status that the Oncilla continuously streams over the scope `/oncilla/status/position/all`.

Configure Transports By default, the component `Circuit` that the modeled motion primitive architecture is mapped to runs in a single process according to the chosen technology mapping (cf. Section 7.2). This is feasible for smaller `Circuits`, but becomes infeasible when the `Circuit` grows and contains heavy computation units such as larger machine learning parts. In this case, the transport annotations provide configuration hints to the underlying component framework that certain communication channels are configured to use a remote transport so that they can be distributed to external machines, i.e. different `Circuits` running in different processes and potentially on different machines distributed in a network.

Fig. 9.4 shows how this annotation looks like. The Component DSL defines transport configuration to determine, if communication between the input and output `Ports`, cf. programming model in Section 7.1.1, is only locally, e.g., in-process communication in the technology mapping introduced in Section 7.2, or remotely over the network. The annotations allow specifying the transport configuration for `Spaces` as shown in Fig. 9.4 to be either only remote, only local, local and remote, or the system default. The M2M transformations that map the Motion Primitive DSL models to the Component DSL models respect those annotations and generate the Component DSL models accordingly.

In the running example, the `Spaces` that represent the connection to the robot are configured to use remote communication, as shown in Fig. 9.4 for one of the `Spaces`. This causes the status `Spaces`, e.g., `left fore leg status` to receive the robot status remotely, in the example shown in Fig. 9.4 all `Joint Angles` sent over the scope

`/oncilla/status/pos/lf`.

9.1.3. Code Generation

When the functional modeling and non-functional modeling is done, code generation makes the models executable by mapping them to general-purpose language (GPL) code through the M2M and M2T transformations introduced in Section 6.4 and Section 8.4 respectively. As an early step to check the modeled motion primitive architecture for plausibility, performing a visual check of the system visualization proved to be very useful in the Hypothesis Test Cycle. Missing connections between `Adaptive Modules`, which is often task-dependent and not easy to detect by the model-checking, can often easily be spotted in the system visualization.

In addition to the system visualization, executable GPL code is generated, as well as the CMake configuration to build the executables, as detailed in Section 8.4.

9.1.4. Execution

While the motion primitive (MP) expert usually performs a visual check of the modeled motion primitive architecture, a system integrator or robotics expert might actually build the executable code, compile, and execute the experiment.

Execution of experiments in robotics is often a complex and labor-intensive task. Reducing the necessity for hardware experiments is therefore a common goal in robotics (**NFR9**). This is especially true for early iterations of an experiment since potential errors can be dangerous and could lead to damage.

First experiments should therefore be executed in simulation if possible. This can often serve as a further step to check the motion primitive architecture hypothesis without the costs and effort of a real robot experiment.

Since the robot interfaces introduced in Section 7.3 provide the same interface for the simulator and the hardware, transition to the hardware is relatively easy once the experiment was successful in simulation.

This can even be exploited when experimenting with motion primitive architectures that include physical human-robot interaction (pHRI), where experimentation cannot easily be done purely simulation as the interaction inputs and triggers are missing. In this case, a single experiment can be conducted on the real robot, while the middleware tooling [Moringen et al., 2013] records the experiment and the experiment data including the pHRI. Afterwards the experiment can be conveniently replayed in simulation due to the same interface.

Experiment data recorded by the middleware tools [Moringen et al., 2013] can also serve as a basis for analysis of the experiment and verification or falsification of the initial hypothesis, motivating refinement, changing or even replacing of the initial motion primitive hypothesis and its respective models, as indicated in Fig. 9.1.

9.2. Discussion

This chapter showed the intended model-driven engineering process targeted to modeling of motion primitive architecture hypotheses based on the proposed domain-specific languages and toolchain.

Similar to state of the art model-driven engineering approaches in robotics, discussed at the beginning of this chapter, the workflow starts with modeling of the platform independent model (PIM), followed by annotation of the platform specific model (PSM) aspects to target the experiment to a certain robot platform. Both steps are independent and can be performed by different roles, i.e. the functional modeling by the motion primitive expert, annotation of non-functional aspects and targeting to a platform by a system integrator and or robotics expert. Other than proposed by MDA, the introduced workflow does not involve explicit modeling of a computation independent model (CIM).

However, the workflow addresses several of the functional and non-functional requirements introduced in Section 4.2. It primarily eases expressing of domain problems and solutions for domain experts (**NFR5**) by allowing specification of motion primitive architectures on higher level of abstraction, independent of in a particular target technology or platform (**NFR6**). This is important so that formulation of motion primitive architectures can survive platform changes, which occur on a regular basis in robotics research, and be reusable across platforms [Dalgarno and Fowler, 2008], which is an explicit non-functional requirement for the proposed approach (**NFR7**).

Chapter 10.

Evaluation and Application

This chapter demonstrates the practical use of the proposed domain-specific languages (DSLs) and the proposed model-driven engineering (MDE) process for modeling and execution of complex motion primitive architectures.

Evaluation of MDE and DSL based approaches such as the one proposed in this thesis is a complex task, as they tend to show their full potential in complex systems and especially in evolution and maintenance of complex systems [Mernik et al., 2005, France and Rumpe, 2007]. Thorough quantitative evaluation of this aspect, however, requires long-term evolution of a complex system with classical development approaches to generate ground truth for comparison with a DSL based approach. This is not feasible for this work. Instead, three, mainly qualitative, evaluation steps are conducted by means of two case studies, which is a suitable research methodology in software engineering [Runeson and Höst, 2009].

One can roughly differentiate two different kinds of evaluation approaches: qualitative and quantitative evaluation. Several MDE approaches in robotics conduct qualitatively evaluation by means of conceptual discussions based on examples, e.g., discussing portability of the semantics to different platforms as done for example by Trojanek [2011], Reckhaus et al. [2010], Klotzbücher et al. [2011]. Laet et al. [2012c] model several typical use cases and show how common errors can be avoided by using its proposed semantics.

Özgür [2007] discusses four different quantitative benefits and corresponding metrics that can be used to evaluate a model-based approach and can serve as a best practice. *Efficiency* for example can be evaluated in terms of performance and memory utilization. Frigerio et al. [2012b] benchmark the generated C++ controller code in its intended use case on platforms with different numbers of degrees of freedom (DoF). Efficiency in this sense was not focus of this work, though. Another measure is *scalability* in terms of compilation time and system size. *Productivity* can be measured in terms of size, effort, or number of change requests. This is for example evaluated by Ringert et al. [2013] and Romero-Garcés et al. [2013]. Both evaluate the usage of a DSL from the developer's perspective against classical approaches by means of empirical software engineering. Non-functional aspects they covered comprise time spent for learning of the technologies, effort for fixing bugs, component reuse, and complexity of understanding reused software artifacts. Klotzbücher et al. [2011] conducted hardware experiments on a PR2 and a KUKA Lightweight Robot IV (KUKA LBR IV) and analyzed the necessary number of lines of code for platform-independent

and robot/framework-specific code. A quantitative aspect that is often a main concern of DSLs is their increased *Expressiveness* (sometimes also "expressivity") over general-purpose languages (GPLs). This typically means that programs are shorter and their semantics are easier to access by processing tools. Programs expressed using a DSL can and should be significantly more concise than expressed in GPLs. This quantitative aspect is evaluated in this work based on a complex real-world example.

This chapter discusses two complex use cases realized with the proposed languages and workflow. Section 10.1 discusses expressiveness and completeness of the languages in two use cases, Section 10.2 conducts a quantitative evaluation of the expressiveness in terms of source lines of code (SLOC) in the second use case.

10.1. Qualitative Evaluation

A qualitative evaluation of the proposed approach is done in two complex industrial use cases concerning two different qualitative aspects. The first use case discussed in Section 10.1.1 is from the *Shop Floor Logistics and Manipulation* task of the European Robotics Challenges (EuRoC)[1] focusing on mobile manipulation in uncertain environments and requires adaptive motion generation capabilities. The second use case discussed in Section 10.1.2 is a complex, real-world industrial application including motion generation integrated with external perception, physical human-robot interaction (pHRI) and machine learning.

10.1.1. Case Study: Shop Floor Logistics and Manipulation

This section discusses Task 4 of the *Shop Floor Logistics and Manipulation* task of the European Robotics Challenges (EuRoC). The EuRoC Challenge 2 is largely a challenge of integration and coordination of different functionalities such as perception, object detection, motion planning, etc.

The task of the EuRoC Challenge 2 is to control a simulated KUKA LBR IV industrial robot arm mounted on an omni-directional mobile platform to grasp objects and place them in defined target areas. The main objective of the challenge is to assess the robot's ability to grasp and place objects in an unstructured and uncertain environment. Position of the objects is a priori unknown, shape, color, size, and mass of the object is known with some uncertainties. Obstacles unknown in position, shape, size, and color may obstruct the robot's path to the objects and their respective target areas [European Robotics Challenges, 2014]. The KUKA LBR IV robot is placed between the objects and target areas, potentially out of reach. Once all objects are picked up and placed or the time is over the task ends. Fig. 10.1 shows a screenshot from the simulated environment with the KUKA LBR IV, obstacles, the manipulation objects, and their respective target areas. The EuRoC consortium provides a simulation environment that *"encapsulates all low-level control and reflex planning loops of the KUKA LBR IV"* [European Robotics Challenges, 2014].

[1] Cf. EuRoC project: http://www.euroc-project.eu/

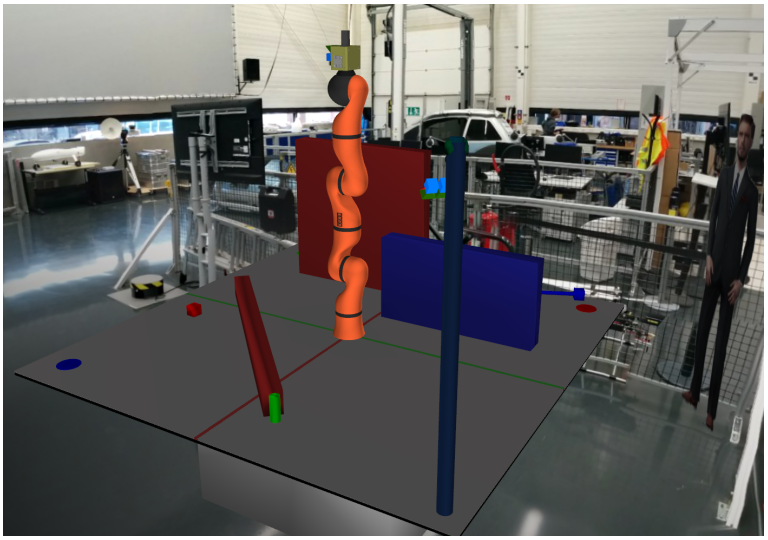


Figure 10.1.: Task 4 of the *Shop Floor Logistics and Manipulation* task of the European Robotics Challenges. The screenshot shows the simulated environment with the obstacles (large blue and red boxes), the manipulation objects (small red, blue, and green objects), and their respective target areas (small red, blue, and green discs on the table).

The base of the robot arm is freely movable in 2D on the table; the robot itself is a redundant manipulator with 7 DoF. The end-effector of the robot is quipped with a 3D time of flight (ToF) camera and a gripper. Small objects (red cube, green cylinder, and blue composite object), their target areas (small colored discs on the table), and larger objects are placed on a table together with the obstacles, as shown in an exemplary screenshot of the simulation environment in Fig. 10.1. The pole in the closest corner of the table in Fig. 10.1 holds an additional ToF camera.

During participation in the European Robotics Challenges challenge, the coordination aspect of the application and its architecture was completely modeled in the Middleware Coordination DSL and successfully employed[2]. This was done based on the rather generic Coordination DSL with the middleware extensions from the Middleware Coordination DSL introduced in Chapter 6, yet *without* the motion primitive extensions from the Primitive Coordination DSL. The software components of the system were developed in Compliant Component Architecture (CCA) and Robotics Service Bus (RSB) and therefore compatible on a technical level. At this time of the challenge, neither the Middleware DSL for structural description of the RSB system, nor the Component Coordination DSL extensions were available. This is briefly explained in Section 10.1.1.1.

To evaluate the ability to make formulation of motion primitive architectures easier

[2]The system successfully succeeded in the qualification stage, "Stage I QUALIFYING: Simulation Contest".

and more concise (**NFR5**), modeling of the motion control part of this application and its coordination is redone by also using the motion primitive (MP) DSLs, i.e. modeling the structural part with the Motion Primitive DSL and expressing the according coordination aspects with the Primitive Coordination DSL. Since the DSLs proposed in this work are targeted to motion primitive architectures, only the system parts responsible for motion control are modeled with the motion primitive specific DSLs and compared with the original modeling. The perception and planning components are considered as external components in accordance with Section 3.3. This is explained and compared in Section 10.1.1.2 and discussed in terms of expressiveness and robustness. Instead of presenting both (the generic and domain-specific) models in detail, major differences between the two approaches are highlighted and discussed on a qualitative level.

10.1.1.1. Modeling with Middleware Coordination DSL

Three RSB processes were implemented for the vision sub-system, the motion control subsystem consists of three controllers and two command filters for the robot arm (all CCA), and additional two controllers for the gripper (position and force controlled, also CCA).

The State Machine consists of 4 Composite States and 28 States, of which 1 Composite State and 8 States are dedicated to motion control. Several of these States consist of sending a goal to the controllers and waiting for them to finish the motion to follow a Transition to the next States, which therefore serves as an example in this section.

Fig. 10.2 shows an exemplary state that incorporates two of the presented functionalities. In its *on entry*-Action a JointAngle is published to the specific Scope the respective controller listens to for commands. The Transition listens for an RSB event on a specific Scope to evaluate whether the robot reached the goal or not. To evaluate this, the payload of the RSB event, the current JointAngle configuration of the robot, is accessed and inspected. A Java Expression Language (JEXL) expression compares the values of the JointAngles with the goal based on a threshold. The outcome of the expression renders the transition either false or true, the latter one resulting in a Transition to the next State.

While this model was working in the EuRoC Challenge, there are some observations regarding expressiveness and robustness. First, the Scopes in this example are hard-coded in the behavioral model due to the absence of a structural model. If the component implementation or configuration changes, i.e. the controllers listening to different Scopes, the specified coordination will fail due to missing communication, which can't be detected at design time. A second observation is the complexity of the Conditions in the two Transitions. The check for convergence of the motion is split up into to checks, one checking if the robot is already at its target configuration (this is necessary when it was already there when the command was sent), the second one checking the convergence status sent by the robot. The JEXL expression in the first condition is a string of > 300 characters, and therefore hard to write, hard to

```
state Initialize Arm (final: false)
  actions:
       on entry: publish rsb event ComponentState:
                   state: EXECUTION
                  to scope: /cca/comp/JointAnglesFilter/statechange/
                 publish rsb event JointAngles:
                   angles: 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
                    to scope: /flexirob/lwr/cmd/all/position

       on exit: publish rsb event ComponentState:
                   state: OFF
                  to scope: /cca/comp/JointAnglesFilter/statechange/
  transitions:
      -> Initial Scene Scan on rsb event on /flexirob/lwr/status/all/position/ with RST rst.kinematics.JointAngles if
          _event.data.data.getAngles(0) lt minDeltaInit and _event.data.data.getAngles(1) lt minDeltaInit and _event...
      -> Initial Scene Scan on rsb event on /flexirob/lwr/status/robotstate with RST rst.devices.flexirob.RobotState
          if _event.data.data.hasMotion() and _event.data.data.motion.toString() eq 'CONVERGED'
```

Figure 10.2.: Example state of the EuRoC coordination to initialize the robotic arm. The conditions are shortened for the sake of visualization, indicated with an ellipsis ("...").

understand, and hard to debug[3].

10.1.1.2. Modeling with Motion Primitive DSL and Primitive Coordination DSL

For a qualitative comparison with the original modeling, the structural aspects of the system are modeled with the Motion Primitive DSL and the behavioral aspects are modeled with the Primitive Coordination DSL. Yet, the structural model only modeled the motion control part of the system, since the Motion Primitive DSL was not able to express the perception and planning parts properly. It is modeled with five Reaching Controllers, one Tracking Controller, each with a Criterion, as well as twelve Spaces.[4]

Through modeling, not only the behavioral aspects the EuRoC system as done in Section 10.1.1.1, but also its structural aspects, stringification[5] of, e.g., Scopes and conditions are completely avoided. Fig. 10.3 shows how this makes description of the State significantly more concise and readable. The Joint Angles are sent to a Space that is connected to an Adaptive Module in the structural model. The same applies to the changing states of when it comes to change the state of a Component. While in Fig. 10.2 the targeted Component is implicit in the Scope, in this model the targeted Adaptive Module is explicitly referenced. When configuration of the Adaptive Module changes, the reference stays valid and mis-configurations are detected by the validation rules of the DSLs at design time. Additionally, the two Conditions in the Transitions shown in Fig. 10.2 are reduced to one Transition with a far simpler Condition referencing the Criterion of the respective Reaching Controller.

[3] The JEXL expression is shortened in Fig. 10.2 for the sake of visualization.

[4] The modeling discussed in Section 10.1.1.2 based on the Motion Primitive DSL and Primitive Coordination DSL could unfortunately not be executed and tested in the EuRoC environment due to timing constraints.

[5] "Stringification" here refers to referencing system parts just by their names stored in a string. This is considered bad practice, as it is not very robust against errors and changes in the code, i.e. errors can't be detected at design time, e.g., by the compiler, but will usually lead to runtime errors.

```
state Initialize Arm (final: false)
  actions:
      on entry: publish JointAngles:
                  angles: 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
                   to space Joint Commands
                change state of JointSpaceController to Execution
      on exit: change state of JointSpaceController to Off/Stop/Stopped
  transitions:
      -> Initial Scene Scan on JointSpaceControl.JointAnglesCriterion converged if <no jexlCondition>
```

Figure 10.3.: Example state of the EuRoC coordination to initialize the robotic arm, using Primitive Coordination DSL extensions.

While the comparison between Fig. 10.2 and Fig. 10.3 shows that specification of the system coordination gets much more explicit and the length of the `State` is significantly reduced in the second case, this comes at the cost of an additional structural model. In the same sense, while the `Condition` is significantly shorter and more concise in the second case, it comes at the price of modeling an `Adaptive Component` with a `Criterion` in the structural model.

However, even then, the more explicit model makes the problem easier to express, easier to understand and accessible for more validation and development support. Errors can be warned about during design time, avoiding errors at runtime that might be costly in experimentation and hard to debug, as discussed in Chapter 6, Chapter 8, and Chapter 9.

Another observation of the use case is that the language modularization, extension, and composition (LMEC) approach and its integration with the toolchain as discussed in Chapter 8 works. The same toolchain and DSLs work for modeling coordination of generic RSB based systems by using only the middleware-specific subset of the proposed DSLs[6], as well as for motion-specific systems using the full set of the proposed DSLs. A second observation is that specification of the system becomes more concise and readable, exemplified with a small example, and – even more important – more explicit and robust. Comparing the more generic modeling based on the Middleware Coordination DSL with the motion-specific modeling based on Motion Primitive DSL highlights some major differences. Note that the comparison is done only on the motion-specific part.

10.1.2. Case Study: Automated Gripping of Laundry with the KUKA Lightweight Robot IV

The second use case comprises a complex robotics setup combining the redundant and compliant KUKA LBR IV, 3D perception and a number of calibration, human-robot interaction, vision, and learning components [Nordmann et al., 2015]. It is used in a real world application arising in the context of an innovation project within the German national leading edge cluster German national leading edge cluster “Intelli-

[6]The Middleware DSL, the Coordination DSL, the Middleware Coordination DSL, and the Graph DSL.

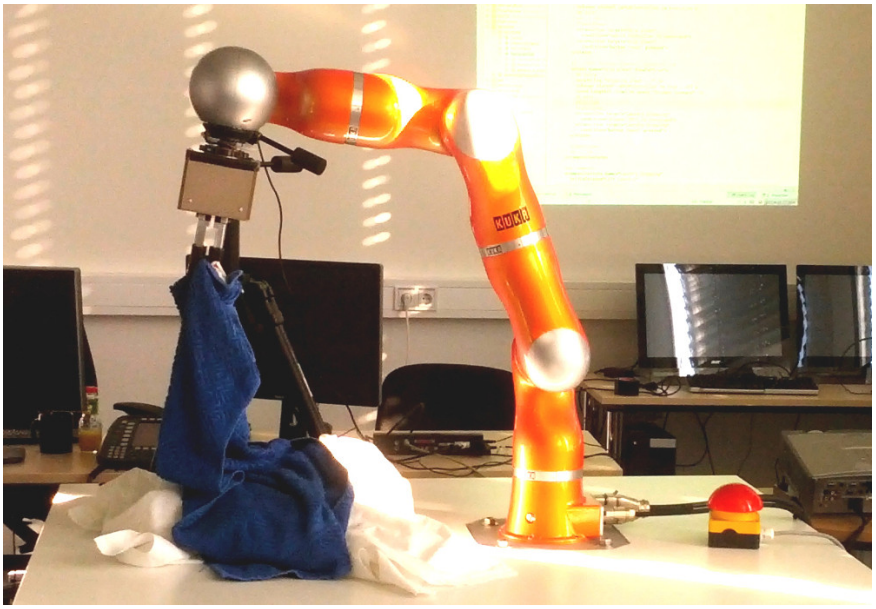


Figure 10.4.: Automated gripping of laundry with the KUKA Lightweight Robot IV.

gent Technical Systems OstWestfalenLippe" (*it's OWL*), where the goal is to reduce time and costs for large automatic laundry washing facilities. The application requires calibrating, interacting, learning kinematics, identifying a pleat in the laundry to automatically grip it, and safely moving the robot into a grasping position to feed the laundry into a further automation process. Although quite typical for robotics application domains, systems of this complexity are not seen often in practice because manual programming and integration of such system would already be very challenging for *handcrafted* development. It rather calls for systematic design methods as the one proposed in this work.

10.1.2.1. Task and Setup

The main objective of this project is automatic handling of laundry, which requires perception of the laundry, detection of a pleat, and control of the robot to feed the laundry into further processing. A further goal of the project is to organize everything in a flexible manner that allows adaption of the system to new environments, which motivates support by the method proposed in this thesis to create a motion primitive architecture that complies with the given objective of the project. The setup for this project comprises the 7 DoF industrial robot arm KUKA LBR IV with a SCHUNK PG70 gripper, as well as a 3D ToF camera for 3D perception, shown in Fig. 10.4. The challenges in designing and developing a complex system of this kind are [Nordmann et al., 2015]:

Control: Gripping of a laundry pleat requires the gripper of the robot to be controlled in all six task dimensions (translatory and rotatory) to reach the pleat and to

be aligned to grip it. Position and orientation of the pleat is previously unknown and requires flexible control of the robot in its task space.

Adaptability: The system is supposed to be deployable in varying environments and near to the machines that are to be fed with laundry for further processing. Restrictions of the robots movement as well as positioning of the 3D ToF sensor are not necessarily a priori known. This calls for flexible calibration and adaption of the system.

This project serves as a case study to evaluate combination of motion primitive (**FR3**) in a complex system and the adaption through machine learning (**FR2**), integration of the motion primitive architecture with external perception (**FR5**), formulation of the architecture in a technology-independent and platform-neutral way (**NFR6**) and execution on a real robot platform (**NFR7**).

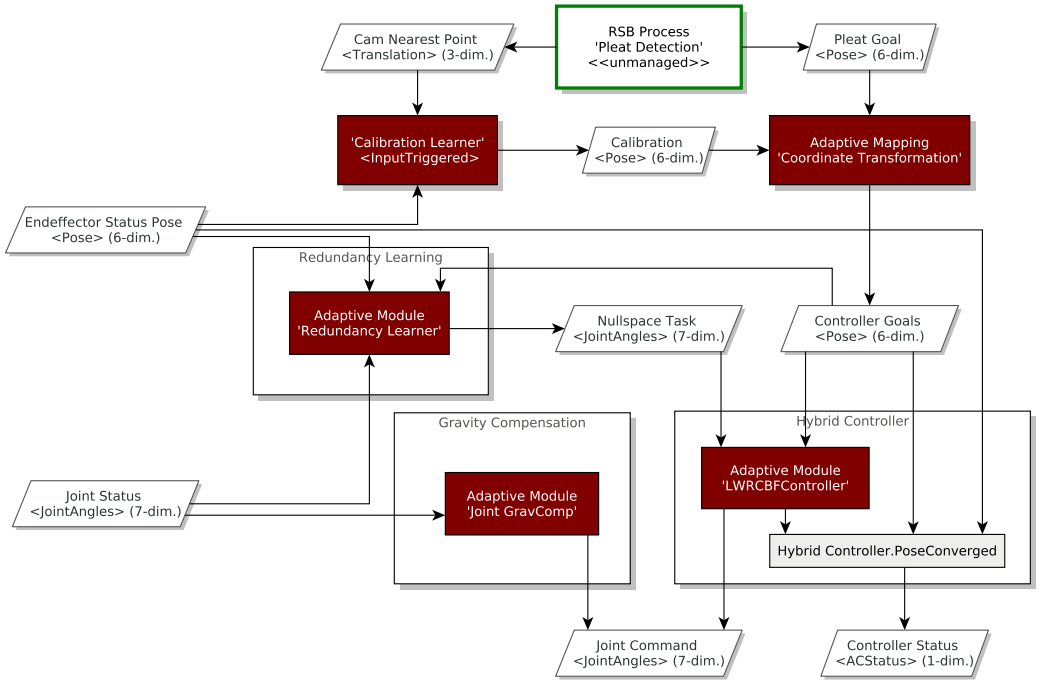
10.1.2.2. Modeling

The motion and manipulation capabilities required for this project are realized with the DSLs and the toolchain proposed in this work. The system is integrated with an external 3D perception stack for pleat detection that was developed in the German national leading edge cluster *it's OWL*.

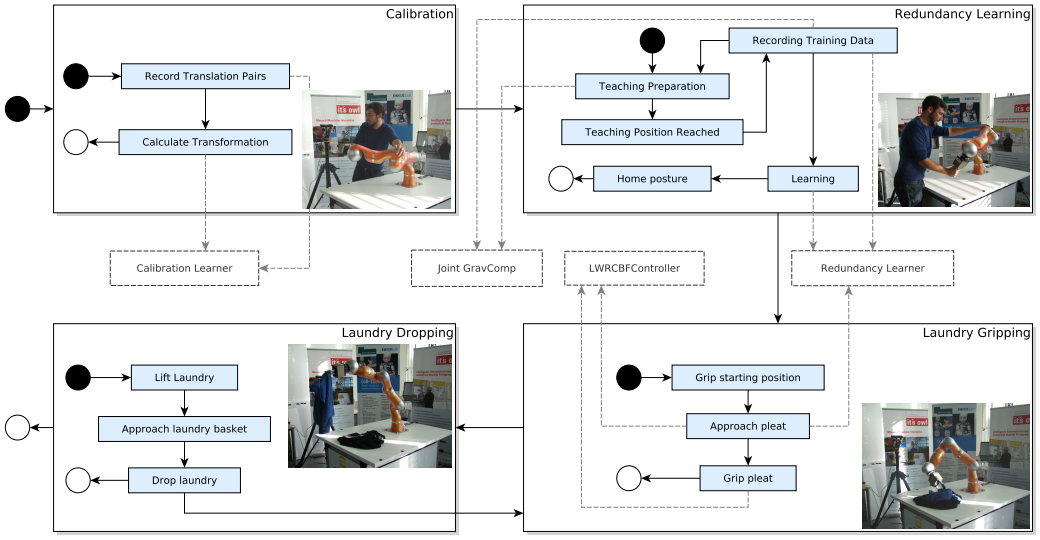
In a first system state, an `Adaptive Module CalibrationLearner` is set into its learning state and learns the 6D transformation between a 3D camera and the KUKA LBR IV IV during a kinesthetic teaching phase, where a human samples the workspace in physical human-robot interaction with the end-effector; cf. Fig. 10.5. In a second system state, the `Adaptive Component RedundancyLearner` containing an Extreme Learning Machine (ELM) [Huang et al., 2006] is configured to learn the desired redundancy resolution in different parts of the workspace. Coordination of the sub-states is done based on interaction forces of the robot and based on the learning states of the ELM `Adaptive Module`. The sub-states manage the component states of the `RedundancyLearner` in the different learning and interaction states.

The third system state is the actual execution where mainly the `Adaptive Component` named `HybridController` is active, moving the gripper to the grip poses given by the external 3D perception component while complying to the redundancy resolution learned previously by the `RedundancyLearner`. Substates like approaching, gripping, and opening the gripper are coordinated by using convergence of movements as a criterion, which is part of the Primitive Coordination DSL.

The complete system involves physical human-robot interaction for calibration and training, 3D perception, and compliant robot control in our framework. Fig. 10.5 shows the auto-generated structural and behavioral system views in Fig. 10.5b. The automatic rendering of structural aspects into the behavioral models allows visualization of both aspects and their dependencies in Fig. 10.5b.



(a) Structural system view with the **Adaptive Components** as *white boxes*, **Adaptive Modules** in *dark-red*, Criterion as *grey box* and Spaces are shown as *parallelograms*. The *green box* on top shows the 3D perception.



(b) **States** are shown in *blue*, *black arrows* represent **Transitions**, *gray-dashed arrows* indicate links to the structural model (*black-dashed*), either as **Condition** for **Transitions**, or triggering different learning states in the **Adaptive Modules**.

Figure 10.5.: Auto-generated system visualization of the industrial use case, manually layouted and reduced for the sake of clarity [Nordmann et al., 2015].

10.2. Quantitative Evaluation

The qualitative evaluation section exemplifies the *Hypothesis Test Cycle* proposed in this thesis in two complex examples and thereby provides a qualitative evaluation of how well the proposed MDE process, DSLs and toolchain are suited to express and develop complex examples of the domain. This section now adds a quantitative evaluation of the presented approach in the automatic laundry-handling case study. Evaluation in terms of use cases is a suitable empirical research methodology in software engineering research [Runeson and Höst, 2009].

The quantitative evaluation compares the lines of DSL code necessary to specify the automatic laundry handling system with the number of generated lines required to visualize and execute the system according to the chosen technology mapping. This provides an estimation for the level of *expressiveness* of the chosen models and DSLs (**NFR5**).

The model-to-model transformation (M2M) introduced in Section 6.4 and model-to-text transformation (M2T) introduced in Section 8.4, transform DSL models into GPL code for system visualization and execution.

For visualization of the architecture of the system, the Motion Primitive DSL and Coordination DSL models are transformed into graph representations, i.e. the rendered graphs shown in Fig. 10.5a and Fig. 10.5b.

For execution, Motion Primitive DSL and Coordination DSL are transformed into C++ code for the CCA components and `Dynamical System` classes, the C++ main file with the component configuration, the glue code for additional C++ components loaded from a software repository based on their deployment descriptors, and three CMake configuration files for software dependency handling and makefile generation. The behavioral aspects are transformed into an additional graph representation and State Chart XML (SCXML) [Barnett et al., 2013] code that represents the specified state machine logic. Furthermore, several C++ files are generated for the connection between the `State Machine` and the structural `Components` of the system.[7] Note, how the cumbersome task of coordinating `Components` state changes and `State Machine Transitions` can be taken care of by the generation tool chain.

Code generation of this case study generated 1 SCXML file, 4 GraphML files, 12 C++ files with 1,248 SLOC, and 3 CMake files from the system specification done in 137 lines of Motion Primitive DSL specifying the structural part and 112 lines of Coordination DSL for coordination, a more compact and explicit specification of the system. See Table 10.1 for all numbers. The auto-generated (and manually adjusted) graph representation of the system, including its structural and behavioral aspects as well as their dependencies, rendered from the generated GraphML files is shown in Fig. 10.5.

The entire system was designed in a language workbench as detailed in Section 8.1. Code generation generated GPL code for execution on the KUKA LBR IV and suc-

[7] This case study was conducted with the first iteration of the technology mapping, i.e. the QT library *scc*, an SCXML compiler, and therefore required generation of C++ files to mediate between the QT state machine and the RSB based system.

	language	number of files	SLOC
Domain-Specific Language	Motion Primitive DSL	1	137
	Coordination DSL	1	112
Technology Mapping	SCXML	1	360
	GraphML	4	4,455
	CMake	4	260
	C++	12	1,248

Table 10.1.: Total source lines of code of the system specification in the proposed DSLs and with the targeted technology mapping.

cessfully executed automated handling of laundry based on the external 3D pleat detection. The generated C++ source code is compiled into two executables, the component circuit and the state machine, and was successfully executed and tested on the KUKA LBR IV in simulation and on the real robot. This was showcased numerous times for project reviews and external guests.

10.3. Discussion

The introduced case studies provide evaluation on the qualitative level, showing the usability of the proposed DSLs and toolchain for specification of complex systems of the targeted domain, and a quantitative evaluation showing the gained expressiveness in the second case study.

Section 10.1.1 evaluates the expressiveness of the models and DSLs by comparing modeling of a complex motion primitive architecture with the more generic Middleware Coordination DSL modeling environment as it was successfully employed in the EuRoC challenge with the more domain-specific modeling of the motion control part of the system with the Motion Primitive DSL and Primitive Coordination DSL. The case study exemplifies how the more domain-specific model leads to a more expressive (**NFR5**), more concise domain model, which helps avoiding common sources of errors, enables additional model checking, and makes it better readable. Section 10.1.2 evaluates the completeness of the approach (**NFR8**) and that the models, DSLs, toolchain and development process are able to express and execute a complex motion primitive architecture. The case study shows the combination of motion primitive (**FR3**) in a complex system and the adaption through machine learning (**FR2**), integration of the motion primitive architecture with external perception (**FR5**), formulation of the architecture in a technology-independent and platform-neutral way (**NFR6**) and execution in simulation ((**NFR9**)) and on a real robot platform (**NFR7**).

Section 10.2 does a quantitative evaluation to further investigate on the expressiveness of the DSLs by comparing the source lines of code of the DSLs code necessary to model the second case study with the source lines of code of the generated code of the targeted technology mapping to visualize the motion primitive archietcture and execute it on a real robot.

Part V.

Conclusion

Chapter 11.

Conclusion

11.1. Summary

Starting point for this work is the observation that a vast amount of robotics research is still done in isolated islands of functionality, experimenting with single aspects rather then fully integrated systems. The targeted domain of motion primitive architectures, however, calls for this integration, since a main – biologically inspired – hypothesis of the domain is that its full potential comes from the combination of the large body of work that has been done on motion primitives.

While biological research supports this hypothesis, its verification in robotics is still hindered by the wide gap between "white board lines and boxes", i.e. the abstract motion primitive on paper and its actual general-purpose language code. The motion primitive is often no longer explicitly visible, but hidden in the source code and interwoven with platform-specific code for perception, planning, etc. The target of this work is to close this gap between concepts and code by providing a conceptual framework that allows description of motion primitive architectures on a higher level of abstraction, while at the same time making them executable by means of model-driven engineering. Hence, a first contribution of this work is a systematic design process to provide a model-driven engineering process and environment for motion primitive architectures, proposed in Chapter 4. The process starts with a domain analysis, introduced in Chapter 3 that leads to the functional and non-functional requirements of this approach.

Chapter 5 to Chapter 7 detail how, based on this approach and the findings of the domain analysis, a metamodel, domain-specific languages (DSLs), and accordingly a programming model are developed to cover the domain. To accommodate for the diversity of the domain, it is separated into a set of concern-specific DSLs following the language modularization, extension, and composition (LMEC) approach proposed by Völter et al. [2013]. The specific language modularization, extension, and composition of this work targeted to the motion primitive architecture domain, detailed in Chapter 6 is a second contribution of this work. Using the proposed models and DSLs eases expression of motion primitive architectures (**FR1**, **FR2**, and **NFR5**) while at the same time restricting its user to the agreed-upon concepts. While this restricts the developer in its freedom of implementation, it ensures compatibility of motion primitives and motion primitive architectures using the proposed approach (**FR3**), which is necessary to investigate the main hypothesis of the domain. At the same time,

the introduced deployment descriptors provide the means for integration of existing legacy work (**NFR3**) as well as prototyping and research of parts that are not yet stabilized in the domain concepts and languages, yet subscribe to the basic concepts and interfaces (**NFR1**).

A third contribution of this work is to make the conceptual framework available for the motion primitive experts to support them in specifying a motion primitive architecture hypothesis and verify it on a robot platform. To achieve this, the proposed languages are available in a DSL integrated development environment (IDE) that provides editing support such as code completion, context help, model checking, and code generation. It thereby forms a convenient environment for the development process detailed in Chapter 9 to formulate motion primitive architectures in platform-independent models (**NFR6**), bind them to a specific robot platform (**NFR7**), and generate the entire code to execute them on of the targeted robot platforms (**NFR8**). While a running example serves as a concrete illustration, Chapter 10 shows the features of the proposed approach, concepts, and toolchain in more complex examples.

Several related model-driven engineering approaches for robotics rose over the last years. The DSL approaches, however, naturally target single domain-specific concerns as discussed by Nordmann et al. [2014], and are often not integrated to a degree that allows generation of complete (in terms of *Completeness*) executable systems. Related approaches that do integrate several DSLs were not found targeting the motion primitive architecture domain. While this work proposed and advertises composition of DSLs and reuse of existing approaches (**NFR1**), it does itself not directly reuse existing DSLs. The problem is that reuse of DSLs is often hindered by technical issues such as technical incompatibilities and incompatibilities on a meta-metamodel level, missing or insufficient documentation, as shown by Nordmann et al. [2014]. This work, however, reuses approaches on a model level, e.g., the Coordination DSL reusing the widely used State Chart model and State Chart XML (SCXML) standard.

Evaluation has shown that the proposed approach and toolchain do work for rather complex robotics systems, exemplified in Chapter 10 and thereby provide a positive answer to the research questions **RQ1** – **RQ3** introduced in Chapter 1. Regarding research question **RQ4**, its introduction into the robotics research context is more ambivalent. A model-driven toolchain such as the one proposed in this work provides a degree of tool lock-in that deters some developers from adopting. Yet, involving developers is often still necessary and important in a research context, especially in the beginning when the DSLs and toolchain are not mature enough. The iterative approach and technical features such as the deployment descriptors, however, helped integrating with legacy work to ease the adoption and thereby proved useful to provide runnable systems, such as the ones discussed in Chapter 10, as it is expected from a research project.

11.2. Outlook

Naturally, the most interesting experiment done with the proposed approach to this point was also the last one and time ran out when things were getting exciting. This outlook shall point to some aspects of particular potential and questions that need further investigation.

Employing LMEC approach in the motion primitive architecture domain worked, as discussed in Chapter 6, Chapter 8, and exemplified in Chapter 10. The introduced approach is therefore proposed for use in other multi-disciplinary domains as well, requiring a dedicated domain analysis and following the steps proposed in Chapter 4 and detailed by this work. While this is only possible in the first place due to the LMEC features of modern language workbenches such as Jetbrains Meta-Programming System (MPS), languages developed with these language workbenches are currently still locked in their particular environment due to meta-metamodel incompatibilities. The ultimate target that the LMEC is pointing to, is however to provide a library of models and languages, such as the ones discussed in Section 3.3, and combine them to more powerful modeling environments. Further development on language workbenches, however, will increase the necessary tool support to get closer to this goal.

Currently, a means to do this in approaches such as the one proposed in this work, is to provide additional transformations to model interchange formats, such as XML Metadata Interchange (XMI), or to other formats such as Unified Modeling Language (UML), Systems Modeling Language (SysML), or Modeling and Analysis of Real Time and Embedded systems (MARTE). This is possible with parallel generators as exemplified in Chapter 6, potentially opening up to a vast amount of additional tool support.

The models and languages developed in this thesis currently mainly target the capability building and system deployment phases of a model-driven development process as proposed by Kraetzschmar et al. [2010]. Extending the models, languages, and toolchain to further development phases is a promising future work. Providing models for run time, for example by modeling the solution space [Ramaswamy et al., 2014c], could enable online selection and combination of motion primitive, which can help exploiting them even further in open-ended scenarios and unstructured environments.

References

IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. *IEEE Std. 1471-2000*, pages i–23, 2000.

S. Adam, M. Larsen, K. Jensen, and U. P. Schultz. Towards Rule-Based Dynamic Safety Monitoring for Mobile Robots. pages 207–218. 2014.

M. Ajallooeian, S. Pouya, A. Tuleu, A. Sproewitz, and A. J. Ijspeert. Towards Modular Control for Moderately Fast Locomotion over Unperceived Rough Terrain. *Dynamic Walking*, (2008):2013, 2013.

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 (JOSER)*, 1(January):3–17, 2010.

A. Angerer, R. Smirra, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif. A Graphical Language for Real-Time Critical Robot Commands. In *Workshop on Domain-Specific Languages and models for Robotic systems*, Tsukuba, 2012.

B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A Survey of Robot Learning from Demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

C. Atkinson and C. Tunjic. Criteria for Orthographic Viewpoints. In *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling - VAO '14*, pages 43–50, York, 2014.

J.-C. Baillie, A. Demaille, Q. Hocquet, and M. Nottale. Events! (Reactivity in urbiscript). In *Workshop on Domain-Specific Languages and models for Robotic systems*, 2010.

J. Barnett, R. Akolkar, R. J. Auburn, M. Bodell, D. C. Burnett, J. Carter, S. McGlashan, T. Lager, M. Helbing, and R. Hosn. State Chart XML (SCXML): State Machine Notation for Control Abstraction, 2013.

L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

G. Biggs, N. Ando, and T. Kotoku. Coordinating Software Components in a Component-Based Architecture for Robotics. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6472 LNAI:168–179, 2010.

R. Bischoff, T. Guhl, E. Prassler, W. Nowak, G. Kraetzschmar, H. Bruyninckx, P. Soetens, M. Haegele, A. Pott, P. Breedveld, J. Broenink, D. Brugali, and N. Tomatis. BRICS Best Practice in Robotics. In *ISR ROBOTIK*, 2010.

E. Bizzi, V. C. K. Cheung, A. D'Avella, P. Saltiel, and M. Tresch. Combining Modules for Movement. *Brain Research Reviews*, 57(1):125–133, 2008.

M. Bordignon, U. P. Schultz, and K. Stoy. Model-Based Kinematics Generation for Modular Mechatronic Toolkits. *ACM SIGPLAN Notices*, page 157, 2010.

D. Brugali and P. Scandurra. Component-Based Robotic Engineering (Part I). *IEEE Robotics & Automation Magazine*, 16(4):84–96, dec 2009.

D. Brugali and A. Shakhimardanov. Component-Based Robotic Engineering (Part II). *IEEE Robotics & Automation Magazine*, 17(1):100–112, mar 2010.

D. Brugali, L. Gherardi, A. Luzzana, and A. Zakharov. A Reuse-Oriented Development Process for Component-based Robotic Systems. In *Simulation, Modeling, and Programming for Autonomous Robots*, 2012.

H. Bruyninckx. Open Robot Control Software: the OROCOS Project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528 vol.3, 2001.

H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali. The BRICS Component Model: A Model-Based Development Paradigm for Complex Robotics Software Systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1758–1764, Coimbra, Portugal, 2013.

D. Bullock and S. Grossberg. VITE and Flete: Neural Modules. *Advances in Psychology*, 62:253–297, 1989.

T. Chaminade, D. W. Franklin, E. Oztop, and G. Cheng. Motor interference between humans and humanoid robots: Effect of biological and artificial motion. *Proceedings of 2005 4th IEEE International Conference on Development and Learning*, 2005 (August 2015):96–101, 2005.

M. Dalgarno and M. Fowler. UML vs. Domain-Specific Languages. *Methods & Tools*, 2008.

S. Degallier. Modeling Discrete and Rhythmic Movements through Motor Primitives: A Review. 2000.

S. Degallier, C. P. Santos, L. Righetti, and A. J. Ijspeert. Movement generation using dynamical systems : a drumming humanoid robot. *IEEE International Conference on Humanoid Robotics*, (1), 2006.

A. V. Deursen and P. Klint. Little Languages: Little Maintenance? *Journal of Software Maintenance: Research and Practice*, 10:75–92, 1998.

S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane. RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7628 LNAI, pages 149–160, 2012.

R. Diankov and J. Kuffner. OpenRAVE : A Planning Architecture for Autonomous Robotics. *Robotics*, (July):–34, 2008.

B. Dittes. Formal System Design for Intelligent Artifacts. 2012.

C. Emmerich, A. Nordmann, A. Swadzba, J. J. Steil, and S. Wrede. Assisted Gravity Compensation to Cope with the Complexity of Kinesthetic Teaching on Redundant Robots. In *International Conference on Robotics and Automation*, 2013.

European Robotics Challenges. Challenge 2 Technical Annex for Stage I, 2014.

A. Feniello, H. Dang, and S. Birchfield. Program Synthesis by Examples for Object Repositioning Tasks. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4428–4435. IEEE, sep 2014.

B. Finkemeyer, T. Kröger, M. Olschewski, and F. M. Wahl. MiRPA: Middleware for Robotic and Process Control Applications. *Proc. of the Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware. IEEE International Conference on Intelligent Robots and Systems, San Diego*, pages 76–90, 2007.

P. Fitzpatrick, G. Metta, and L. Natale. Towards Long-Lived Robot Genes. *Robotics and Autonomous Systems*, 56(1):29–45, 2008.

M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages, 2005.

R. France and B. Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)*, (May 2007), 2007.

M. Frigerio, J. Buchli, and D. 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, 2012a.

M. Frigerio, J. Buchli, and D. G. Caldwell. Code Generation of Algebraic Quantities for Robot Controllers. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2346–2351. IEEE, oct 2012b.

M. Frigerio, J. Buchli, and D. Caldwell. A Domain Specific Language for Kinematic Models and Fast Implementations of Robot Dynamics Algorithms. In *Workshop on Domain-Specific Languages and models for Robotic systems*, 2013.

S. Gerard and B. Selic. In *IFAC Proceedings Volumes (IFAC-PapersOnline)*, volume 17, pages 6909–6913, Seoul, Korea, 2008.

J. L. Gordillo. LE: A High Level Language for Specifying Vision Verification Tasks. In *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, number April, pages 1433–1439. IEEE Comput. Soc. Press, 1991.

R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.

S. Haddadin, M. Suppa, S. Fuchs, T. Bodenmüller, A. Albu-Schäffer, and G. Hirzinger. Towards the Robotic Co-Worker. In *International Symposium on Robotics Research (ISRR2007), Lausanne, Switzerland*, volume 1, 2009.

D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., New York, NY, USA, 1998.

M. Haruno, D. M. Wolpert, and M. Kawato. Mosaic model for sensorimotor learning and control. *Neural computation*, 13:2201–2220, 2001.

T. Henderson and E. Shilcrat. Logical Sensor Systems. *Journal of Robotic Systems*, 1 (2):169–193, 1984.

N. Hochgeschwender, S. Schneider, H. Voos, and G. K. Kraetzschmar. Towards a Robot Perception Specication Language. In *Workshop on Domain-Specific Languages and models for Robotic systems*, 2013.

N. Hochgeschwender, S. Schneider, H. Voos, and G. K. Kraetzschmar. Declarative Specification of Robot Perception Architectures. In *Simulation, Modeling, and Programming for Autonomous Robots*, number August, pages 291–302, 2014.

G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew. Extreme Learning Machine: Theory and Applications. *Neurocomputing*, 70(1):489—-501, 2006.

A. J. Ijspeert, J. Nakanishi, and S. Schaal. Learning Attractor Landscapes for Learning Motor Primitives. *Advances in Neural Information Processing Systems 15 (NIPS2002)*, pages 1547–1554, 2002.

A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal. Dynamical Movement Primitives: Learning Attractor Models for Motor Behaviors. *Neural computation*, 25:328–73, 2013.

Jetbrains.com. Jetbrains Meta Programming System. *http://www.jetbrains.com/mps/*, 2003.

S. Kajita and B. Espiau. Springer Handbook of Robotics. *Springer Handbook of Robotics*, pages 361–389, 2008.

K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA)-Feasibility Study. 1990.

G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel. Design Guidelines for Domain Specific Languages. *9th OOPSLA Workshop on Domain-Specific Modeling*, page 7, 2009.

L. C. Kats and E. Visser. The spoofax language workbench. *ACM SIGPLAN Notices*, 45(10):444, oct 2010.

P. Kilgo, E. Syriani, and M. Anderson. A Visual Modeling Language for RDIS and ROS Nodes Using AToM3. In *Simulation, Modeling, and Programming for Autonomous Robots*, 2012.

M. Klotzbücher, R. Smits, H. Bruyninckx, and J. De Schutter. Reusable Hybrid Force-Velocity Controlled Motion Specifications with Executable Domain Specific Languages. In *International Conference on Intelligent Robots and Systems*, pages 4684–4689, 2011.

G. K. Kraetzschmar, A. Shakhimardanov, J. Paulus, N. Hochgeschwender, and M. Reckhaus. Best Practice in Robotics: Best Practice Assessment of Software Technologies for Robotics. *Best Practice in Robotics*, 2010.

H. Krahn, B. Rumpe, and S. Völkel. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353—-372, 2001.

T. Kröger, B. Finkemeyer, U. Thomas, and F. M. Wahl. Compliant Motion Programming : The Task Frame Formalism Revisited. *Mechatronics and Robotics*, pages 1–6, 2004.

T. D. Laet, S. Bellens, H. Bruyninckx, and J. D. Schutter. Geometric Relations between Rigid Bodies (Part 2): From Semantics to Software. *IEEE Robotics and Automation Magazine*, (September), 2012a.

T. D. Laet, S. Bellens, R. Smits, E. Aertbelien, H. Bruyninckx, and J. D. Schutter. Geometric Relations between Rigid Bodies (Part 1): Semantics for Standardization. *IEEE Robotics and Automation Magazine*, (June), 2012b.

T. D. Laet, W. Schaekers, J. de Greef, and H. Bruyninckx. Domain Specific Language for Geometric Relations between Rigid Bodies targeted to Robotic Applications. In *Workshop on Domain-Specific Languages and models for Robotic systems*, 2012c.

D. Lee and C. Ott. Incremental Kinesthetic Teaching of Motion Primitives using the Motion Refinement Tube. *Autonomous Robots*, 31(2-3):115–131, 2011.

A. Lemme. *Bootstrapping Movement Primitives from Complex Trajectories*. PhD thesis, Bielefeld University, 2015.

A. Lotz, J. F. Inglés-Romero, C. Vicente-Chicote, and C. Schlegel. Managing Run-Time Variability in Robotics Software by Modeling Functional and Non-functional Behavior. In *Enterprise, Business-Process and Information Systems Modeling*, pages 441—-455. Springer Berlin Heidelberg, 2013.

A. Lotz, A. Hamann, L. Ingo, D. Stampfer, M. Lutz, and C. Schlegel. Modeling Non-Functional Application Domain Constraints for Component-Based Robotics Software Systems. In *Sixth International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob)*, 2015.

T. Luksch, M. Gienger, M. Mühlig, and T. Yoshiike. A Dynamical Systems Approach to Adaptive Sequencing of Movement Primitives Dynamical Systems. *Proceedings of the 7th German Conference on Robotics (ROBOTIK)*, 2012.

I. Lütkebohle and S. Wachsmuth. Requirements and a Case-Study for SLE from Robotics: Event-oriented Incremental Component Construction. In *Software Language Engineering for Cyber-physical Systems (INFORMATIK 2011)*, Berlin, Germany, 2011.

K. Martin and B. Hoffman. *Mastering CMake*. Kitware, 2010.

B. Matthias, S. Kock, H. Jerregard, M. Kallman, and I. Lundberg. Safety of Collaborative Industrial Robots: Certification Possibilities for a Collaborative Assembly Robot Concept, 2011.

M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

O. Michel. Webots: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):39—-42, 2004.

J. Moringen, A. Nordmann, and S. Wrede. Cross-Platform Data Acquisition and Transformation for Whole-Systems Experimentation. *ERF2013 Working Session on Infrastructure for Robot Analysis and Benchmarking*, pages 4–5, 2013.

F. L. Moro, N. G. Tsagarakis, and D. G. Caldwell. A Human-Like Walking for the COmpliant huMANoid COMAN based on CoM Trajectory Reconstruction from Kinematic Motion Primitives. In *2011 11th IEEE-RAS International Conference on Humanoid Robots*, pages 364–370. IEEE, oct 2011.

M. Mühlig, M. Gienger, and J. J. Steil. Interactive imitation learning of object movement skills. *Autonomous Robots*, 32(2):97–114, 2012.

F. a. Mussa-Ivaldi and E. Bizzi. Motor learning through the combination of primitives. *Philosophical transactions of the Royal Society of London. Series B, Biological sciences*, 355(1404):1755–1769, 2000.

F. a. Mussa-Ivaldi, S. F. Giszter, and E. Bizzi. Linear Combinations of Primitives in Vertebrate Motor Control. *Proceedings of the National Academy of Sciences of the United States of America*, 91(16):7534–7538, 1994.

S. Nakaoka, A. Nakazawa, K. Yokoi, and K. Ikeuchi. Leg Motion Primitives for a Dancing Humanoid Robot. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 610–615. IEEE, 2004.

K. Neumann, A. Lemme, and J. J. Steil. Neural Learning of Stable Dynamical Systems based on Data-Driven Lyapunov Candidates. *International Conference on Intelligent Robots and Systems*, 1:1216–1222, nov 2013.

V.-c. Nguyen, X. Qafmolla, and K. Richta. Domain Specific Language Approach on Model-driven Development of Web Services Background. 11(8):121–138, 2014.

A. Nordmann and S. Wrede. A Domain-Specific Language for Rich Motor Skill Architectures. In *Workshop on Domain-Specific Languages and models for Robotic systems*, Tsukuba, 2012.

A. Nordmann, C. Emmerich, S. Ruether, A. Lemme, S. Wrede, and J. Steil. Teaching Nullspace Constraints in Physical Human-Robot Interaction using Reservoir Computing. In *Proceedings - IEEE International Conference on Robotics and Automation*, pages 1868–1875, 2012a.

A. Nordmann, M. Rolf, and S. Wrede. Software Abstractions for Simulation and Control of a Continuum Robot. In I. Noda, N. Ando, D. Brugali, and J. J. Kuffner, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7628 LNAI of *Lecture Notes in Computer Science*, pages 113–124, Berlin, Heidelberg, 2012b. Springer Berlin Heidelberg.

A. Nordmann, A. Tuleu, and S. Wrede. A Domain-Specific Language and Simulation Architecture for Motor Skill Exploration. In *Workshop on Software Development and Integration in Robotics (SDIRVIII)*, Karlsruhe, 2013a.

A. Nordmann, A. Tuleu, and S. Wrede. A Domain-Specific Language and Simulation Architecture for the Oncilla Robot. In *Workshop on Developments of Simulation Tools for Robotics & Biomechanics*, Karlsruhe, 2013b.

A. Nordmann, N. Hochgeschwender, and S. Wrede. A Survey on Domain-Specific Languages in Robotics. In *International Conference on Simulation, Modeling and Programming for Autonomous Robots*, Bergamo, 2014.

A. Nordmann, S. Wrede, and J. Steil. Modeling of Movement Control Architectures based on Motion Primitives using Domain Specific Languages. In *International Conference on Automation and Robotics*, 2015.

OMG. OMG Unified Modeling Language TM (OMG UML), Superstructure v.2.3. *Informatik Spektrum*, 21(May):758, 2010.

OMG. Model Driven Architecture Guide Rev. 2.0. Technical Report June, 2014.

A. B. ORMSC. Model Driven Architecture (MDA). Technical report, Architecture Board ORMSC, 2001.

T. Özgür. *Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specific Modeling in the Context of the Model-Driven Development*. Master, Blekinge Institute of Technology, 2007.

E. Oztop, T. Chaminade, and D. Franklin. Human-humanoid interaction: is a humanoid robot perceived as a human? *4th IEEE/RAS International Conference on Humanoid Robots, 2004.*, 2(August 2015), 2004.

P. Pastor, H. Hoffmann, T. Asfour, and S. Schaal. Learning and generalization of motor skills by learning from demonstration. *2009 IEEE International Conference on Robotics and Automation*, 2009.

A. S. Phung, J. Malzahn, F. Hoffmann, and T. Bertram. Get Out of the Way - Obstacle Avoidance and Learning by Demonstration for Manipulation. *IFAC World Congress*, pages 11514–11519, 2011.

M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an Open-Source Robot Operating System. In *International Conference on Robotics and Automation*, 2009.

A. Ramaswamy, B. Monsuez, and A. Tapus. SafeRobots: A Model-Driven Framework for Developing Robotic Systems. In *Intelligent Robots and Systems*, 2014a.

A. Ramaswamy, B. Monsuez, and A. Tapus. Model-Driven Software Development Approaches in Robotics Research. *Proceedings of the 6th International Workshop on Modeling in Software Engineering - MiSE 2014*, pages 43–48, 2014b.

A. Ramaswamy, B. Monsuez, and A. Tapus. Solution Space Modeling for Robotic Systems. *Journal of Software Engineering for Robotics*, 5(May):89–96, 2014c.

M. Reckhaus, N. Hochgeschwender, P. G. Ploeger, and G. K. Kraetzschmar. A Platform-Independent Programming Environment for Robot Control. In *Workshop on Domain-Specific Languages and models for Robotic systems*, 2010.

M. Reichardt and K. Berns. On Software Quality-motivated Design of a Real-time Framework for Complex Robot Control Systems. *Proceedings of the 7th International Workshop on Software Quality and Maintainability (SQM)*, 2013.

M. Reichardt, T. Föhst, and K. Berns. Introducing FINROC: A Convenient Real-Time Framework for Robotics Based on a Systematic Design Approach. (III):1–8, 2012.

F. Reinhart and J. J. Steil. Neural Learning and Dynamical Selection of Redundant Solutions for Inverse Kinematic Control. In *IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pages 564–569, 2011.

R. F. Reinhart and J. J. Steil. Learning Whole Upper Body Control with Dynamic Redundancy Resolution in Coupled Associative Radial Basis Function Networks. *IEEE International Conference on Intelligent Robots and Systems*, pages 1487–1492, 2012.

J. Ringert, B. Rumpe, and A. Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Workshops and Tutorials Proceedings of the 2013 IEEE International Conference on Robotics and Automation*, pages 10–12, 2013.

J. O. Ringert, B. Rumpe, and A. Wortmann. Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In *Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 41—-47, 2015.

H. Ritter, R. Haschke, and J. J. Steil. A Dual Interaction Perspective for Robot Cognition: Grasping as a "Rosetta Stone". *Studies in Computational Intelligence*, 77:159–178, 2008.

A. Rodrigues da Silva. Model-Driven Engineering: A Survey Supported by a Unified Conceptual Model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

A. Romero-Garcés, L. Manso, M. Gutierrez, R. Cintas, and P. Bustos. Improving the Lifecycle of Robotics Components using Domain-Specific Languages. In *Workshop on Domain-Specific Languages and models for Robotic systems*, 2013.

P. Runeson and M. Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

S. Schaal. Dynamic Movement Primitives - A Framework for Motor Control in Humans and Humanoid Robotics. In *Adaptive Motion of Animals and Machines*, pages 261–280. Springer-Verlag, Tokyo, 2006.

S. Schaal, J. Peters, J. Nakanishi, and A. Ijspeert. Learning Movement Primitives. *Robotics Research*, pages 1–10, 2005.

C. Schlegel, T. Hassler, A. Lotz, and A. Steck. Robotic Software Systems: From Code-Driven to Model-Driven Designs. *2009 International Conference on Advanced Robotics*, 2009.

C. Schlegel, A. Steck, D. Brugali, and A. Knoll. Design Abstraction and Processes in Robotics: From Code-Driven to Model-Driven Engineering. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6472 LNAI:324–335, 2010.

C. Schlegel, A. Lotz, M. Lutz, D. Stampfer, J. F. Inglés-Romero, and C. Vicente-Chicote. Model-Driven Software Systems Engineering in Robotics: Covering the Complete Life-Cycle of a Robot. *it - Information Technology*, 57(2):85–98, 2015.

D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

S. Schneider, N. Hochgeschwender, and G. K. Kraetzschmar. Structured Design and Development of Domain-Specific Languages in Robotics. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 231–242, 2014.

B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

A. Shukla and A. Billard. Coupled dynamical system based hand-arm grasp planning under real-time perturbations. *Robotics: Science and Systems*, 2011.

B. Siciliano and O. Khatib, editors. *Springer Handbook of Robotics*. Springer-Verlag Berlin Heidelberg, 2008.

D. Spinellis. Notable Design Patterns for Domain-Specific Languages. *Journal of Systems and Software*, 56:91—-99, 2001.

A. Spröwitz, L. Kuechler, A. Tuleu, M. Ajallooeian, M. D'Haene, R. Moeckel, and A. J. Ijspeert. Oncilla Robot – A Light-weight Bio-inspired Quadruped Robot for Fast Locomotion in Rough Terrain. *Symposium on Adaptive Motion of Animals and Machines (AMAM2011)*, pages 63–64, 2011.

T. Stahl, M. Völter, J. Bettin, A. Haase, and S. Helsen. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

A. Steck and C. Schlegel. SMART TCL: An Execution Language for Conditional Reactive Task Execution in a Three Layer Architecture for Service Robots. In *Int. Workshop on DYnamic languages for RObotic and Sensors systems (DYROS)*, pages 274–277, 2010.

A. Steck and C. Schlegel. Managing Execution Variants in Task Coordination by Exploiting Design-Time Models at Run-Time. *International Conference on Intelligent Robots and Systems*, pages 2064–2069, sep 2011.

A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. *Proceedings of the 28th International Conference on Machine Learning, ICML 2011*, (Ml):609–616, 2011.

U. Thomas, B. Finkemeyer, T. Kroger, and F. Wahl. Error-tolerant execution of complex robot tasks based on skill primitives. In *International Conference on Robotics and Automation*, volume 3, Taipei, Taiwan, 2003.

U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *International Conference on Robotics and Automation*, 2013.

J.-P. Tolvanen and M. Rossi. MetaEdit+. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '03*, number August 2015, page 92, New York, New York, USA, 2003. ACM Press.

P. Trojanek. Model-Driven Engineering Approach to Design and Implementation of Robot Control System. In *Workshop on Domain-Specific Languages and models for Robotic systems*, 2011.

A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM Sigplan Notices*, 2000.

D. Vanthienen and H. Bruyninckx. The 5C-Based Architectural Composition Pattern: Lessons Learned from Re-Developing the iTaSC Framework for Constraint-Based Robot Programming. *Journal of Software Engineering for Robotics*, 5(May):17–35, 2014.

M. Völter. Software Architecture – A Pattern Language for Building Sustainable Software Architectures. *EuroPLoP 2006*, sep 2005.

M. Völter. From Programming To Modeling and back again. *IEEE Software*, 2010.

M. Völter. Language and IDE modularization and composition with MPS. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7680 LNCS:383–430, 2013.

M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering – Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 2013.

M. Völter, B. Kolb, and J. Warmer. Language-Oriented Business Applications, 2014.

T. Waegeman, M. Hermans, and B. Schrauwen. MACOP modular architecture with control primitives. *Frontiers in computational neuroscience*, 7(July):99, 2013.

J. Wienke and S. Wrede. A Middleware for Collaborative Research in Experimental Robotics. In *2011 IEEE/SICE International Symposium on System Integration (SII)*, pages 1183–1190, Kyoto, dec 2011. IEEE.

J. Wienke, A. Nordmann, and S. Wrede. A Meta-Model and Toolchain for Improved Interoperability of Robotic Frameworks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7628 LNAI, pages 323–334, 2012.

S. Wrede, C. Emmerich, R. Grünberg, A. Nordmann, A. Swadzba, and J. J. Steil. A User Study on Kinesthetic Teaching of Redundant Robots in Task and Configuration Space. *Journal of Human-Robot Interaction*, 2(Special Issue: HRI System Studies): 56–81, 2013.

J. Zhang and B. H. Cheng. Model-Based Development of Dynamically Adaptive Software. In *ICSE 2006*, 2006.

Part VI.

Appendix

Appendix A.

Related References by the Author

Nordmann et al. [2012a] present an early iteration of the KUKA Lightweight Robot IV (KUKA LBR IV) based vertical prototype introduced in Section 7.3 and one of the domain examples discussed in Section 3.3 for learning of redundancy resolution in kinesthetic teaching.

Nordmann et al. [2012b] introduce parts of the programming model and technology mapping discussed in Chapter 7 based on the Bionic Handling Assistant, a soft continuum kinematics robot. Short excerpts of this work are included in Chapter 7.

Nordmann and Wrede [2012] present an early version of the proposed model-driven engineering (MDE) approach and domain-specific languages (DSLs) and discuss them on a conceptual level based on the domain examples introduced in Section 3.3.

Nordmann et al. [2013a] presents an early version of the proposed Hypothesis Test Cycle and exemplifies its usage together with a quadruped robot simulation environment. It shows an early iteration of the proposed design process, and details the Oncilla quadruped robot (Oncilla) mock platform introduced in Section 7.3.

Nordmann et al. [2013b] present the Oncilla mock platform and the accompanying software interfaces and architecture. Excerpts of this work are included in Section 7.3.

Wrede et al. [2013] discuss a user study with 49 industrial workers at a medium-sized manufacturing company, teaching the KUKA LBR IV based vertical prototype introduced in Section 7.3. It thereby shows a further iteration step of the introduced programming model and technology mapping.

Nordmann et al. [2014] survey the available literature on DSLs in robotics and discuss them from the perspective of users and developers of model-based approaches in robotics along a set of quantitative and qualitative research questions. Excerpts of this survey are used for the discussion of related approaches in Chapter 2 and the domain analysis in Chapter 3.

Nordmann et al. [2015] present the approach proposed in this thesis and evaluate it in a complex case study for automatic laundry handling with the KUKA LBR IV. Section 10.1.2 is based on the results and discussion of this work.

Appendix B.

Domain Analysis

Following Kang et al. [1990], the domain analysis consisted of a survey conducted with domain experts, namely the project partners of the European AMARSi project, as well as analysis of existing software and systems. Section B.1 shows two feature models resulting from the analysis of existing compliant robot control software and motion primitive based systems. Section B.2 shows an exemplary completion of the survey for one of the surveyed motion primitive approaches, the Dynamical Movement Primitive (DMP).

B.1. Feature Models

Feature models are representations of a group of systems in terms of their features [Kang et al., 1990], usually represented by means of feature diagrams. A *feature* in this context is defined as a *"prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system"* [Kang et al., 1990]. Section B.1.1 and Section B.1.2 show two of the feature models resulting from the domain analysis, describing aspects of compliant robot control and motion primitive based systems respectively in terms of their mandatory, optional, and alternative features.

B.1.1. Actuation and Controller Boards

Fig. B.1 shows feature models of actuators and controller boards, primarily based on the analysis of control libraries and interfaces of compliant robots, e.g., of the iCub humanoid robot (iCub), of the Oncilla discussed in Section 7.3.1, and of the KUKA LBR IV discussed in Section 7.3.2.

Primary feature of actuators and controller boards respectively the software libraries, firmware, and frameworks to control them, is to allow control of the respective actuator. Active compliant robots, such as a compliant version of the iCub and the KUKA LBR IV, provide a set of control features ranging from position to force control. The different control modes, if available, allow control of the respective feature, and often additionally allow sensing of the respective feature, e.g., through encoders in case of joint position control. Several of the surveyed libraries and boards additionally provide calibration routines, usually for system initialization. Controller boards and controller libraries often additionally allow configuration, e.g., of the controller gains or technical aspects like communication settings and communication rates.

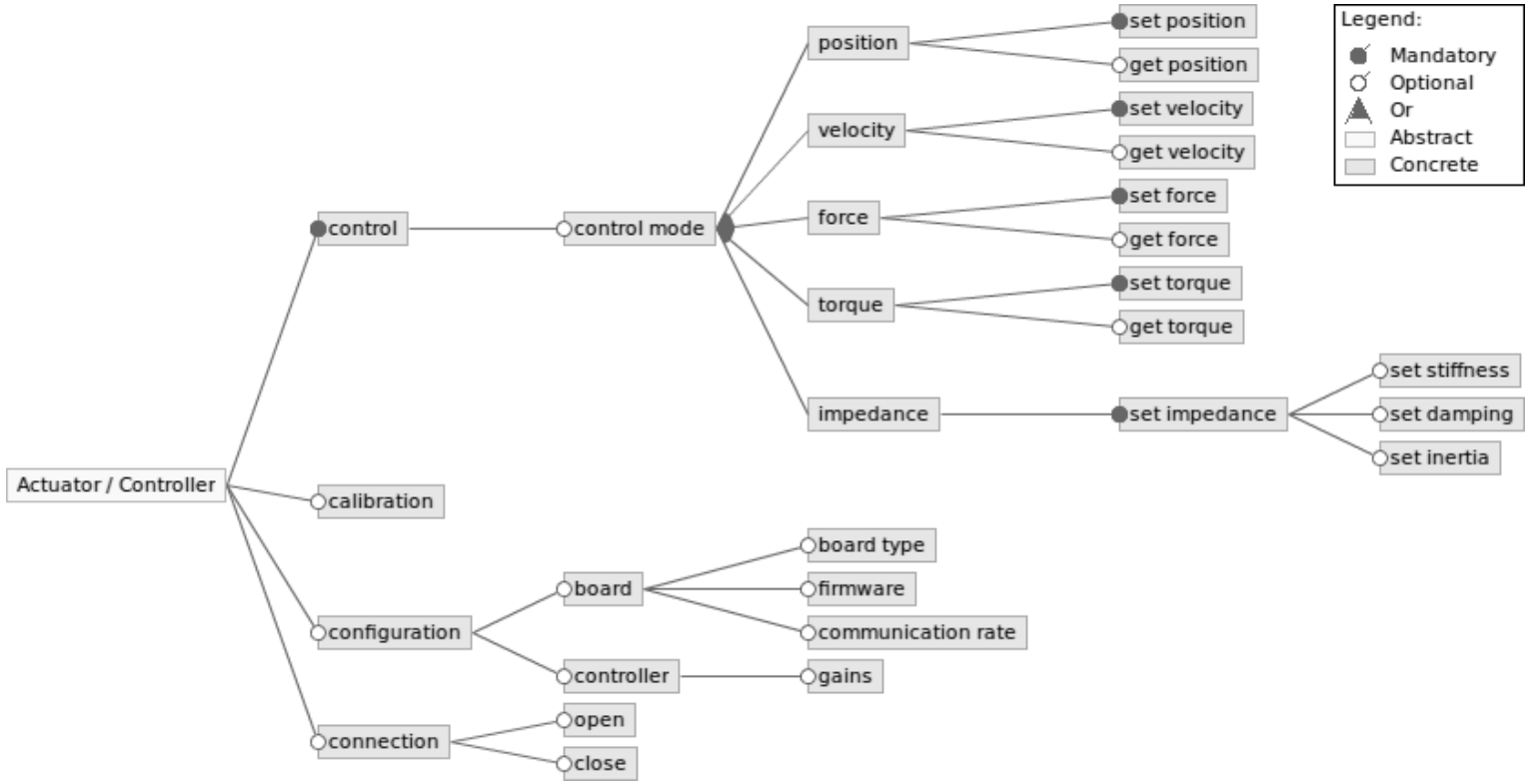


Figure B.1.: Feature model of actuators and controller boards, primarily based on the analysis of control libraries and interfaces of compliant robots, e.g., the iCub, as well as the Oncilla discussed in Section 7.3.1 and the KUKA LBR IV discussed in Section 7.3.2.

B.1.2. Motion Primitives

Fig. B.2 shows features of motion primitives as found in the domain analysis (cf. Section 3.3) and examined with the survey (cf. Section B.2).

A mandatory feature of a motion primitive is the contained Dynamical System, which can be an ordinary differential equation (ODE) or a complex Dynamical System such as an artificial neural network (ANN). As an optional feature, motion primitives with adaption capabilities comprise additional machine learning mechanisms that adapt the Dynamical System, e.g., adapting the weights of the ANN.

The surveyed motion primitives provide a range of dedicated inputs and outputs, e.g., to receive the goal of the motion, feedback from the robot, to send controller output, or report on their motion status, cf. Section B.2. The control output of the motion primitive is mandatory, as this creates the actual motion. The motion can often be externally configured, e.g., in terms of the speed or shape of the motion.

Timing of the surveyed motion primitive, i.e. how often they calculate a new motion setpoint, is either based on a fixed clock signal, based on the timing of incoming inputs, or based on a combination of these.

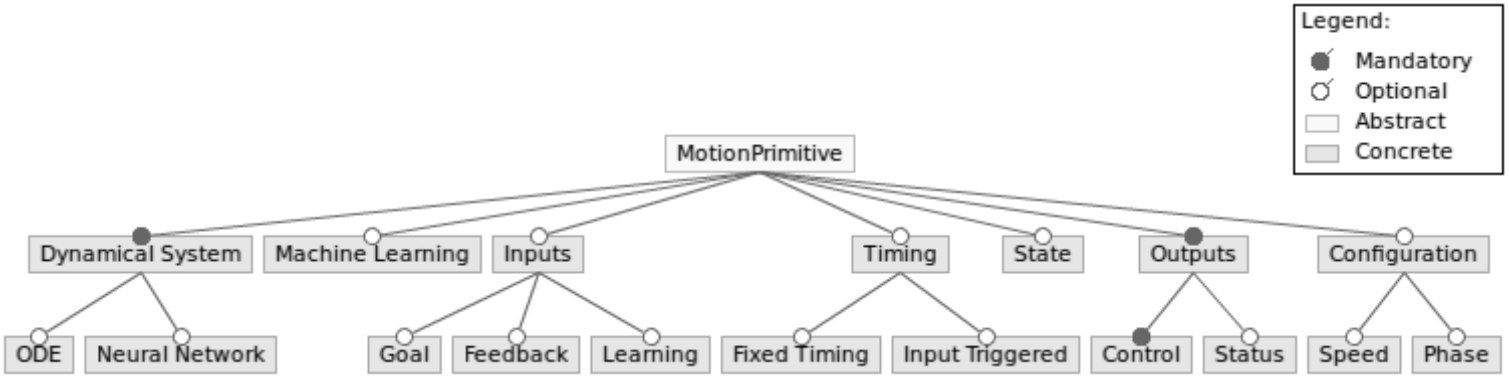


Figure B.2.: Feature model of motion primitives based on the domain analysis (cf. Section 3.3) and examined with the survey (cf. Section B.2).

B.2. Adaptive Module Survey

The Adaptive Module survey comprised meta information (rows 1 – 2 in Table B.1), functional properties (rows 3 – 17), and non-functional properties (rows 18 – 23). The survey was completed for twelve different Adaptive Module approaches by partners of the European AMARSi project. Table B.1 shows an exemplary completion of the Adaptive Modules survey for DMPs, cf. Ijspeert et al. [2013].

Name	**Dynamical Movement Primitive**
Group / Author	EPFL – BIOROB
Paper(s) / Reference(s)	[Ijspeert et al., 2013]
Dynamics	Non-linear convergent
Nonlinearity	Parametrized additive nonlinearity
Representation	Trajectory (position and/or velocity and/or acceleration)
Attractor	Point attractor or limit cycle attractor (world / joint coordinate)
Coupling	One system per DoF, coupled by the canonical system. If there are multiple canonical systems, canonical systems are phase-synchronized.
Generalization	Change point attractor
Movement generation	Transient to attractor
Learning algorithm	LWPR; mixture of local functions
Supervised / unsupervised	Supervised; reinforcement learning
Sensory feedback integration	Yes; affecting phase and amplitude
Number of state variables	4
Online vs. offline	Online or offline
Robustness to perturbations	Yes
Adaptation to perturbations	On-the-fly
Local vs. Global Stability	Global
Representation and Interface	Input: current position/velocity/acceleration, (internally: current states, 2-3 values), Output: first derivative of states including the current value in trajectory space (pos/vel/acc). Trajectories are modeled as linear combination of phase-driven basis functions.
Timing	The phase variable of the canonical system works as the internal clock for coordination. As the computations are light, small Δt values can be considered.
Robustness and Reliability	Anytime guarantee, satisfies reasonable real-time constraints, more basis functions to model more curvatures, and less ones to have a smoother output (when having lots of noise).
Dependencies	Usually depends on proprioception, but can also work open-loop, when controller (position or velocity or acceleration) is good enough.
Runtime / States	Offline Learning, online Learning, evaluation, recognition (with the help of an external tool). Learning is one-shot or online, and execution is simple and lightweight.
Software Availability	MATLAB or C

Table B.1.: Exemplary completion of the Adaptive Modules survey for the Dynamical Movement Primitive.

Glossary

AMARSi

Large scale European integration project in the Seventh Framework Programme, see: https://www.amarsi-project.eu/.

it's OWL

German national leading edge cluster "Intelligent Technical Systems OstWestfalenLippe".

5Cs

Communication, Computation, Configuration, Coordination, and Composition.

AADL

Architecture Analysis and Design Language.

ADL

architecture description language.

ADSL

architecture DSL.

ANN

artificial neural network.

API

application-programming interface.

AST

abstract syntax tree.

BCM

BRICS Component Model.

CBSE

component-based software engineering.

CCA

Compliant Component Architecture.

Central Pattern Generator

Neural networks (biological or artificial) capable of producing rhythmic patterns.

CI

continuous integration.

CIM

computation independent model.

CMake

A cross-platform free and open-source software for managing the build process of software.

Computation Independent Model

A computation-independent model is the business or domain model and uses the vocabulary that is familiar to the domain experts. It presents exactly what the system is expected to do, but hides all information technology-related specifications to remain independent of the implementation..

CPC

Component-Port-Connector.

CPG

central pattern generator.

dataflow

A dataflow network is a network of concurrently executing processes that communicate by sending data over so-called channels..

DMP

Dynamical Movement Primitive.

DoF

degrees of freedom.

Domain-Specific Language

A programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain, as opposed to General-Purpose Languages..

DSL

domain-specific language.

DSML

domain-specific modeling language.

Eclipse Modeling Framework

An Eclipse-based modeling framework and code generation facility for building tools and other applications based on a structured data model.

Eclipse Modeling Project

Open-source project focusing model-based development technologies within the Eclipse framework, providing a set of modeling frameworks, tools, and implementations..

Ecore

The core (meta-)metamodel of the Eclipse Modeling Framework (EMF), allowing to express metamodels.

ELM

Extreme Learning Machine.

EMF

Eclipse Modeling Framework.

EMP

Eclipse Modeling Project.

EuRoC

European Robotics Challenges.

Feature-Oriented Domain Analysis

The feature-oriented domain analysis [Kang et al., 1990] is a formal domain analysis method developed in 1990 that introduced features models and feature modeling..

FIFO

first in, first out.

FODA

feature-oriented domain analysis.

General-Purpose Language

A programming language designed to be used in a wide variety of application domains, as apposed to Domain-Specific Languages..

GPL

general-purpose language.

GPML

general purpose modeling language.

GraphML

GraphML is an eXtensible Markup Language (XML) based file format for graphs..

iCub

iCub humanoid robot.

IDE

integrated development environment.

Integrated Development Environment

Programming environments normally consisting of a source code editor, code completion, build automation tools, and debuggers.

iTaSC

instantaneous Task Specification using Constraints.

JEXL

Java Expression Language.

Kahn Process Network

Distributed model of computation where a group of deterministic sequential processes are communicating through unbounded first in, first out (FIFO) channels..

KUKA LBR IV

KUKA Lightweight Robot IV.

LMEC

language modularization, extension, and composition.

LTL

Linear Temporal Logic.

LWPR

Locally Weighted Projection Regression.

M2M

model-to-model transformation.

M2T

model-to-text transformation.

MARTE

Modeling and Analysis of Real Time and Embedded systems.

MATLAB

MATLAB (matrix laboratory) from MathWorks is a multi-paradigm numerical computing environment and programming language, especially for matrix manipulations, plotting of functions and data, and implementation of algorithms..

MDA

Model Driven Architecture.

MDE

model-driven engineering.

MP

motion primitive.

MPS

Jetbrains Meta-Programming System.

ODE

ordinary differential equation.

OMG

Object Management Group.

Oncilla

Oncilla quadruped robot.

OOP

object-oriented programming.

Ordinary Differential Equation

Ordinary differential equations are differential equations with only one independent variable. ODEs that are linear differential equations have exact closed-form solutions, which makes them attractive for robot control..

PbD

programming-by-demonstration.

pHRI

physical human-robot interaction.

PIM

platform independent model.

Platform Independent Model

A platform-independent model provides formal specifications of the structure and function of the system that abstracts away technical details..

Platform Specific Model

A platform-specific model combines the specifications in the platform independent model (PIM) with the details on how a system uses a particular type of platform..

PSM

platform specific model.

QoS

quality of service.

RCI

Robot Control Interface.

rFSM

reduced finite-state machine.

ROS

Robot Operating System.

RPC

remote procedure call.

RSB

Robotics Service Bus.

RST

Robotics System Commons.

RST

Robotics System Types.

SCXML

State Chart XML.

SLOC

source lines of code.

Synchronous Data Flow

Restriction of Kahn Process Networks, where nodes produce and consume a fixed number of data items per firing..

SysML

Systems Modeling Language.

ToF

time of flight.

UML

Unified Modeling Language.

W3C

World Wide Web Consortium.

XMI

XML Metadata Interchange.

XML

eXtensible Markup Language.

Xtext

Eclipse EMF Xtext.

YARP

Yet Another Robot Platform.