ARENBERG DOCTORAL SCHOOL
Faculty of Engineering

Domain Specific Languages for Hard Real-Time Safe Coordination of Robot and Machine Tool Systems

Markus Klotzbücher

Dissertation presented in partial fulfillment of the requirements for the the degree of Doctor in Engineering

April 2013

Domain Specific Languages for Hard Real-Time Safe Coordination of Robot and Machine Tool Systems

Markus KLOTZBÜCHER

Jury:
Prof. dr. ir. P. Verbaeten, chair
Prof. dr. ir. H. Bruyninckx, supervisor
Prof. dr. ir. Y. Berbers
dr. S. Michiels
dr. ir. M. Engels
(FMTC)
dr. ir. F. Ingrand
(LAAS/CNRS Toulouse)
dr. ir. M. J. G. van de Molengraft
(Technische Universiteit Eindhoven)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

April 2013

© KU Leuven – Faculty of Engineering
Celestijnenlaan 300B box 2420, B-3001 Heverlee (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

D/2013/7515/31
ISBN 978-94-6018-645-5

Preface

In 2007 I was becoming increasingly uneasy. This was, strangely enough, due to my professional life going decidedly well. My job as a freelance consultant was running smoothly, I was regularly teaching industrial seminars and had in general reached a level of confidence necessary to make consulting fun. My discomfort resulted from my long standing wish to pursue a PhD, to acquire deeper knowledge of Robotics and Automation and to have the chance to spend a significant amount of time focusing on *one* topic. I knew that if I didn't start soon, it was not going to happen anymore.

Moreover, the number of possibilities was further reduced by the self-imposed constraint of wanting to work with Free and Open Source Software. All the happier I was when Carsten Emde from OSADL pointed me in the direction of Herman Bruyninckx of the robotics group at KU Leuven. After exchanging a couple of emails and a first visit to Leuven, I was soon convinced to have found the perfect place: a group of friendly and knowledgeable people, deep involvement in Open Source through the OROCOS project, industrial robots and a Professor who replied to email inline and in plain text...

This work would not have been possible without the help of many. Firstly, I want to thank Herman for accepting me without hesitation as a latecomer and for patiently helping me getting into the domain of robotics and control. I very much appreciated your motivating, yet critical coaching style and your efforts to literally almost always be available to discuss ideas, while at the same time offering the freedom to work independently. I've also grown fond of eating ice cream for supper.

I would like to thank the members of my assessor committee and jury Yolande Berbers, René van de Molengraft and Marc Engels for continuously providing valuable feedback to my research. Together with the other members of my jury, Félix Ingrand and Sam Michiels, your comments have much improved my thesis. Thank you Prof. Verbaeten for chairing my defence.

I also much appreciate the help of Joris de Schutter, who, though less directly involved in my research, has repeatedly provided valuable suggestions and feedback.

I am very grateful to Carsten Emde for pointing me in the right direction at the right time.

I want to thank my former colleagues Wolfgang Denk and Detlev Zundel, who supported my endeavor by helping me during the first hybrid working–pre-doctoral year. Thank you Detlev for getting me into functional programming, it has profoundly influenced the way I develop software.

I would also like to acknowledge the members of the BRICS project, in particular those part of the Component Model Task Force, for the countless fierce but constructive discussions, which have certainly influenced this work.

Thank you European taxpayers for funding my research through the BRICS and Rosetta European FP7 research projects.

I would also like to acknowledge the Orocos community for the numerous, long mailing list discussions, for testing and adopting my software and for providing feedback that has helped to strengthen this work.

Thank you Tinne for having helped with almost everything which was important to me throughout the last years: starting from questions on kinematics and control theory over PhD school organizational issues to the topic of “kids and babies in Belgium”. I also very much appreciate you translating my abstract to Dutch on very short notice!

Thank you Peter Soetens for the friendly welcome in Leuven, for generously sharing your experience as a former PhD student and for supporting my plans to overhaul RTT’s scripting and state machines. Thank you Klaas for providing industrial relevant use-cases and Ruben for having helped a lot to improve my understanding of motion control and of course for the enjoyable hours of fighting with early versions of the YouBot. Thank you Wilm for helping with questions on control and of course for our enjoyable road-trip to Yosemite National Park together with Nick.

I would also like to thank my fellow PhD students: Enrico for taking care that the caffeine flow is not interrupted, Hans and Koen for helping me to get Ernie and Bert back to life, Steven for the help with Observers, Nick, Niccoló and Enea for early testing and adopting my software, Lin for showing and explaining his newest gadgets, Azamat for the interesting discussions on functional composition and Bart for proofreading my Dutch abstract.

I am very grateful to Omi Brigitta for all her help and for coming to Leuven countless times to help out when I was traveling.

I want to sincerely thank my parents Kurt and Angela Klotzbücher for their tireless support and encouragement, and the many visits to help with the kids. Thank you Dad for inspiring me to become an engineer.

Dorothee, thank you so much for supporting me from the start, for having the courage to start a family abroad and for taking such good care of our wonderful kids. Thank you Yara and Lukas for reminding your Dad each day of what really matters.

Markus Klotzbücher,
Leuven, March 2013.

Abstract

The software controlling modern robotic and machine tool systems is becoming increasingly complex. This has several reasons: distributed system architectures involving autonomous, loosely connected subsystems are becoming more and more prevalent and are replacing simpler, centralized ones. At the same time, the need to keep software development costs low mandates reuse and integration of existing subsystems, which further contributes to complexity due to the need for additional harmonization and adaptation layers. The increasing universality of modern lightweight robots and mobile platforms is also reflected on the software, which must cope with stronger demands in terms of variability, composability and reconfigurability.

One approach to deal with this complexity is the Model Based Engineering (MBE) paradigm, which has already been successfully applied to domains such as automotive, aerospace and control engineering. In this approach the focus is shifted from traditional programming to capturing the required information in models. These models can subsequently be transformed to executable form, but without having to assume beforehand how this will be done. Moreover, these models can remain available at runtime to permit online adaptation by the robot itself. This thesis explores the applicability of MBE to the domain of distributed, real-time robotic and machine tools systems with a focus on the coordination of the discrete behavior of such systems.

Coordination is a system level concern that governs how and when functional subsystems interact. By explicitly modeling coordination, the desired behavior of a system is formalized while permitting functional computations to remain free of application logic and hence more reusable. More concretely, an approach of modeling using domain specific languages (DSL) is chosen. In this approach, dedicated and composable languages are constructed to model a well confined domain. In contrast to rich, general purpose modeling languages, this approach yields minimal models that capture the essence of the problem while avoiding the overhead of generality.

The main contribution of this thesis is the development of a composable DSL named rFSM for modeling coordination in distributed and real-time constrained robotics and machine tool systems, and an associated approach of applying this model. The advantages of providing a minimal, but extensible and composable model are demonstrated. Further contributions support, make use of or extend this contribution. Supporting work demonstrates how DSL models can be instantaneously executed in hard real-time, and introduces the uMF DSL for modeling structural constraints on DSL models themselves. The applicability and potential for improving reuse is shown in experiments using a hybrid force-velocity task specification language together within an rFSM model to define an assembly task. The work on the Coordination–Configurator pattern formalizes a frequently recurring architectural pattern that allows to increase the performance and reusability of coordination.

Beknopte samenvatting

De controlesoftware voor hedendaagse robotica- en machinesystemen wordt steeds complexer. Dit heeft verschillende redenen: gedistribueerde systeemarchitecturen met autonome, losjes met elkaar verbonden subsystemen komen steeds meer voor en vervangen eenvoudigere en gecentraliseerde systemen. Om de softwareontwikkelingskosten laag te houden is er tegelijkertijd nood aan het hergebruiken en het integreren van bestaande subsystemen. Dit op zijn beurt draagt bij tot een verhoging van de complexiteit door de nood aan extra harmonisatie- en aanpassingslagen. De software weerspiegelt eveneens de toegenomen universaliteit van moderne lichtgewicht robots en mobiele platformen die moeten omgaan met de toegenomen vraag naar variabiliteit, samenstelbaarheid en herconfigureerbaarheid.

Eén manier om met deze complexiteit om te gaan is gebaseerd op het Model-Based Engineering (MBE) paradigma dat al succesvol is toegepast in domeinen zoals de automobieltechnologie, luchtvaart encontroleontwerp. Deze benadering verplaatst de aandacht van het traditioneel programmeren naar het vastleggen van de vereiste informatie in modellen. Deze modellen kunnen vervolgens omgezet worden naar een uitvoerbare vorm, zonder op voorhand vast te leggen hoe dit zal gebeuren. Ze blijven bovendien beschikbaar tijdens de uitvoering om het gedrag van de robot in reële tijd aan te passen. Dit proefschrift verkent de toepasbaarheid van MBE en in het bijzonder de coördinatie van het discrete systeemgedrag, op het domein van gedistribueerde reële tijd robotica- en machinesystemen.

Coördinatie, een bezorgdheid op systeemniveau, regelt hoe en wanneer functionele subsystemen met elkaar interageren. Het expliciet modelleren van de coördinatie leidt tot het formaliseren van het gewenste systeemgedrag terwijl de functionele berekeningen toch vrij blijven van toepassingsspecifieke logica. Dit zorgt voor een verhoogde herbruikbaarheid. Dit proefschrift gebruikt hiertoe domeinspecifieke talen (domain specific languages, DSL). In deze aanpak worden domeinspecifieke en samenstelbare talen opgesteld die een goed afgelijnd

domein modelleren. Dit leidt tot minimale modellen die, in tegenstelling tot programmeertalen voor algemeen gebruik, de essentie van het probleem bevatten en tegelijkertijd de kosten ten gevolge van te grote veralgemening vermijden.

De belangrijkste bijdrage van dit proefschrift is de ontwikkeling van enerzijds een samenstelbare DSL, genaamd rFSM, voor het modelleren van coördinatie in gedistribueerde robotica- en machinesystemen en anderzijds een bijhorende aanpak om dit model toe te passen. Het proefschrift demonstreert de voordelen van een minimaal maar uitbreidbaar en samenstelbaar model. De andere bijdrages ondersteunen, gebruiken of breiden deze kernbijdrage uit. Ondersteunend werk toont de uitvoering van DSL modellen in reële tijd en introduceert een uMF-DSL voor het modelleren van structurele beperkingen op DSL-modellen zelf. Assemblagetaken gebruik makend van een taal voor hybride-kracht-snelheidscontrole in combinatie met een rFSM model tonen experimenteel de toepasbaarheid en het potentieel voor toegenomen hergebruik. Het werk over het Coördinatie-Configuratiepatroon formaliseert een vaak terugkerend architectuurpatroon en verhoogt de performantie en het hergebruik van de coördinatie.

Abbreviations

3D,2D,1D	Three-,two-,one- dimensional
AADL	Architecture Analysis and Design Language
AOP	Aspect Oriented Programming
API	Application Programming Interface
CIM	Computation Independent Model
DFA	Discrete Finite Automaton
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
FMTC	Flanders' Mechatronics Technology Centre
FRI	Fast Research Interface (KUKA)
FSM	Finite State Machine
fUML	Foundational UML (OMG standard)
iTaSC	Instantaneous Task Specification using Constraints
LOC	Lines of Code
LWR	Light Weight Robot (e.g. Kuka LWR)
MARTE	The UML profile for Modeling and Analysis of Real-Time and Embedded Systems
MBE	Model Based Engineering
MDA	Model Driven Architecture (OMG standard)
MDE	Model Driven Engineering
MiB	Mebibyte= 1024^2 Byte (IEC 60027)

MOF Meta Object Facility (OMG standard)

OCL OMG Object Constraint Language
OMG Object Management Group
OOP Object Oriented Programming
OROCOS Open Robot Control Software
OSADL Open Source Automation Development Lab

PID Proportional-integral-derivative Controller
PIM Platform Independent Model
PLC Programmable Logic Controller
PSM Platform Specific Model

QoS Quality of Service
QVT Query View Transformation (OMG standard)

rFSM reduced Finite State Machine
ROS Robot Operating System
RTES Real Time and Embedded Systems
RTT Real Time Toolkit

SAE SAE International (formerly Society of Automotive Engineers)
SFC Sequential Function Charts
SysML OMG Systems Modeling Language

TFF Task Frame Formalism

uMF Micro Modeling Framework
UML Unified Modeling Language

XML Extensible Markup Language
XSD XML Schema Definition
xUML Executable UML

Contents

Abstract v

Contents xi

List of Figures xvii

List of Tables xxi

1 Introduction 1

1.1 Requirements 3

1.2 Approach 5

1.3 Research Objectives 5

1.4 Outline 7

2 Background and Positioning 11

2.1 Model Based Engineering 11

2.2 Domain Specific Languages 13

2.3 Separation of Concerns 14

2.4 Real-Time Systems 15

2.5 Behavioral Modeling Languages and Formalisms 15

2.6 Robot Software Architectures 19

2.7 Coordination Languages 20
2.8 Separating the four C's 21
2.9 Conclusion of Literature Survey 22

3 Hard real-time Control and Coordination using Lua 23

3.1 Abstract 23
3.2 Introduction 24
3.3 Related work 25
3.4 Approach 25
3.5 Experiments 26
3.5.1 Lua Cyclictest 27
3.5.2 Event messages round trip 28
3.5.3 Cartesian Position Tracker 30
3.5.4 Coordination Statechart 31
3.6 Conclusions 33

4 Coordinating Robotic Tasks and Systems using rFSM Statecharts 35

4.1 Abstract 35
4.2 Introduction 36
4.2.1 Motivating example 37
4.2.2 Contributions 39
4.2.3 Related Work 39
4.2.4 Outline 40
4.3 Review of Coordination Models 41
4.3.1 Classical Finite State Automatons 41
4.3.2 Harel Statecharts 41
4.3.3 OMG UML State Machines 42
4.3.4 Statecharts in Robotics 42

4.3.5 IEC 61131-3 Sequential Function Charts 43
4.3.6 Behavior Trees 44
4.3.7 The Task Description Language 44
4.3.8 The Urbiscript Language 45
4.3.9 Simulink Stateflow 45
4.3.10 Statecharts in Modelica 46
4.3.11 ROS SMACH 46
4.3.12 Conclusion of Literature Review 46
4.4 The rFSM Model 48
4.5 Structural Semantics 52
4.5.1 Fundamental State Machine Elements 52
4.5.2 Hierarchical State Machines 55
4.5.3 UML Pseudo-States 58
4.5.4 State Machine Extension 63
4.6 Execution Semantics 63
4.6.1 Fundamental Execution Semantics 64
4.6.2 Event Selection 65
4.6.3 Computing the Enabled Transition Set 65
4.6.4 Discussion 66
4.6.5 Evaluating Composite Transitions 67
4.6.6 Transition Execution 68
4.6.7 rFSM Transition Semantics 68
4.7 Event Semantics 70
4.7.1 UML ChangeEvent 70
4.7.2 UML Time Event 70
4.7.3 UML Call Event 71
4.7.4 UML Completion Event and Final State 71

4.7.5 UML AnyReceiveEvent and unlabeled Transitions . . . 72
4.7.6 Edge- and Level-triggered Events 72
4.7.7 Deferred Events 73
4.8 Concurrency Semantics 74
4.9 Reference Implementation 75
4.9.1 Software Framework Integration 77
4.9.2 Considerations for Hard Real-Time Execution 78
4.9.3 Representing Events 78
4.10 Patterns and Best Practices 78
4.10.1 Models of State Machine Progression 79
4.10.2 Defining Platform and Robot independent Coordination Models 80
4.10.3 Best practice *Pure Coordination* 82
4.10.4 Event Memory 83
4.10.5 Distributed Substates 84
4.10.6 Serialised Locally Distributed States 86
4.10.7 Discrete Preview Coordination 86
4.11 Example: constructing coordination for a dual-robot haptic coupling 89
4.12 Discussion 95
4.13 Conclusion and Future Work 96

5 Reusable motion specifications with executable DSL 97

5.1 Abstract 97
5.2 Introduction 98
5.2.1 Related work 100
5.3 Domain Specific Languages for M1, M2, M3 101
5.3.1 M1- and M2-level TFF-DSLs 102

5.3.2 M3 model: Ecore 103
5.3.3 Software Framework integration: from M1 to M0 . . . 104
5.3.4 Composing individual TFF-DSL motions into skills using the rFSM statechart DSL 105
5.3.5 Dealing with robot dependencies 107
5.4 Experiments on PR2 and KUKA LWR 108
5.5 Conclusions 110

6 Specifying and Validating Internal DSL 111

6.1 Abstract 111
6.2 Introduction 112
6.2.1 Internal DSL in Lua 113
6.2.2 Related work 114
6.3 The uMF micro-modeling framework 115
6.3.1 uMF Types 115
6.3.2 uMF Specs 115
6.4 Real World DSL 117
6.4.1 The Task Frame Formalism DSL 117
6.5 Evaluation and Results 120
6.5.1 The limits of untyped conforms-to 121
6.6 Conclusion and Outlook 121

7 Pure Coordination using the Coordinator–Configurator Pattern 123

7.1 Abstract 123
7.2 Introduction 124
7.2.1 Prior usage 125
7.2.2 Outline 126
7.3 Approach 126

7.4 Example 126
7.5 Modeling configuration and its application 129
7.6 Discussion 131
7.6.1 Deployment 132
7.6.2 Composition 132
7.7 Conclusions 133

8 Conclusions 135
8.1 Contributions 135
8.2 Discussion 138
8.3 Impact 143
8.4 Suggestions for Future Work 144

A Tools 147
A.1 rFSM online visualization 147
A.2 rFSM commander 148

Bibliography 151

Curriculum 165

Publications 167

List of Figures

1.1 The KUKA lightweight robot (left) and the Universal Robots UR5 (right). . . . 2

1.2 Chapters ordered according to the OMG MOF metamodeling layers of metametamodel (M3), metamodel (M2) and model (M1). Higher levels can be understood as mechanisms used to realize lower levels, while lower levels *conform to* higher levels. . . . 8

3.1 Sequence diagram of event round trip test. . . . 29

3.2 Coordinating the grasping of an object. . . . 31

3.3 Dealing with low memory. . . . 32

4.1 Data-flow architecture of Ball tracking application. . . . 37

4.2 Ball-tracking coordination state machine. . . . 38

4.3 Complete rFSM Ecore model. . . . 48

4.4 Coordinating a gripper. . . . 52

4.5 Hierarchical state machine. . . . 56

4.6 Boundary crossing. . . . 57

4.7 UML Junction. . . . 59

4.8 UML Junction with split. . . . 59

4.9 UML History states. . . . 62

4.10 Scope of transitions. . . . 64

4.11 Flowchart of the step procedure. 69

4.12 Dealing with inter-step history. 83

4.13 Distributed state. 84

4.14 Distributed substate. 84

4.15 Object retrieval without Preview Coordination. 87

4.16 Object retrieval with Preview Coordination. 88

4.17 The youbot coupling demo at the Automatica trade fair. The forces of pulling on one robot arm are felt on the other side. Once the forces rise above a threshold, the coupling is disabled and both robot arms are put into floating mode. 89

4.18 Computational component architecture of the dual youbot haptic coupling. 90

4.19 Component architecture with extensions to support coordination. 92

4.20 Constructing the coordinator, step 1: modeling the communication quality constraint. 92

4.21 Constructing the coordinator, step 2: modeling the force threshold constraint. 93

4.22 Constructing the coordinator, step 3: adding modes 94

5.1 The four levels in OMG's standard for the role of Domain Specific Languages in Model Driven Engineering. 100

5.2 The alignment task executed by the KUKA LWR, with an implementation based on Orocos/RTT. 101

5.3 The alignment task executed by the PR2, with an implementation based on Orocos/RTT and ROS. 102

5.4 The Ecore metamodel (M2 level in Figure. 5.1) that represents the formal model that all DSLs for hybrid force/velocity control should conform-to. 104

5.5 rFSM Statechart for the alignment skill. 105

5.6 Harmonizing the KUKA FRI operational modes. 107

5.7 Forces, velocities and positions of the PR2 robot. 109

7.1 Bidirectional youBot coupling demonstration: each robot copies the cartesian position of its peer robot. 125

7.2 The relationship between the Coordinator, the Configurator and Computational components. 127

7.3 rFSM Coordination Statechart for the youBot coupling demo. . 128

7.4 Component diagram illustrating the components and connections running on each of the youBots in the example. 129

A.1 rFSM online visualization tool. 148

A.2 rFSM online interaction tool. 149

List of Tables

3.1 Cyclictest results (μs) 28
3.2 Comparison of impact of garbage collector modes 28
3.3 Results of event round trip test. 29
3.4 Results of Cartesian Position Tracker Benchmark 31

5.1 Platform-independent code 109
5.2 Robot and Framework specific code. 110

Chapter 1

Introduction

The domain of robotics is concerned with construction and control of mechanical systems that interact with their environment to carry out tasks. Today, robots are being applied to an ever-growing number of application domains, including medical systems such as surgical robots, applications in factories and warehouses to assist workers or service robots in domestic environments for supporting humans with everyday tasks, just to name some.

As a result of this growing diversity of applications, there is a trend towards developing robot systems that are increasingly *universally applicable*. Figure 1.1 shows two examples of such robots, the KUKA lightweight robot (LWR) and the Universal Robots UR5 robot. This generality implies the need to carry out a broader range of tasks, to support higher degrees of task variability and the possibility to assign multiple robots to a task when one is insufficient by itself. At the same time, the need to keep development costs low mandates reusing as many existing software elements as possible.

A second major trend is the deployment of robots in environments that are *shared with humans* or even in which *robots and humans engage in physical interaction*. This merging of workspaces poses many challenges; most importantly the robot system must be safe for humans. On the other hand, also the robot must be capable of protecting itself from damage. Such measures are necessary for intelligent factories, in which humans and robots work together but even more so for personal service robots supporting people in home environments. This safety can be achieved in different ways: as of today the majority of personal service robots (as for instance vacuum cleaning robots) are designed to be inherently safe by lacking the force to harm humans. However, for more generic and useful robots this will not be possible, hence the ultimate responsibility

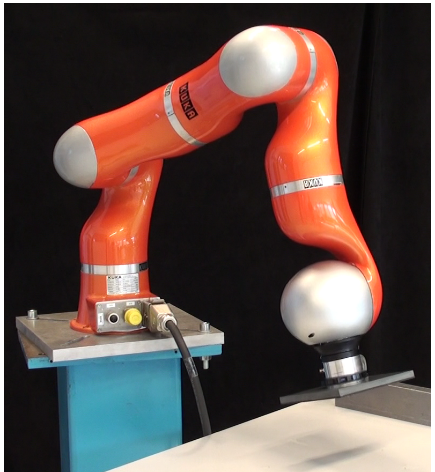 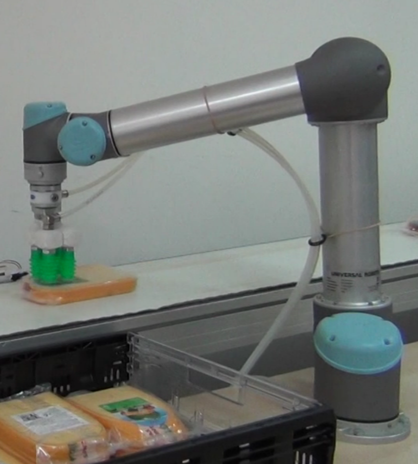

Figure 1.1: The KUKA lightweight robot (left) and the Universal Robots UR5 robot[1] (right).

of safe operation is placed on the robot control software. The IEC standard 61508 (International Electrotechnical Commission 2010) provides requirements and guidelines to realize functional safety, which go beyond the scope of this work. Here the focus is on software reliability as one prerequisite for safety. The ISO/IEC/IEEE standard 24765:2010 defines software reliability as *"the probability that software will not cause the failure of a system for a specified time under specified conditions" Systems and Software Engineering – Vocabulary* (ISO/IEC/IEEE 2010).

Moreover, shared environments often introduce uncertainty. *As a consequence, robots must cope with unexpected conditions*, such as obstructed paths, vanishing and reappearing objects or even collisions. Operating reliably in such environments places high requirements on the robustness of robot control mechanisms. The ISO/IEC/IEEE standard 24765:2010 defines robustness as *"the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions"* (ISO/IEC/IEEE 2010).

These trends, and the requirements resulting therefrom lead to a *drastic increase in complexity of robot software systems.* Robot task related complexity arises from the need for reusable and thus robot and software platform independent tasks, which can be customized for different use-cases or composed as subtasks within larger applications. Building safe systems requires introducing additional

[1] Picture courtesy of Intermodalics BVBA.

safety mechanisms to self-monitor and respond in case of anomalies. Likewise, shared and uncertain environments further contribute to complexity, since fewer assumptions can be made about the state of the world, and thus more conservative self-validation becomes necessary to ensure reliability.

If the complexity arising from these sources is not dealt with, a system may exhibit undefined or invalid behavior or perform insufficiently. This thesis aims to contribute to the ongoing discussion on how to deal with this complexity by applying the Model Based Engineering (MBE) approach to the domain of Robotics. The MBE approach shifts the focus from writing programs to specifying models, that then can be transformed to executable form. This approach promises several advantages. Firstly, the necessary information is captured without having to assume beforehand *how* it will be used, thus contributing to reusability. Model checking facilitates early detection of defects, which in turn improves reliability and robustness. A further advantage is that the MBE paradigm is suited to bring structure to the development process by introducing tools and workflows to guide the evolution of models. The fundamental research question investigated in this work is:

> ***How can Model Based Engineering Techniques improve the Software of Industrial and Service Robotic Systems in terms of Robustness, Reliability, Safety and Reusability?***

1.1 Requirements

The goal of this thesis is to develop methodologies and tools based on the Model Based Engineering paradigm for supporting the development of the software of complex robot systems. This section formulates in more detail the requirements for these developments and defines the term *complex robot system* in the context of this work.

An important characteristic of this class of systems is that these do not only consist of one single computational node, but instead of a *number of distributed nodes*, that may be connected through unreliable communication such as Ethernet or wireless connections. Hence, it is a requirement that in the event of communication failures, these subsystems behave robust and maintain autonomous operation (even though this can be at a lower Quality of Service (QoS)).

A similar requirement exist in terms of *reliability*: in many cases it is acceptable that a run-time fault leads to a temporary degradation of the service provided by the system as a whole, in contrast fatal, unrecoverable errors should be

avoided at all costs. In this manner, this work shall contribute to safety of these systems by improving reliability; on the other hand, efforts such as Risk and Hazard Analysis (International Electrotechnical Commission 2010) are outside the scope of this thesis.

Moreover, complex robot systems often exhibit *real-time constraints*, such as maximum permitted worst case durations after which a controller switch must have completed, or maximum permissible response times to failure events. It is crucial that this requirement is taken into account from the start, since real-time safety is not a property that a system can easily be extended with later.

The software development for a complex robotic system almost never starts from scratch, since the costs of this approach would be too high. Instead, by *reusing existing application elements*, software development costs can be reduced. This requirement needs to be foreseen by any developed tooling or development methodology. Likewise, it is necessary to define how new software elements shall be developed to facilitate their reuse in different applications. A requirement related to reusability is the need for composable models. Composability may refer both to constructing complex models from simple models of the same type as well as combining heterogeneous models into a new composite model.

A further requirement is that models are both available and can be adapted at run-time. This is, on the one hand, to support high availability of robot systems by facilitating software updates to be applied to the running system. On the other hand this is to support models that can be adapted according to changes in robot task or environment.

Lastly, the approach shall permit and encourage specification of models that are minimal. A minimal model is a model containing only those primitives necessary to express the concepts of the respective domain. Composability plays an important role to achieve minimality by permitting i) to include existing models instead of redefining concepts and ii) to avoid new primitives by expressing these in terms of existing ones. This way, the minimality requirement serves to increase the potential of model reuse by avoiding specialized, rich models, and on the other hand to encourage simple models that are easy to understand.

Improving the performance (e.g. in terms of speed, accuracy, stability of controllers) of the systems developed is *not* a primary requirement. On the other hand, the devised approach shall not introduce any constraints that would lead to a significant degradation of performance compared to existing approaches.

1.2 Approach

The modeling approach taken in this thesis is based on the concept of *domain specific languages* (DSL). A DSL is a formal language that is tailored to express the concepts of a particular domain. In contrast to general purpose modeling languages like the Unified Modeling Language (UML), DSL incorporate the concepts and terminology of the respective domain and are developed for use by the human domain expert.

More specifically, this work makes extensive use of so called internal or embedded DSL. An internal DSL is constructed on top of a general purpose host language, whose infrastructure and run-time is reused. In contrast, an external DSL is developed from scratch, which generally requires more effort since it also involves the definition of syntax and associated parsers.

An important aspect is the host language employed to define the internal DSL. While in principle most programming languages can be used, this work opted for a scripting language. This choice has the following advantages. Models can be instantaneously executed without offline transformation and compilation. Verification and model transformations within the final execution environment are facilitated, while additional code-generation steps are avoided. Moreover, safety and robustness are improved, since the scripting environment acts as a virtualized *sandbox* that limits the impact of model execution failures.

For this work the Lua programming language was chosen as a host language for internal DSL, since it combines multiple desirable properties. These include the suitability for building DSL with a human readable syntax, the feasibility to be embedded within resource constrained environments and the suitability to be extended such as to permit execution satisfying real-time constraints. A disadvantage of this approach is that the possible DSL syntax is constrained by the Lua language. Furthermore, the challenge of executing the interpreted Lua DSL in hard real-time needs to be addressed.

1.3 Research Objectives

The research objectives investigated in this work fall into three categories. The first are related to addressing limitations hindering the application of internal DSL to the domain of real-time robotics and machine tools. The second are concerned with identifying and realizing robotics specific DSL. The third aim to implement applications using these DSL and to identify recurring patterns.

An overview showing these categories and the corresponding chapters is given in Figure 1.2.

The following introduces the objectives of this work in terms of research questions, which have been explored. The given order largely reflects the sequence in which the research was carried out, though Chapter 4 on rFSM Statecharts was subject to ongoing work throughout the entire thesis.

The chosen approach of using an interpreted scripting language as an DSL execution environment poses the question of how such dynamic interpretation can be combined with the need for temporal determinism. More concretely, the following research questions is investigated in Chapter 3:

> ***How can models be executed safe, embeddable, instantaneously and satisfying real-time constraints?***

A major area of focus is the coordination of complex robotic systems. Coordination is a system level concern dealing with supervising and monitoring of the functional parts of a system such that the system as a whole behaves as expected. The following research question is explored in Chapter 4:

> ***Which minimal and composable model is suitable to coordinate complex, distributed robot systems involving real-time constraints and legacy components?***

As a more specific case of Coordination, the topic of specifying robot tasks in a reusable way is investigated. Reusability can be challenging, since a robot task specification not only depends on the robot used, but also on the underlying software framework and the respective coordination mechanism employed. The following research question is investigated in Chapter 5:

> ***How can robot tasks be modeled in a robust, yet robot and software platform independent way?***

The approach of modeling with internal domain specific languages blends the activities of *modeling* with traditional *programming*. While this increases the flexibility and permits taking advantage of the host language, it also places a higher burden on the internal DSL developer who has to account for the lesser formalization by implementing more manual validation. This observation lead to the investigation of the following research question in Chapter 6:

> ***How can the structure of internal DSL be formalized and validated?***

After developing the coordination of several complex robot systems, recurring patters begin to emerge. Identifying such patterns is essential to support inexperienced developers to make optimal design choices. Furthermore, the formalization of recurring patterns facilitates development of *dedicated* tooling. Thus, the research question explored in both Chapter 4 and 7 is:

> ***What are patterns and best practices for developing the coordination of robot systems?***

1.4 Outline

Each of the core Chapters 3-7 is based on a peer reviewed conference or workshop paper, except Chapter 4 on rFSM Statecharts, which is based on a journal paper published in the JOSER journal.

Chapter 2 explores the literature relevant to the requirements and research objectives of this thesis. Since the core chapters are based on publications and include discussions of related work, this literature survey provides a broad overview of the state of the art without anticipating individual chapters.

Chapter 3 describes an approach for using the Lua programming language in systems with hard real-time constraints. This chapter lays the groundwork for successive chapters, that introduce multiple real-time safe Lua DSL.

Chapter 4 introduces the rFSM Statechart model, that was developed for the purpose of modeling coordination in complex robot systems. This chapter can be divided into four parts. Firstly, the rFSM coordination model is derived by means of an extensive discussion of state-of-art coordination models. Next, a real-time safe reference implementation of this model as an internal Lua DSL is described. To illustrate the use and extension of rFSM, several usage patterns and best practices are provided. Lastly, a step-by-step example explains the suggested methodology to derive the coordination for a given system.

Chapter 5 introduces a DSL to model motion specifications expressed in a hybrid force-velocity control formalism, and illustrates how such a task DSL can be composed within a rFSM Statechart. To demonstrate the reusability, an alignment task is specified and executed on a Willow Garage PR2 and a KUKA Lightweight Robot (LWR). The resulting application is analyzed with respect to its reusability among different software and robot platforms.

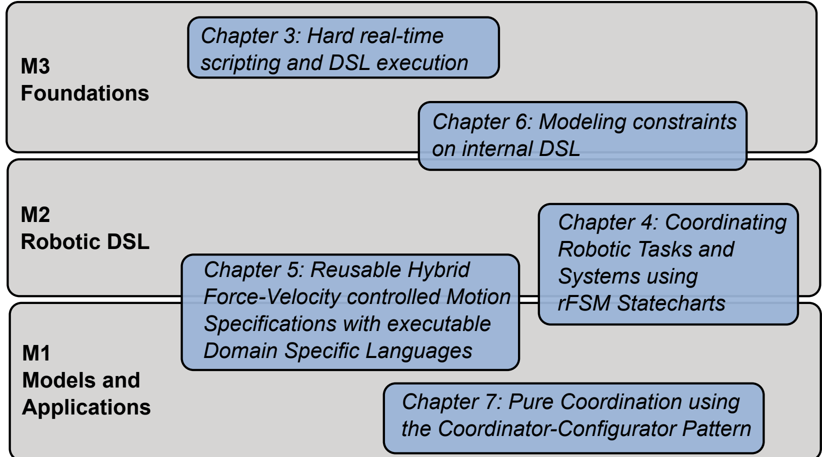

Figure 1.2: Chapters ordered according to the OMG MOF metamodeling layers of metametamodel (M3), metamodel (M2) and model (M1). Higher levels can be understood as mechanisms used to realize lower levels, while lower levels *conform to* higher levels.

One shortcoming of the internal DSL approach is the lack of strict formalization of metamodels. **Chapter 6** describes the uMF metameta modeling language that permits formalizing and validating structural constraints on embedded Lua DSL. Also, the notion of an *open model* is introduced, which is a model who's structure is only partially constrained.

Chapter 7 describes the Coordinator–Configurator architectural pattern, that enables increased reusability, robustness and deterministic timing behavior.

An alternative arrangement to the chronological ordering of the chapters is given by Figure 1.2, which groups the chapters according to the three modeling levels of metametamodel, metamodel and model as defined by the OMG MOF specification (Object Management Group 2006). Chapters 3 and 6 are concerned with establishing the metametamodels (M3), that are then used to construct robotic DSL (M2, metamodels), which are introduced in Chapters 4 and 5. However, the latter two chapters not only describe the languages themselves, but also provide case studies using concrete models (M1) that were developed using these DSL. Chapter 7 is likewise situated at M1, since it describes a blueprint of an application. Since all described DSLs are instantaneously executable, the implementation level (M0) is omitted.

The core chapters are based on publications and are as such self-contained. Therefore no strict reading order is necessary. Nevertheless, earlier chapters

introduce mechanisms used by later chapters. For instance, Chapter 6 introduces an approach to constrain domain specific languages which is used by a concrete DSL defined in Chapter 7, and Chapter 4 introduces rFSM Statecharts which are used in several subsequent chapters. Hence, following the suggested order will result in the best understandability.

Chapter 2

Background and Positioning

This section provides a broad survey of existing tools and methods related to the requirements formulated in section 1.1. Since each individual chapters includes a related work section, this chapter will not duplicate but refer to these where appropriate. This chapter concludes with a discussion of the limitations of the state of the art with respect to satisfying the requirements of this research.

2.1 Model Based Engineering

Model Driven Engineering (MDE) or Model Based Engineering (MBE) describe a software development approach that promises to improve the quality of developed software. The central idea is to define a modeling language (also called a meta-model) suitable to capture the aspects relevant to a particular domain. This language is then used to specify concrete models that can then be analyzed, validated, transformed or even executed. The latter activities are greatly facilitated by having a formalization of the meta-model available. The main benefit of this approach is the clean separation of the *domain knowledge* from technical implementation details.

MDE has been described and standardized by different entities. One of the most widespread efforts is the *Model Driven Architecture* (MDA) initiative (Miller and Mukerji 2003) by the Object Management Group. MDA describes several modeling levels: the computation independent model (CIM) is an informal, high level description. The Platform Independent Model (PIM) describes the software system independently of the Software Platform it will later run on; that

information is added during the transformation to the Platform Specific Model (PSM). The latter model can then be transformed to programming language code. A thorough introduction including a discussion of the promises of MDA can be found in Bézevin (Bézivin 2005).

MDA is built around different other OMG standards: the Meta-Object Facility (MOF) specification (Object Management Group 2006) describes the meta-meta modeling language and architecture that is used to specify meta-models. The Query View Transformation (QVT) (Object Management Group c) standard provides several languages to support describing model transformations. The Unified Modeling Language (UML) (Object Management Group 2011) provides a comprehensive list of standard diagram types for modeling different aspects of software (and is itself described using MOF). The Object Constraint Language (OCL) (Object Management Group a) is used to specify additional constraints on models that can not be expressed by UML alone.

A shortcoming of the OMG approach is the lack of rigid semantic formalization or reference implementation, which leads to tool vendors making mutually incompatible implementation choices. *Executable UML* (xUML) (Mellor and Balcer 2002) attempts to close this gap by formalizing the semantics of a subset of UML. The idea of xUML evolved out of the Shlaer-Mellor method (Shlaer and Mellor 1988) and has been adopted by OMG as foundational UML (fUML) (Object Management Group d). According to the fUML standard, the intention is “to encourage use of the broadest possible subset of UML constructs that can be reduced to a small set of elements” and to provide a “precise definition of the execution semantics of that subset.” (Object Management Group d, p. 1).

Within mainstream MDE, two fundamental modeling philosophies can be distinguished: profiling and meta-modeling. The UML based *profiling* approach encourages the extension or redefinition of existing UML meta-models for use in the context of a specific domain. To this end *Stereotypes* permit refining existing model elements, *Stereotype attributes* can introduce new named values. *Constraints*, typically expressed using OCL, permit to specify conditions that must be satisfied after the profile has been applied. This can be important when multiple profiles are applied simultaneously. Today, a multitude of profiles exists for different domains. Relevant in the context of this thesis are the following profiles: The System Modeling Language (SysML) (Object Management Group 2012) defines a general purpose systems engineering language. It is worth mentioning that SysML does not make any modifications to UML State Machines, apart from excluding protocol state machines, noting that standard FSM are considered sufficient for expressing protocols too. The MARTE profile (Modeling and Analysis of Real-Time and Embedded Systems) introduces concepts to support the development of Real Time and Embedded Systems (RTES).

For the robotics domain, the OMG robotics domain task force (DTF) is promoting and extending OMG standards for the purpose of developing component based robotics systems. Several standards have been defined, including the Robotic Technology Component (RTC) upon which the OpenRTM framework is based (Ando, Suehiro, and Kotoku 2008), the Robotic Localization Service (RLS) and the Robotic Interaction Service (RoIS). Burmester et al. (Burmester, Giese, and Tichy 2004) introduce Mechatronic UML as an extension to UML for modeling hybrid real-time systems.

In contrast to profiling, the second approach of *meta-modeling* advocates defining a completely new meta-model from scratch. This process is carried out by defining the desired concepts and relationships of the new modeling language by using a so called meta-modeling language. This approach is mainly supported by the Eclipse Modeling Framework (EMF) project (Eclipse Foundation b) and its Ecore meta-modeling language, although OMG MOF could be used equally. Modeling with the micro-modeling framework (uMF) described in Chapter 6 also falls into the meta-modeling category. Which approach is more appropriate when is subject of ongoing discussion. Obvious advantages of the profiling approach are the ability to reuse significant amounts of models and of tooling. On the other hand, meta-modeling from scratch can yield much smaller models that are (arguably) better adapted to the needs of the respective domain.

The Architecture Analysis and Design Language (AADL) (SAE International) is a textual and graphical modeling language standardized by SAE. Although originally designed for the domain of avionics (thus the former name Avionics Analysis and Design Language), AADL has evolved into a more general modeling language for real-time and embedded systems. Similar to MARTE, AADL can model applications, hardware execution platforms and deployment. A comparison between both is given by Mallet (Mallet and Simone 2009). AADL has been criticized for not raising the abstraction level significantly compared to an implementation in a programming language (Delanote, Van Baelen, Joosen, and Berbers 2008).

AUTOSAR (Autosar Consortium 2003) is a component based software architecture and modeling language driven by a consortium of automotive OEMs and suppliers. The AUTOSAR mantra is “Cooperate on standards, compete on implementation” (Autosar Consortium 2003), therefore unlike UML or AADL no support for modeling behavior is provided. Early in the project AUTOSAR models were described using an UML profile, later on a meta-model based approach was chosen.

2.2 Domain Specific Languages

A *Domain Specific Language*, contrary to a general purpose language, is a language that has been specifically tailored to express the concepts of a particular domain. DSL have been used for decades, especially in Unix and were first described as *little languages* by Bentley (Bentley 1986). Famous DSL are the `make` language to describe software builds, `sed` and `awk` for text processing or XML to described hierarchically structured data. A comprehensive overview of research on DSL can be found in (van Deursen, Klint, and Visser 2000). (Spinellis 2001) and (Mernik, Heering, and Sloane 2005) describe patterns and tradeoffs involved in developing DSL.

DSL are central to Model Driven Engineering; any meta-model can essentially be understood as a DSL, although the term seems to be stronger associated with textual than graphical modeling languages. Generally, two types of DSL are distinguished (Fowler 2005). *External DSL* are constructed from scratch, usually by using tools such as lex and yacc or ANTLR[1]. In contrast, *internal* or *embedded DSL* are constructed within an existing host language. Examples of languages that are frequently used for developing internal DSL are bash, Haskell, Ruby or Lisp, though many others have been used too.

Both approaches have their merits: external DSL offer more freedom over the DSL syntax, but require more effort to implement, since a parser for that syntax must be realized. On the other side, internal DSL are typically constrained to a similar syntax as the host language, but in return can reuse the host language's infrastructure for parsing, error reporting and computing.

DSLs have a been used in robotics for a long time (Kaelbling 1987; Kaelbling 1988; Gat 1991). MAESTRO (Coste-Maniere and Turro 1997) is a language for specification, validation and control of robotic missions. Frob (Petersson, Hudak, and Elliott 1999) and AFRP (Hudak, Antony, Nilsson, and Peterson 2003) are internal DSL built using the Haskell language for programming robots based on the Functional Reactive Paradigm. Bjarnason et al. (Bjarnason, Hedin, and Nilsson 1999) describe a toolchain to interactively develop DSL. A case study based on an industrial robot programming language is presented and the need for parametrizable and composable DSL is identified (e.g to compose DSLs for specifying the application and motion control level). Two solutions, based on multi-layered grammars and procedure inheritance are proposed.

[1] Which are itself DSLs for describing parsers!

2.3 Separation of Concerns

The term *Separation of Concerns*, firstly used by Dijkstra (Dijkstra 1982), refers to modularizing a system such that the individual parts overlap as little as possible.

The paradigm of *Aspect Oriented Programming* (AOP) (Kiczales 1996) recognizes that any modularization is only optimal with respect to the chosen primary modularization criteria. Thus, secondary concerns are often spread out across the program or tightly coupled with other aspects (called *scattered* and *tangled* in AOP terminology). Canonical examples of such concerns are logging, caching or security checks. To avoid scattering and tangling, AOP languages introduce Aspects as first-class entities. Aspects consist of point-cuts and advice. Point-cuts specify where or when in a program a piece of advice shall be applied and typically takes the form of a programming language specific regular expression. Advice is executed on the matches of the point-cut (the so-called join points) and can modify or extend it, thereby automatically merging the concern. Applying aspects to program code is carried out by a tool called an *aspect weaver*.

The ideas of aspect orientation have been applied to a wide range of domains, including to MDE in form of Aspect Oriented Modeling (AOM) (Elrad, Aldawud, and Bader 2002). While this thesis does not use AOP or AOM tools directly, it is influenced by its ideas as well as the principle of Separation of Concerns (see the *5Cs* in Section 2.8).

2.4 Real-Time Systems

A *real-time system* can be defined as a system that delivers a response to a stimuli within bounded time. If this boundary is permitted to be violated, the system is called a *soft real-time system*. A software example for this is an audio player; while it is obviously desirable that music playback takes place smoothly, a sporadic “stuttering” will remain without consequences. This is not true for a *hard real-time system*. The term *hard* refers to the strictly bounded worst-case response time; violating this will have severe consequences such as damage of equipment or even loss of life. Classical examples are aerospace, automotive or chemical process control systems. Robotic systems may also exhibit different types of real-time constraints: for example, to guarantee safety a system may be required to complete the enabling of its brakes within a maximum time frame. A second example is a periodically triggered control loop that controls the *active compliance* of a robot system (Albu-Schäffer, Haddadin, Ott, Stemmer, Wimböck, and Hirzinger 2007). Besides not being allowed to

miss an update, such periodic systems often place further constraints on the permissible deviation from the expected trigger time. This deviation is called *jitter*.

2.5 Behavioral Modeling Languages and Formalisms

A wide variety of architectures and formalisms exist to model behavior of discrete or hybrid systems. This section gives a broad overview over the topic. Prior art related to state-based formalisms is discussed in detail in section 4.3.

Petri nets were introduced by Petri (Petri 1962) as a mathematical model to study concurrency. A good overview is given by Hruz and Zhou (Hrúz and Zhou 2007). Basic Petri nets consist of places, transitions and directed arcs. Places are connected to each other by an arc emanating from a place to a transition and a second arc emanating from the transition to the place. The state of a Petri net is represented by tokens residing on places and is called a marking. The marking changes atomically when the Petri net *fires*: tokens on the input place are consumed and placed on the output places of the transition. Weighted input arcs permit defining the amount of tokens required on the input place for the transition to fire, while weighted output arcs define the amount of tokens that will be placed on the output place. Petri net models can be analyzed to determine different properties, such as to show whether a certain marking can be reached from a given initial marking. A discussion of Petri net properties can be found in Hruz (Hrúz and Zhou 2007). Over the years, the basic Petri net model has been extended in various ways: colored petri nets extend the tokens with identity (the color) (Jensen 1987), which permits more compact descriptions. Ramchandani introduced *Timed Petri nets* in which firing of a transition takes non-zero time. An approach is described to use Petri-nets for modeling the interconnections of finite state machines. Petri Nets have also found application in Robotics, such as to describe assembly tasks (McCarragher 1994) (Rosell 2004). The IEC standardized sequential function charts (SFC) (International Electrotechnical Commission 2003) have ultimately been derived from Petri Nets and are discussed in chapter 4.

Reactive Systems are systems that respond to stimuli from their environment. Many models and tools have been developed for this class of systems. *Synchronous Languages* such as Esterel, Lustre, and Signal have been in use for decades for modeling real-time, safety critical, reactive systems. Akin to hardware description languages, synchronous languages rest on the synchrony hypothesis that defines that reactions to stimuli occur instantaneously and without delay. Since this simplification obviously does not hold true for physical

time, synchronous languages introduce a virtual clock, that permits satisfying this assumption by permitting reactions to complete before the next event arrives. The synchrony hypothesis greatly facilitates validation of temporal requirements and improves the determinism of the generated program. On the other hand the approach is restricted to closely collocated computations.

In contrast to synchronous languages, *Communicating Sequential Processes* (CSP) (Hoare 1978) (Hoare 1985) can be used to describe asynchronously executing processes that by means of message passing can engage in rendezvous with each other. Owing to the mathematical formalization, systems described in CSP can be analyzed to detect common errors such as dead or livelocks. CSP is a representative of the family of process calculi. The OCCAM programming language is based upon CSP.

The *Actor Model* (Hewitt, Bishop, and Steiger 1973) is closely related to CSP. The most prominent difference between the CSP and the Actor Model is that in CSP communicating an event (a message in Actor terms) is synchronous, since the sender side will effectively block until the receiving side participates in the rendezvous. In contrast, sending a message in the Actor model is asynchronous.

Supervisory Control Theory (SCT) (Ramadge and Wonham 1987) is a method introduced by Ramadge and Wonham for automatically synthesizing a discrete controller to supervise a discrete process. The system to be supervised can be described by the events it generates; these events can be understood as forming a *regular language* that can be modeled using a discrete finite automaton (DFA). Events are partitioned into the two sets of controllable and uncontrollable events. Given a specification of the desired behavior (also in form of a DFA), a third supervisor DFA can be generated that controls the controllable events with the goal of satisfying the specification in a maximally permissive way.

Discrete Event System Specification (DEVS) is a formal method introduced by Zeigler (Zeigler 1976) for modeling and simulating discrete reactive and hierarchically composable systems. Using DEVS, elementary systems are modeled using a variant of FSM called *atomic DEVS*. The FSM changes state either based on externally received events or when a certain time has elapsed (internal events). Composite systems are modeled using *Coupled DEVS*, which describes the components (either atomic or coupled DEVS) that form the system and how they are interconnected. A coupled DEVS model is closed under the coupling property, which means it can be transformed to an equivalent atomic DEVS model. This way, an execution environment for atomic DEVS is sufficient to simulate a coupled DEVS model.

Today DEVS is widely used and still subject of ongoing research. A multitude of extensions have been developed to permit use of DEVS for different applications

such as for hybrid or real-time systems. A recent overview can be found in Wainer et al. (Wainer and Mosterman 2011). Borland (Borland 2003) shows how to transform Statechart models to DEVS to exploit its simulation and validation capabilities.

Hybrid automata are a formalism to describe systems that exhibit both discrete and continuous properties (Henzinger 1996). A model consists of a DFA and a finite set of continuous variables described by ordinary differential equations. The discrete *control modes* are represented by a DFA, defining the initial condition, the invariant condition that must hold while the state is active and the flow condition governing the evolution of the continuous variables. The edges of the DFA are labeled with jump conditions that, together with the invariant condition of the edge, define when switches between control modes take place.

Timed Automata (Alur and Dill 1994) are a special case of hybrid automata where the continuous variables represent a set of real-valued clocks. Timed automata have been thoroughly studied and used for analyzing and proving properties of real-time systems.

BIP (Behavior, Interaction, Priority) is a methodology and language for modeling real-time components and architectures (Basu, Bozga, and Sifakis 2006). The focus of BIP is on modeling heterogeneous components; heterogeneity referring to component execution and interaction semantics (synchronous vs. asynchronous) and to variations in abstraction level of the models used. Components are constructed from three basic elements: atomic components, connectors and priority relations. Atomic components are defined as a set of ports, variables and a Mealy Machine (with guard predicates) that transitions when an interaction involving the specified port takes place. Connectors specify the nature of an interaction by defining which ports must and can be involved, thereby permitting to specify various degrees of synchronization between atomic components. Analog to atomic components, interactions may specify a guard predicate and a statement to be executed when the interaction takes place. Lastly, priorities permit to prioritize interactions that might be simultaneous enabled. Compound components are components that are defined by composing existing (atomic or compound) components, specifying connectors between these and defining their priorities. The BIP approach is supported by tooling for validation and C++ code generation.

Besalem et al. (Bensalem, de Silva, Ingrand, and Yan 2011) apply the BIP approach to construct a complex, autonomous robot system. To that end GenoM module specifications (Fleury, Herrb, and Chatila 1997) are automatically transformed to BIP components, which can then be formally verified to detect possible deadlocks. Furthermore, it is shown how connectors can be used to

define constraints that are checked at run-time. Abdellatif et al. (Abdellatif, Bensalem, Combaz, de Silva, and Ingrand 2012) extend this approach by introducing a textual DSL for specifying these constraints in a more succinct and user-friendly form; these constraints are then automatically transformed to the lower level connector based specification. Furthermore, a real-time version of BIP is described, that permits expressing real-time constraints of components using timed automata. The real-time BIP engine will schedule the activated transitions according to their priorities and report violations of deadlines.

The Abstract State Machines (ASM) Method (Börger and Stärk 2003) is a uniform, software and hardware systems engineering approach that aims to bring together requirement engineering with actual development. The approach is centered around the operational ASM model which permits describing systems at different abstraction levels; starting from a high level *ground model* that captures the requirements, the system is step-wise refined while simultaneously validation through simulation and more formal verification is carried out. Later, the ASM model can be used for testing and validating the implementation.

A basic ASM model consists of the following: a vocabulary that identifies the domain concepts (function-, predicate and domain names) and a set of states which assign values, called *interpretation*, to the vocabulary. Different types of transition rules define state changes by describing when and how the interpretation changes. This is achieved by using update instructions that modify the (memory) *location* of a value. During an ASM step all transition rules of the current state whose guards are true are executed simultaneously. As a consequence the system is switched to the next state, provided there are no inconsistencies. If there are, the system remains in the current state. Inconsistencies are defined by multiple updates of the same location within one step.

While formal methods have proven useful for developing highly reliable and provably correct systems, these methods have some disadvantages. Generally, formal methods are more costly than traditional development, due to the necessary, rigorous formalization. Moreover, formal methods typically do not deal well with legacy systems, since to be able to prove properties of systems it is necessary to model these entirely using the method in question. These limitations have spawned interest in so-called *lightweight formal methods* (Jackson and Wing 1996), (Agerholm and Larsen 1999). The lightweight approach suggests to focus on validation of selected subsystems opposed to validating the entire system, to use dedicated (domain specific!) validation tools opposed to universal ones and to carry out partial analysis when complete analysis is infeasible. Tools supporting lightweight verification are Alloy (Jackson 2002) or IDP (Wittocx, Mariën, and Denecker 2008).

2.6 Robot Software Architectures

Bass et al. (Bass, Clements, and Kazman 2003) define a software architecture as follows: "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

Important aspects of robotic architectures include the underlying paradigm used (Medeiros 1998), the programming system used to develop the architecture (Biggs and MacDonald 2003) and the communication mechanisms used to connect software elements (Shakhimardanov, Hochgeschwender, Reckhaus, and Kraetzschmar 2011). As pointed out by (Kortenkamp and Simmons 2008, pg 202), devising a robot architecture is (still) more of an art than a science. This is because it involves trading off between a number of requirements such as flexibility, usability or performance, which can be both difficult to measure and significantly influenced by personal preferences.

For a more detailed overview including a list of concerns for consideration when choosing an architecture, the interested reader is referred to the *Robotics Systems Architectures and Programming* chapter of the Springer Handbook of Robotics (Kortenkamp and Simmons 2008, p. 187). It is also worth noting that the coordination approach introduced in this thesis is not tied to any particular architectural choice.

2.7 Coordination Languages

A comprehensive introduction to Coordination is given by Arbab (Arbab 1998), who defines Coordination as follows: "[...] the study of the dynamic topologies of interactions among Interaction Machines, and the construction of protocols to realize such topologies that ensure well-behavedness."

Arbab classifies coordination languages according to two characteristics. Firstly, a language may be data- or control-oriented. For the former, coordination emerges as a result of interacting with data, while the latter is concerned with defining activation and deactivation of control flow. Secondly, a language may be endogenous or exogenous. An endogenous language requires computations to make use of specific primitives to be able to be coordinated (e.g. Linda), while an exogenous language can do so with coordination-agnostic computations. As pointed out by Arbab, endogenous languages have the fundamental disadvantage of intermixing computation with coordination.

A thorough survey of Coordination languages is given by Papadopoulos and Arbab (Papadopoulos and Arbab 1998).

Gelernter was the first to introduce an explicit coordination model and language in form of *Linda* (Gelernter 1985). This language defines a mechanism to communicate between and coordinate concurrent computations. To achieve that, four primitives are used that operate on a global, shared and associative data-structure called a tuple-space. Tuples can be read or written in a blocking or non-blocking manner, new tuples can be output and new processes can be created. In the context of this thesis, the fundamental insight of this approach is orthogonality (Gelernter and Carriero 1992). Orthogonality refers to the achieved separation of computation from coordination by use of a dedicated coordination language; this is essentially an early advocation of the concept of a coordination DSL.

Manifold (Arbab, Herman, and Spilling 1993) is an exogeneous, control-oriented coordination language to manage the interactions among concurrent computations. The primitives involved are processes, events, ports and streams. Processes define ports and can be of two types: *atomic processes* are black-box computational components. *Manifolds*, specified using the Manifold language, are used to coordinate atomic processes by connecting and disconnecting ports using streams and by reacting to events. The Reo model (Arbab 2004) extends Manifold and defines *components* as entities that execute in a *location*. Components communicate using connectors, which are essentially composed of a graph of nodes and channels. The coordination protocol between components is a result of the topology of channels within the connector.

A more recent example of a coordination language is the Hierarchical Timing Language (HTL) (Ghosal 2008), which focuses on specification, verification and compilation of hierarchically decomposed real-time systems.

The rFSM formalism presented in Chapter 4 is an exogeneous, control-oriented Coordination model and language, designed for dynamic[2] real-time safe Coordination of component based systems.

Ptolomy is a methodology and framework (Eker, Janneck, Lee, Liu, Liu, Ludvig, Neuendorffer, Sachs, and Xiong 2003) that supports constructing systems from heterogeneous components. Heterogeneity refers to the paradigm (and thus assumptions) used to realize individual components, as for instance continuous time models like ODE or discrete FSM models. Ptolomy addresses the problem that a composition of components developed with different paradigms will not necessarily be well-defined. To achieve this, the fundamental idea is to compose heterogeneous components using a model of computation (MOC) that governs

[2] dynamic referring to the ability to support run-time changes

both control and dataflow among the components. Since this composition can be considered atomic at the next level, it can again be aggregated with other components using a different MOC. That way, Ptolomy permits treating systems as locally homogeneous and well-defined, while simultaneously enabling hierarchical composition of heterogeneous models. Today Ptolomy supports a wide variety of MOC including CSP, ODE, different forms of Process Networks, Synchronous Languages and FSM.

2.8 Separating the four C's

The separation of computation and coordination advocated by the field of coordination languages is an important step towards reusability. Yet, in the context of development of reusable component based systems the separation of further concerns has been proposed: Andrade et al. (Andrade, Fiadeiro, Gouveia, and Koutsoukos 2002) suggest to separate systems according to the three aspects of Computation, Coordination and Configuration. Radestock and Eisenbach (Radestock and Eisenbach 1996) propose separation of the four aspects (the 4C's) of Communication, Computation, Configuration and Coordination. Bruyninckx et al. (Bruyninckx, Hochgeschwender, Gherardi, Klotzbuecher, Kraetzschmar, Brugali, Shakhimardanov, Paulus, Reckhaus, Garcia, Faconti, and Soetens) extend the 4C's with the concern of *Composition*, leading to the 5C's. This extension emphasizes the fact that separation of concerns is only part of the solution to achieve reusability, the other challenge being the re-composition of these concerns into a working system. One approach permitting composition of multiple, possibly overlapping concerns is described by (Tarr, Ossher, Harrison, and Sutton 1999), a second using DSL is described in this thesis (see Section 4.10.2). A working hypothesis of this thesis is that high levels of reusability and robustness of component based robotic systems can be achieved when these aspects are kept separated.

2.9 Conclusion of Literature Survey

This chapter has given a broad overview of the state-of-the-art tools and methods for developing complex robotic systems. Although individual approaches satisfy some of the requirements of this thesis, none satisfy all. Most modern finite state machine and statechart formalisms provide rich sets of primitives but pay little attention to **extensibility and composability** with domain specific models. Formal methods permit building highly reliable systems by means of automated verification techniques. On the other hand, these approaches typically

require rigorous application, which conflicts with the goal of **supporting legacy components and subsystems**. Moreover, the majority of related work does not follow an approach of **separating concerns**. Most approaches couple several concerns and thus reduce reusability or robustness. Of those approaches that do separate concerns, none separate the four or even five C's. Modern Coordination languages like Reo go furthest in separating Coordination from Computation, but their rich semantics do not easily lend themselves to application in hard real-time systems. Lastly, most of the prior art described aims at providing generic and universally applicable tools and techniques. In contrast, this thesis explores the benefits of **exploiting domain specific knowledge**.

Chapter 3

Hard real-time Control and Coordination using Lua*

3.1 Abstract

Control and Coordination in industrial robot applications operating under hard real-time constraints is traditionally implemented using languages such as C/C++ or Ada. We present an approach to use Lua, a lightweight and single threaded extension language that has been integrated in the Orocos RTT framework. Using Lua has several advantages: increasing robustness by automatic memory management and preventing pointer related programming errors, supporting inexperienced users by offering a simpler syntax and permitting dynamic changes to running systems. However, to achieve deterministic temporal behavior, the main challenge is dealing with allocation and recuperation of memory. We describe a practical approach to real-time memory management for the use case of Coordination. We carry out several experiments to validate this approach qualitatively and quantitatively and provide robotics engineers the insights and tools to assess the impact of using Lua in their applications.

*This chapter is based on Klotzbücher, M., Bruyninckx, H. (2011). Hard Real-Time Control and Coordination of Robot Tasks using Lua. *Proceedings of the Thirteenth Real-Time Linux Workshop*. Czech Technical University, Prague, 20-22 October 2011 (pp. 37-43) Open Source Automation Development Lab (OSADL) eG.

3.2 Introduction

This work takes place in the context of component based systems. To construct an application, computational blocks are instantiated and interconnected with anonymous, data-flow based communication. Coordination refers to the process of managing and monitoring these functional computations such that the system behaves as intended. Keeping Coordination separate from Computations increases reusability of the latter blocks as these are not polluted with application specific knowledge. Examples of typical coordination tasks are switching between controllers upon receiving events, reconfiguring computations (e.g. changing controller gains) and dealing with erroneous conditions. A complex example of an robot applications constructed using this paradigm can be found in Smits et al. (Smits, De Laet, Claes, Bruyninckx, and De Schutter 2008).

To implement coordination we propose to use the Lua extension language (Ierusalimschy, de Figueiredo, and Filho 1996). Using an interpreted language for this purpose has several advantages. Firstly, the robustness and hence safety, of the system is increased. This is because scripts, in contrast to C/C++, can not easily crash a process and thereby bring down unrelated computations that are executed in sibling threads. This property is essential for the aspect of coordination, which, as a system level concern, has higher robustness requirements than regular functional computations.

Secondly, the use of a scripting language facilitates less experienced programmers not familiar with C/C++ to construct components. This is important for the robotics domain, where users are often not computer scientists. Moreover, rapid prototyping is encouraged while leaving the option open to convert parts of the code to compiled languages after identification of bottlenecks. At last, the use of an interpreted language permits dynamic changes to a running system such as hot code updates. This is essential for building complex and long running systems that can not afford downtime.

The major challenge of using Lua in a hard real-time context is dealing with allocation and recuperation of memory. Previously we sketched two strategies to address this: either running in a zero-allocation mode and with the garbage collector deactivated or in a mode permitting allocations from a pre-allocated memory pool using a O(1) allocator and with active but controlled garbage collection (Klotzbuecher, Soetens, and Bruyninckx 2010). In practice, especially when interacting with C/C++ code it may be inconvenient to entirely avoid collections, hence now we consider it necessary to run the garbage collector.

The rest of this chapter is structured as follows. The next section gives an overview over related work. Section 3.4 describes how we address the issue of memory management in a garbage collected language used for coordination.

Section 3.5 describes four experiments with the two goals of demonstrating the approach and giving an overview of the worst-case timing behavior to be expected. Robustness is discussed in the context of the last experiment, a coordination statechart. We conclude in section 3.6.

3.3 Related work

The Orocos RTT framework (Soetens 2006) provides a hard real-time safe scripting language and a simple state machine language. While both are much appreciated by the user community, the limited expressivity of the state machine model (e.g. the lack of hierarchical states) and the comparably complex implementation of both scripting language and state machines have been recognized as shortcomings. This work is an effort to address this.

The real-time Java community has broadly addressed the topic of using Java in hard real-time applications (Real-Time for Java Expert Group (RTJEG) 2005). The goal is to use Java as a replacement to C/C++ for building multi-threaded real-time systems. To limit the impact of garbage collection, parallel and concurrent collection techniques are used (Sun Microsystems 2006). For our use-case of building domain specific coordination languages we chose to avoid this complexity, as coordination can be defined without language level concurrency. In return this permits taking advantage of the deterministic behavior of a single threaded scripting language.

The Extensible Embeddable Language (EEL) (Olofson 2005) is a scripting language designed for use in real-time application such as audio processing or control applications. Hence, it seems an interesting alternative to Lua. Lua was ultimately chosen because of its significantly larger user community.

3.4 Approach

To achieve deterministic allocations, the Lua interpreter was configured to use the Two-Level Segregate Fit (TLSF) O(1) memory allocator (Masmano, Ripoll, Balbastre, and Crespo 2008). This way, memory allocations are served from a pre-allocated, fixed pool. Obviously, this raises the question of how to determine the required pool size such that the interpreter will not run out of memory. We address this in two ways. Firstly, by examining memory management statistics the worst case memory consumption of a particular application can be determined and an appropriate size set. Due to the single threaded nature of Lua a simple coverage test can give high confidence that this value will not

be exceeded in subsequent runs. Furthermore, to achieve robust behavior the current memory use is monitored online and appropriate actions are defined for the (unlikely) case of a memory shortage. What actions are appropriate depends on the respective application.

This leads to the second challenge for using Lua in a hard real-time context, namely garbage collection. In previous work (Klotzbuecher, Soetens, and Bruyninckx 2010) we suggested to avoid garbage collection entirely by excluding a set of operations that resulted in allocations. However, in practical applications that transfer data between the scripting language and C/C++ this is not always possible. Consequently the garbage collector can not be disabled for long periods and must be either automatically or manually invoked to prevent running out of memory. For achieving high determinism, it is necessary to stop automatic collections and to explicitly invoke incremental collection steps when the respective application permits this. Only this way it can be avoided that an automatic collection takes place at an undesirable time.

The Lua garbage collector is incremental, meaning that it may execute the garbage collection cycle in smaller steps. This is a necessary prerequisite for achieving low garbage collection latencies, although of course no guarantee; ultimately the latency depends on various factors such as the amount of live data, the properties of the live data[1] and the amount of memory to be freed. The control and coordination applications we have in mind generally tend to produce little garbage, because the scripting language is primarily used to combine calls to C/C++ code in meaningful ways. Nevertheless, to increase robustness the worst-case duration of collection steps can be monitored for the purpose of dealing robustly with possible timing violations.

The following summarizes the basic approach. First, the desired functionality is implemented and executed with a freely running garbage collector. This serves to determine the maximum memory use from which the necessary memory pool size can be inferred by adding a safety margin (e.g. the maximum use times 2). Next, the program is optimized to stop the garbage collector in critical paths and incremental steps are executed explicitly. The worst case timing of these steps is benchmarked, as is the overall memory consumption. The program is then executed again with the goal to confirm that the explicitly executed garbage collection is sufficient to not run low on memory.

[1] In Lua, for instance, tables are collected atomically. Hence large tables will increase the worst-case duration of an incremental collection step.

3.5 Experiments

In this section we describe the experiments carried out to assess worst-case latencies and overhead of Lua compared to using C/C++ implementations. All tests are executed using Xenomai (Gerum 2004) (v2.5.6 on Linux-2.6.37) on a Dell Latitude E6410 with an Intel i7 quad core CPU and 8 GiB of RAM, with real-time priorities, current and future memory locked in RAM and under load.[2] Apart from the cyclictest all tests are implemented using the Orocos RTT (Soetens 2006) framework. The source code is available here[3].

3.5.1 Lua Cyclictest

The first test is a Lua implementation of the well known cyclictest (Gleixner 2011). This test measures the latency between scheduled and real wake up time of a thread after a request to sleep using `clock_nanosleep(2)`. The test is repeated with different, absolute sleep times. For the Lua version, the test is run with three different garbage collector modes: `Free`, `Off` or `Controlled`. `Free` means the garbage collector is not stopped and hence automatically reclaims memory (the Lua default). `Off` means the allocator is stopped completely[4] by calling `collectgarbage('stop')`. `Controlled` means that the collector is stopped and an incremental garbage collection step is executed after computing the wake up time statistics (this way the step does not add to the latency as long as the collection completes before the next wake up).

The purpose of this test is to compare the average and worst case latencies between the Lua and C version and to investigate the impact of the garbage collector in different modes.

Results Table 3.1 summarizes the results of the cyclictest experiments. Each field contains two values, the average (“a”) and worst case (“w”) latency given in microseconds, that were obtained after fifteen minutes of execution.

Comparing the C cyclictest with the Lua variants as expected indicates that there is an overhead of using the scripting language. The differences between the three garbage collection modes are less visible. Table 3.2 shows the average of the worst case latencies in microseconds and expressed as a ratio to the average

[2] `ping -f localhost`, and `while true; do ls -R /; done`.

[3] http://people.mech.kuleuven.be/~mklotzbucher/2011-09-19-rtlws2011/source.tar.bz2

[4] This is possible because the allocations are so few that the system does not run out of memory within the duration of the test.

sleep time	500	1000	2000	5000	10000
	a, w	a, w	a, w	a, w	a, w
C	0, 35	0, 31	0, 45	1, 35	1, 30
Lua/free	2, 41	2, 39	3, 39	3, 45	5, 46
Lua/off	2, 38	2, 39	3, 38	3, 43	5, 38
Lua/ctrl	2, 38	2, 42	3, 37	3, 36	5, 46

Table 3.1: Cyclictest results (μs)

test	WC avg (μs)	ratio to C
C	35.2	1
Lua/free	42	1.19
Lua/off	39.2	1.11
Lua/ctrl	39.8	1.13

Table 3.2: Comparison of impact of garbage collector modes

worst case of C. Note that the average of a worst-case latency is only meaningful for revealing the *differences* between the four tests, but not in *absolute* terms. A better approach might be to base the average on the 20% worst-case values.

This table reveals (unsurprisingly) that a freely running garbage collector will introduce additional overhead in critical paths. Running with the garbage collector off or triggered manually at points where it will not interfere adds approximately 11% and 13% respectively compared to the C implementation. Of course the first option is only sustainable for finite periods. 13% of overhead does not seem much for using a scripting language, however it should be noted that this is largely the result of *only* traversing the boundary to C twice: first for returning from the sleep system call and secondly for requesting the current time.

3.5.2 Event messages round trip

The second experiment measures the timing of timestamped event messages sent from a requester to a responder component, as shown in Figure 3.1. The test simulates a simple yet common coordination scenario in which a Coordinator reacts to an incoming event by raising a response event, and serves to measure the overhead of calls into the Lua interpreter. The test is implemented using the Orocos RTT framework and is implemented using event driven ports connected by lock free connections. Both components are deployed in different threads.

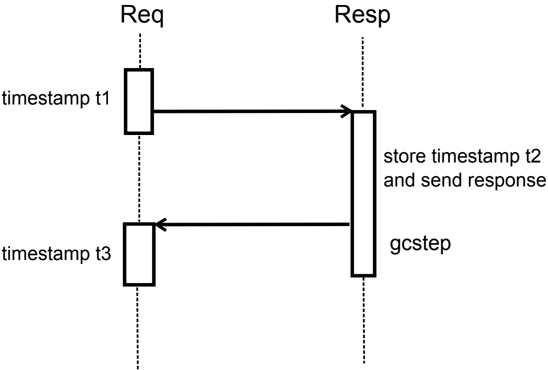

Figure 3.1: Sequence diagram of event round trip test.

	req (μs)	resp (μs)	total (μs)	Lua/C of total
	a, w	a, w	a, w	a, w
C	9, 37	7, 18	16, 50	-
Lua	15, 47	11, 59	26, 106	1.63, 2.12

Table 3.3: Results of event round trip test.

Three timestamps are recorded: the first before sending the message, the second at the responder side and the third on the requester side after receiving the response. The test is executed using two different responder components implemented in Lua and C++.

In this setup the Lua responder takes advantage of the fact that the requester component will wait for 500us before sending the next message. This permits executing an incremental garbage collection step after sending each response. If this assumption could not be made, the worst-case garbage collection delay would have to be added to the response time (as is the case for experiment 3.5.3).

Results Table 3.3 summarizes the average (“a”) and worst-case (“w”) duration of this experiment for the request ($t2 - t1$), response ($t3 - t2$) and total round trip time ($t3 - t1$); all values in microseconds.

On average, the time for receiving a response from the Lua component is 1.6 times slower than using the C responder. The worst case is 2.2 times slower. Of

the 1 MiB memory pool, a maximum of 34% was in use. It is worth noting that for the initial version of this benchmark, the response times were approximately eight times slower. Profiling revealed that this was caused by inefficient access to the time-stamp message; switching to a faster foreign function interface yielded the presented results.

3.5.3 Cartesian Position Tracker

The following two experiments illustrate more practical use cases. The first experiment compares both a Lua and C++ implementation of a so-called "Cartesian position tracker", typical in robotics, and running at 1KHz, by measuring the duration of the controller update function. In contrast to the previous example the incremental garbage collection step is executed during the controller update and hence contributes to its worst case execution time.

Listing 3.1 shows the simplified code of the update function. Note that `diff` function is a call to the Kinematics and Dynamics Library (KDL) (Smits, Bruyninckx, and Aertbeliën 2001) C++ library, hence the controller is not implemented in pure Lua. This is perfectly acceptable, as the goal is not to replace compiled languages but to improve the simplicity and flexibility of composing existing primitives.

```
pos_msr = rtt.Variable("KDL.Frame")
pos_dsr = rtt.Variable("KDL.Frame")
vel_out = rtt.Variable("KDL.Twist")
local vel, rot = vel_out.vel, vel_out.rot

function updateHook()
  if pos_msr:read(pos_msr) == 'NoData' or
     pos_dsr:read(pos_dsr) == 'NoData' then
    return
  end

  diff(pos_msr, pos_dsr, vel_out, 1)

  vel.X = vel.X * K[0]
  vel.Y = vel.Y * K[1]
  vel.Z = vel.Z * K[2]
  rot.X = rot.X * K[3]
  rot.Y = rot.Y * K[4]
  rot.Z = rot.Z * K[5]

  vel_out:write(vel_out)
  luagc.step()
end
```

Listing 3.1: Cartesian Position Tracker

Note that for Lua versions prior to 5.2 invoking the incremental garbage collector (`collectgarbage('step')`) restarts automatic collection, hence

type	duration (μs)	Lua/C
	a, w	a, w
C	5, 19	-
Lua	68, 128	13.6, 6.7

Table 3.4: Results of Cartesian Position Tracker Benchmark

`collectgarbage('stop')` must be invoked immediately after the first statement. The custom `luagc.step` function executes both statements.

Results Table 3.4 summarizes the results of the worst case execution times in microseconds. The average execution time is approximately 14 times, the worst case duration 7 times slower than the C version. The worst case garbage collection time measured was 29us, of the 1MiB large memory pool a maximum of 34% was in use.

In the current implementation the majority of both execution time spent and amount of garbage generated results from the multiplication of the `K` gains with the output velocity. If performance needed to be optimized, moving this operation to C++ would yield the largest improvement.

3.5.4 Coordination Statechart

The second real-world example is a coordination Statechart that is implemented using the Reduced Finite State Machine (rFSM) domain specific language (Klotzbuecher 2011), a lightweight Statechart execution engine implemented in pure Lua. The goal is to coordinate the operation of grasping an object in an uncertain position. The grasping consists of two stages: approaching the object in velocity control mode and switching to force control for the actual grasp operation when contact is made. This statechart is shown in Figure 3.2.

The real-time constraints of this example depend largely on the approach velocity: if the transition to the grasp state is taken too late, the object might have been knocked over. To avoid the overhead of garbage collection in this hot path, the collector is disabled when entering the `approach` state and enabled again in `grasp` after the respective controllers have been enabled.

Besides the actual grasping, it is necessary to monitor the memory use to avoid running out of memory. With an appropriately sized memory pool and sufficient garbage collection steps, such a shortage should not occur. Nevertheless, to

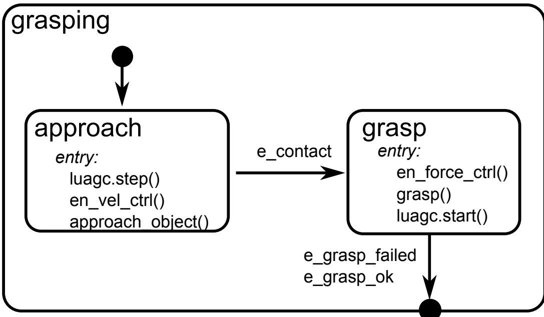

Figure 3.2: Coordinating the grasping of an object.

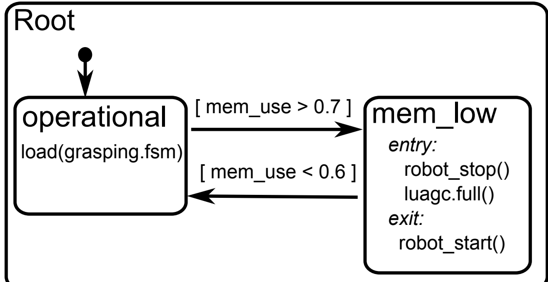

Figure 3.3: Dealing with low memory.

guarantee robust and safe behavior this condition must be taken into account and the robot put into a safe state. This is shown in Figure 3.3.

As the grasping task can only take place while enough memory is available, it is defined as a substate of `operational`. This way, the structural priority rule of the Statechart model (Harel and Naamad 1996) guarantees that the transition to `mem_low` has always higher priority than any transitions in the grasping state machine.

Identifying the required memory pool size currently has to be done by empirically measuring the maximum required memory of a state machine and adding a safety margin. To avoid this, it would be desirable to infer the expected memory use from the state machine description. Predicting the static memory used by the state machine graph is straightforward (Klotzbuecher, Soetens, and Bruyninckx 2010); also the run-time memory use of the rFSM core is predictable as it depends on few factors such as the longest possible transition and the maximum number of events to be expected within a time step. However, predicting

the memory use of the user supplied programs would require a more detailed analysis/simulation, which is currently outside of the scope of this work. In robotics most user supplied programs are in C/C++.

Results The previously described grasping coordination Statecharts are tested by raising the events that effect the transitions from `grasping`, `approach` to `grasp`. The time is measured from receiving the `e_contact` event until completing the entry of the `grasp` state. After this, the same sequence of events is repeated. The functions for enabling the controller are left empty, hence the pure overhead of the FSM execution is measured. Running the test repeatedly for five minutes indicates a worst-case transition duration between `approach` and `grasp` of 180us. The memory pool size was set to 1 MiB and the TLSF statistics report a maximum use of 58%. To test the handling of low memory conditions, in a second experiment the collector is not started in the `grasp` state. As a result no memory is recovered, eventually leading to a low memory condition and a transition to the `mem_low` state. For this test the worst case maximum memory use was as expected 70%.

This test does not take into account the latencies of transporting an event to the state machine. For example, when using the Orocos RTT event driven ports, the experiments from Section 3.5.2 can complement this one. Moreover it should be noted that so far no efforts have been put into minimizing rFSM transitions latencies; we expect some improvement by optimizing these in future work.

Robustness considerations As described, basic robustness of coordination state machines is achieved by monitoring of memory and current real-time latencies. However, the system level concern of coordination unfortunately combines the two characteristics of (i) requiring higher robustness than functional computations and (ii) being subject to frequent late modifications during system integration, the latter of course being susceptible to introduce new errors. The combination of scripting language and rFSM model can mitigate this effect in two ways. Firstly the scripting language inherently prevents fatal errors caused by memory corruption, thereby making it impossible to crash the application. Secondly, rFSM statecharts execute Lua user code in safe mode[5]. This way errors are caught and converted to events that again can be used to stop the robot in a safe way.

[5] Using the Lua `pcall` function

3.6 Conclusions

We have described how the Lua programming language can be used for hard real-time coordination and control by making use of an O(1) memory allocator, experimentally determining worst-case memory use and manually optimizing garbage collection to not interfere in critical paths. Several experiments are carried out to determine worst-case latencies and to illustrate the approach.

As usual, benchmark results should be judged with caution and mainly serve to remind that appropriate validation should be repeated for each critical use. In particular when real-time allocation and collection is involved, run time validation of real-time constraints must be considered as an integral part of the application.

The major shortcoming of the current approach is that worst-case memory use can be difficult to predict. To deal with this we currently allocate additional safety margins. As the overall memory usage of the Lua language is comparably small, such a measure will be acceptable for many systems, save the very resource constrained.

To conclude, we believe the results demonstrate the feasibility of our approach to use a scripting language for hard real-time control and coordination and permits to significantly improve robustness and safety of a system. The price of these improvements are (i) increased, yet bounded worst-case latencies, (ii) additional computational overhead, as well as (iii) requiring extra effort such as manual scheduling of garbage collection. In summary, we believe this constitutes a modern and practical approach to building hard real-time systems that shifts the focus from lowest possible latency to sufficient latency while maximizing reliability.

Future work will take place in two directions. On the high level we are exploring how to automatically generate executable domain specific languages from formal descriptions. Implementationwise we intend to investigate if and how the presented worst case timing behavior can be improved by using the luajit (Pall 2011) implementation, a high performance just-in-time compiler for Lua.

Chapter 4

Coordinating Robotic Tasks and Systems using rFSM Statecharts*

4.1 Abstract

Coordination is a system-level concern defining execution and interaction semantics of functional computations. Separating coordination from functional computations is a key principle for building complex, robust and reusable robotic systems. This work introduces a minimal variant of Harel statecharts called rFSM designed to model coordination of robotic tasks and systems with a minimal number of semantic primitives. Firstly, the semantics of the rFSM language are derived by analyzing state-of-the-art discrete event models and implementations and extracting a motivated and semantically well-defined subset that is considered best practice for the domain of robotic coordination. Secondly, a real-time capable reference implementation of rFSM is presented, which has been loosely integrated into the OROCOS/RTT framework. The application of rFSM is illustrated using a detailed description of a dual robot coordination problem. Lastly, several best practices and patterns are presented with the goal of i) supporting development of robust Coordination models, ii) illustrating how limitations of the statechart model can be overcome by

*This chapter is based on Klotzbücher, M., Bruyninckx, H. (2012). Coordinating Robotic Tasks and Systems with rFSM Statecharts. *JOSER: Journal of Software Engineering for Robotics*, 3 (1), 28-56.

extending the execution semantics, and iii) offering guidance in designing pure coordination components that optimize reusability.

4.2 Introduction

The design of todays complex robot systems is driven by multiple and often partically conflicting requirements. Apart from the primary functionality, these requirements commonly include the need to reuse existing parts, for the system to behave robust or safe in the presence of errors or to facilitate reconfiguration of the system due to changed requirements.

The extent to which these goals can be fulfilled is strongly influenced by the approach used to divide the system into parts. This work builds on the suggestion of Radestock and Eisenbach (Radestock and Eisenbach 1996) to separate the systems according to the four concerns of Communication, Computation, Configuration and Coordination: Communication defines how entities communicate. Computation defines the functionality and hence what is communicated. Configuration defines how computations are configured and which computations communicate with each other. Lastly, Coordination is responsible for managing the individual entities such that the system as a whole behaves as specified.

Most of today's robotic software frameworks support the separation of Communication, Computation and Configuration. For example, the Orocos RTT framework (Soetens 2006) permits to separately specify communication connections between components and the respective parameters such as buffering policies. Component configurations can be defined separately and need only be applied to the respective component instances at runtime. Similarly, the ROS framework (Willow Garage 2008) permits separate storage of node configuration using the so called parameter service.

In contrast, the concern of Coordination is not yet recognized as a first-class design aspect in many of today's complex robotic systems. Instead, Coordination is often implicitly incorporated into Computation and Communication. For example, in the Orocos RTT framework prior to version 2, coordination was often realized using the RTT::Event mechanism. Using this, a computational component declares the ability to raise an event represented as a parametrized function. Other computational components register handlers matching the event interface, which are then called to be notified of the event occurrence. As a result of this tight coupling, the event recipient is polluted with system-level coordination that limits its reusability. Moreoever, such implicit coordination

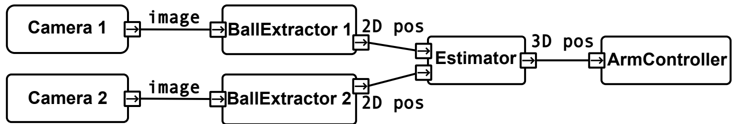

Figure 4.1: Data-flow architecture of Ball tracking application.

can lead to reduced robustness by requiring to compromise between reusability and robustness. The following example further elaborates these issues.

Although the idea of explicit coordination is not limited to component based systems, we assume this context throughout this chapter, since the majority of modern robotics software frameworks (Soetens 2006; Ando, Suehiro, and Kotoku 2008; Korean Institute for Advanced Intelligent Systems ; Quigley, Conley, Gerkey, Faust, Foote, Leibs, Wheeler, and Ng 2009) are component oriented. When using the term coordination or computation, we are referring to the concern, unless explicitely stated otherwise as in *coordination component*.

4.2.1 Motivating example

The following example introduces the concern of coordination. A ball swinging on a string is observed using two cameras and shall be followed by a robot manipulator. The data-flow component diagram in Figure 4.1 shows the involved computational components and the communicated data. 2D ball positions are extracted from the camera images by BallExtractor components and passed to an estimation component. The estimated 3D position is then sent to the RobotController actuating the robot arm. To avoid confusion with state machine diagrams, we utilize the SysML flowport notation (Object Management Group 2012), a small box with an arrow pointing in or out to describe component input or output data-flow ports respectively.

This system behaves as intended for the nominal case of the ball being visible to the cameras. However, if the ball swings out of the observed camera range the behavior is not well-defined, since different estimators will produce different results; for instance an estimator based on a constant velocity model will predict the ball motion to continue with the last estimated velocity, while one based on a constant position model will continuously predict the last estimated position.

This example constitutes a typical coordination problem, characterized by undesirable behavior at the system level even though the individual parts are functioning correctly.

Assume the requirement that the robot arm shall stop close to the last observed ball position. A naive solution to this problem would be to extend the estimator to stop the robot controller once the ball left the field of view of the camera. This solution is suboptimal for several reasons: firstly, the reusability of the estimator is severely reduced by adding this application-specific feature. Secondly, the exchange of this component with one not making this application-dependent assumption is prevented. Lastly, the solution provides limited robustness in case of communication failures between BallExtractor and Estimators, even though short interruptions may well be dealt with gracefully.

The latter shortcoming illustrates the previously mentioned compromise between robustness and reusability: the developer is forced to choose between either reduced reusability or to accept undefined behavior in corner cases or in the event of errors. The proposed solution to avoid these shortcomings is described by the following steps:

1. **Making relevant internal state visible**: the estimation component is extended to raise two events: `e_untracked` and `e_tracked`[1] when the ball becomes invisible and visible respectively (i.e. when the confidence in the ball position drops beneath/rises above a configurable threshold). This extension is both generic and non-intrusive to the estimator, as it only makes its internal state explicit but does not alter its behaviour.

2. **Introducing explicit coordination**: The dependency between Estimator and ArmController is encapsulated by introducing a separate Coordinator entity (typically, but not necessarily, deployed in a component itself) that reacts to the estimator's events. The state machine in Figure 4.2 models the desired behavior: the events `e_untracked` and `e_tracked` trigger the transitions from the nominal `following` state to `paused` and back respectively. The robot arm is stopped when entering and restarted when exiting the `paused` state.

This way, the reusability of the estimator is preserved while the desired behavior is specified in an explicit manner. Moreover, the proposed Coordination will deal gracefully with communciation failures between BallExtractor and Estimator; a sufficiently long interruption will result in the raising of the `e_untracked` event that stops the robot, while a short interruption will be dealt with gracefully. If a distinct reaction to this condition is required, it can be easily specified by extending the Coordination FSM to react to an event that represents the loss of communication.

[1] To improve readability events are prefixed with e_ throughout this chapter.

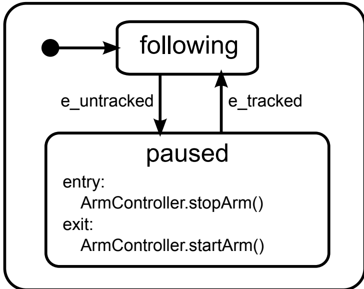

Figure 4.2: Ball-tracking coordination state machine.

4.2.2 Contributions

This chapter advocates the introduction of explicit Coordination components into every robotics software system and offers guidelines to do so. Its contributions can be divided into four parts. Firstly, it presents a study of task and system coordination mechanisms with respect to their suitability for the robotics domain. Secondly, based on the identified shortcomings of existing models a minimal variant of Harel statecharts addressing these issues is proposed, in form of the restricted Finite State Machine (rFSM) model. The term minimal is to be understood in terms of *the smallest number of primitives necessary for humans to construct practical coordination statecharts*, and not as minimal, yet with equal computational expressiveness. Thirdly, a real-time safe reference implementation of rFSM as an executable Domain Specific Language (DSL) is presented and the integration into robotic frameworks described. Lastly, to facilitate adoption of the described approach, a set of best practice patterns and guidelines are provided, to help system and component developers with the design of efficient, robust and reusable coordination.

4.2.3 Related Work

This Section summarizes previous work on analysis, formalization and classification of finite state machine semantics. A detailed review of state-of-the art coordination models themselves is presented in Section 4.3.

Von der Beeck (von der Beeck 1994) compares twenty different statechart variants according to a set of distinctive features. Some suggestions are given about which features should be included in a statechart formalism, however it is not clear what the motivation for inclusion or not is.

Eshuis (Eshuis 2009) identifies a set of constraints for which subsets of the three common statechart semantics—STATEMATE, Fixpoint and UML—behave identically. Besides suggesting improvements to statechart semantics, this work differs significantly from ours that is focussed on using statecharts for the particular purpose of coordination.

Breen (Breen 2004) identifies several shortcomings of Harel statecharts for the purpose of system specification. Most issues are related to parallel states (modeling concurrency), and hence do not affect our rFSM model that excludes this model element. Breen observes that the difficulty of understanding the complex relationship between parallel states (which is a consequence of broadcast event communication and the inevitable non-determinism of communication delays) can be avoided by simply using separate statecharts instead of parallel states. This observation, albeit for different reasons discussed in Section 4.8, confirms our approach to exclude the parallel state element. Furthermore, Breen describes the following shortcoming of the statechart model that is related to hierarchy and also affects the rFSM model: by observing just the graphical model of a state machine, the designers' motivation for using hierarchy is not always obvious. This is due to multiple use-cases for using hierarchical states. To this end, the following use-cases are identified: clustering, abstraction, transition dependence, state variable instantiation and expression of constraints. Breen concludes by pointing out that the described problems will not be significant in relatively simple models.

Simons (Simons 2000) examines the semantics of UML State machines and suggests a revised interpretation. Extensions are identified that can be considered either redundant or harmful to the compositional properties of the model. The latter class include the inversion of conflict resolution priorities in UML with respect to Harel statecharts and the special semantics of completion transitions. Moreover, the revised semantics attempt to improve the differentiation between statecharts and flowcharts. The rFSM semantics described later follow several, but not all, of these suggestions.

4.2.4 Outline

The rest of this work is structured as follows. Section 4.3 reviews existing coordination models and implementations. After briefly introducing the rFSM model in Section 4.4, the insights gained from these reviews are used in the discussion of structural, execution and event semantics in Sections 4.5–4.7, that ultimately lead to the derivation of rFSM. Section 4.8 is dedicated to motivate the exclusion of the widely used parallel state element from the rFSM model. Section 4.9 describes the reference implementation that was developed. Best

practices and patterns are described in Section 4.10. Section 4.11 presents a detailed example that illustrates step by step the process of developing the coordination for a dual robot haptic coupling system. We provide a discussion of our contributions in Section 4.12, and conclude in Section 4.13.

4.3 Review of Coordination Models

4.3.1 Classical Finite State Automatons

Finite state machines (FSM), sometimes called finite state automata, are mathematical models of behaviour. FSM are widely used, ranging from application in computational linguistics to language parsing and artificial intelligence. A FSM consists of a finite set of states connected by transitions.

Commonly, two types of FSM are distinguished: *acceptors* and *transducers*. Acceptors, which are also known as recognisers, return a binary result of whether or not a certain input was recognized, and are mostly used in language recognition. Transducers can generate output with the help of *actions*, and can be divided into two classes: *Moore* and *Mealy* state machines. In the Moore model, the output is a function only of the current state, while in the Mealy model the output is a function of the input and the current state. While both models are equally expressive, in practice often formalism as UML 2 state machines are used, where the output can be a function only of state or of state and input.

4.3.2 Harel Statecharts

The *Harel statechart* (Harel 1987; Harel and Naamad 1996) is the first state machine based formalism to extend classical flat automatons with hierarchy, parallelism and broadcast events. Hierarchical states may contain states themselves, thereby facilitating abstraction and modularization of larger models. Additionally, hierarchy enables transition prioritisation using a strategy named structural priority: in case of two or more conflicting transitions the one with the least nested source state (within the global hierarchy) takes priority. Parallelism permits substates contained in a so called AND state to be simultaneously active. Broadcast events are a more controversial feature (Breen 2004) causing that events raised in one sub-statechart (such as a parallel substate) are observed and hence can trigger transitions throughout the entire statechart, thus violating the scope introduced by each level of hierarchy. The aforementioned controversy

arises from the fact that the behaviour of a large statechart can become considerably intricate, especially in combination with statechart parallelism.

The quasi-standard statechart semantics are documented and implemented in the STATEMATE model and tool (Harel and Naamad 1996), (Harel, Lanchover, Naamad, Pnueli, Politi, Sherman, Shtull-Trauring, and Trakhtenbrot 1990). Apart from STATEMATE, numerous other variations exist (von der Beeck 1994), which is to some extent a consequence of the fact that a detailed description of statechart semantics was only provided years after the initial proposal. Today, the most widely used variants are *UML 2* (Object Management Group 2009) state machines, the W3C *State Chart XML* (SCXML) (W3C 2010) model and the commercial *Simulink Stateflow* tool of the MathWorks (MathWorks).

4.3.3 OMG UML State Machines

UML state machines (Object Management Group 2011) are derived from Harel statecharts and introduce various syntactic and semantic extensions with the goal to enable application in an object oriented context. One limitation of UML in general are the so-called *semantic variation points* that intentionally leave the precise semantics of model entities open; for example, the order in which events are dequeued from the internal event queue is left undefined. This is to give tool-implementers of the standard more liberty to define these semantics themselves. As a consequence, practically all implementations of UML state machines yield incompatible execution behaviour, *and* the precise semantics often remain hidden in the *implementation* of the tool.

A second issue with UML statecharts arises from semantic deviations made from Harel statecharts with the goal to make UML statecharts behave more like objects as in the object-oriented programming paradigm; the consequences are discussed in Section 4.6.

In spite of these limitations, UML state machines are widely used and supported by many modeling tools. Hence, the statecharts proposed in this work follow these semantics wherever appropriate, in order to facilitate automatic transformation of UML state machines to rFSM models.

4.3.4 Statecharts in Robotics

Statecharts have been previously used in robotic systems. Merz et. al. (Merz, Rudol, and Wzorek 2006) use an augmented flavor of Statecharts called Extended State Machines (ESM) to specify control and data flow in a robotic control framework. In contrast to our approach that advocates the separation of

the concerns of computation and coordination, ESM states tightly couple control and data-flow by permitting these to also output data via data ports. While this combination might be convenient, reusability is reduced by tightly coupling application coordination with functional computations. Billington et. al. (Billington, Estivill-Castro, Hexel, and Rock 2010) propose an approach to requirement engineering using UML Statemachines that are extended with non-monotonic logic for describing domain knowledge. This logic is used to describe the behavior of single FSM and the corresponding predicates to be used as guard conditions. This formulation then allows to validate certain properties such as the exclusivity of guards. Similarly to our approach, the behavior of the system is defined by multiple interacting state machines, though the focus is on modeling complex high-level behavioral protocols. In contrast, our work is concerned with preserving reusability of computational components.

4.3.5 IEC 61131-3 Sequential Function Charts

Sequential function charts (SFC) are a PLC programming language specified in the IEC 61131-3 (International Electrotechnical Commission 2003) standard. The SFC language is based on the Grafcet language (David and Alla 1992), which in turn is derived from Petri nets. This heritage reflects in the SFC semantics that generally impose less constraints with respect to hierarchies and permitted transitions.

SFCs are constructed by linking *steps* with *transitions*. Each transition has a *condition* that defines when control passes to the next step. *Simultaneous divergence* permits to activate two successor states simultaneously while *simultaneous convergence* permits to synchronize the parallel execution again. In order to specify the in-step behaviour, steps can be associated with *Action blocks*. *Action qualifiers* define the exact manner in which the action is executed. Example qualifiers are `N` (non-stored) for executing an action as long as the state is active, `S` (stored) for executing an action permanently until disabled by the `R` (reset) qualifier, `P` (pulse) executing an action only when entering and/or leaving a state, or `L` and `D` for limiting or delaying the execution of the action to/for a certain time period. SFCs support hierarchical decomposition by recursive specification of an action with a SFC (or any other IEC diagram).

Bauer and Engell (Bauer and Engell 2002) compare SFC to statechart semantics. Summarised, the major differences are:

- Although SFC support hierarchical steps by recursively specifying an action with a SFC, none of the constraints of statecharts are enforced. This is because there is no transitively upwards closed active state list. In

contrast, if a hierarchically nested state is active in a statechart, then so is the parent state. For SFC it is possible that a parent step is deactivated but the child step remains active.

- In contrast to statecharts, SFC do not permit to specify inter-level transitions.
- A statechart will behave non-deterministically if *conflicting transitions* are unable to be resolved by means of the conflict resolution mechanisms. If conflicting transitions are found in a SFC then *all* transitions in question will be enabled.[2]
- IEC 61131-3 does not define the order of action execution and transition evaluation for SFCs, however this ordering may influence the overall behaviour.
- In statecharts, higher priority transitions can prevent lower-level transitions from triggering. Whether this is possible in SFC depends on the respective implementation of hierarchy.

Bauer and Engell conclude by suggesting that a combination of statecharts (for specifying the high level operational modes and safety aspects) and Sequential Function Charts (for defining the sequences of lower-level computations), would yield the benefits of both models. A possible execution semantics for the combined model is proposed.

4.3.6 Behavior Trees

Behaviour trees (BT) (Dromey 2003) are a graphical language to support the process of behavior engineering. The BT language consists of different types of states that are organized in a tree form to express behavior. For example, states can represent conditions (`IF`, `WHEN`), dataflow (`Data-out`) or state (`System`, `Internal`). The behavior engineering process consists of several steps, each producing a new behavioral tree. Firstly, individual functional requirements are modelled using requirements behavioral trees (RBT). Next, these requirements are integrated into the design behavioral tree (DBT) that composes all requirements. Using the DBT, the component interaction network (CIN) and the component's behavior tree (CBT) are derived. The first describes which components interact and the seconds models each components behavior.

[2] According to IEC61131-3, pg. 100, this is an error.

4.3.7 The Task Description Language

The Task Description Language (TDL), introduced by (Simmons and Apfelbaum 1998), is a language to describe Robot Tasks. TDL is based on the task tree datastructure, whose nodes can contain commands, goals, monitors or exceptions. Goals describe higher level tasks whos children may be goals or commands. Monitors are invoked repeatedly to validate certain conditions; exceptions can be used to signal and handle erronous conditions. When a goal node is expanded, new children (goals or commands) are added to the goal itself, thus defining what needs to be handled before the goal completes. This way, the tree datastructure implies an ordering between execution of parent and child nodes. In contrast, sibling nodes are executed concurrently unless ordering constraints are imposed using one of the synchronization mechanisms. A TDL program is a C++ program with additional syntax that is transformed to pure C++ by a dedicated compiler.

4.3.8 The Urbiscript Language

The *urbiscript* language (Baillie 2004) by GOSTAI is a multi-paradigm scripting language targeted towards the domain of robotics. To that end, it provides syntax to specify concurrent execution of statements and primitives to supporting event driven programming. The GOSTAI Studio IDE supports development of hierarchical state machines that can be transformed to urbiscript. However, as of today no information about the semantics of this state machine formalism is publicly available; with the exception that, according to GOSTAI, the semantics are close to a subset of UML 2.

4.3.9 Simulink Stateflow

Stateflow (MathWorks) is a synchronous statechart extension to the Matlab/Simulink environment into which it is tightly integrated. Stateflow offers an even larger set of primitives than statecharts: *Condition actions* are executed if a guard is true, even if the transition itself is not taken, *inner transitions* emanate from the inner boundary of a composite state and are similar to self-transitions but do not result in exiting and entering the source state.

Apart from these convenience features, major differences to Harel statecharts are that Stateflow only supports processing of one event at a time and does not transition based on more than one event.

Stateflow models can be configured in great detail, for instance to define transition priorities on the chart or state level or to define if supersteps may occur or not. This permits fine grained configuration and optimization, though at the price of compromising compatibility with other implementations, and reuse in other application contexts.

4.3.10 Statecharts in Modelica

Modelica (Modelica Association) is a language and toolchain for modeling and simulation of physical systems. Modelica offers support for hierarchical state machines in form of the StateGraph2 package (Otter, Malmheden, Elmqvist, Mattson, and Johnsson 2009). These state machines are derived from the Graphcet language (David and Alla 1992). Pohlmann et al. (Pohlmann, Dziwok, Suck, Wolf, Loh, and Tichy 2012) extend StateGraph2 with support for modeling asynchronous and synchronous communication as well as with real-time constraints. An overview of the different state-based models available in Modelica can be found in (Elmqvist, Gaucher, Mattsson, and Dupont 2012).

4.3.11 ROS SMACH

ROS SMACH (Bohren and Cousins 2010; Willow Garage 2008) is a Python library for specifying models of robotic behaviour. States encapsulate computations that are executed while a state is active. Depending on the result of a computation, a state is exited via a so called *state outcome*. These outcomes are used to model different accept conditions such as sucess or failure. Additionally, states may specify *userdata* that is either required by a state to do processing or provided as an output of the computation of the state. When specifying a SMACH state machine the connections between input and output userdata can be defined. SMACH supports parallel execution of state behaviours as well as controlled preemption.

Nevertheless, in contrast to its name the SMACH library has actually more in common with a flowchart than with a classical state machine: (i) states do not represent conditions of the system but rather processing steps, and (ii) SMACH does not foresee a mechanism to *react* to external events.

4.3.12 Conclusion of Literature Review

Based on the previous analysis, we have selected Statecharts as a starting point for deriving a minimal coordination mechanism suitable for the robotics

domain. This choice was based on the following requirements. To support reuse of coordination, the model must be *composable* and permit recombination of existing models within each other. Since the Statechart model allows states to contain states this is easily possible. It is worth noting that the reuse potential of statecharts has also been recognized by other domains such as Game AI development (Dragert, Kienzle, and Verbrugge 2011) or for building reusable webservices (Gamha, Bennacer, Ben Romdhane, Vidal-Naquet, and Ayeb 2007).

Moreover, compositionality is of primary concern, since the goal is to support construction of complex models by combining simple ones while maintaining predictability of the global system behaviour. *Compositional robustness* is closely related to compositionality; we define this property as follows: a system which is constructed from elementary parts is compositionally robust if it behaves robustly as a whole under changes or errors in individual parts. This characteristic is given particular attention in the following, because robotic coordination can often be conveniently modelled as a composition of multiple, heterogeneous layers of abstraction.

The ability to satisfy real-time constraints is a fundamental requirement for robot coordination related to aspects such as motion control or safety mechanisms. This does not only involve guaranteeing deterministic timing behaviour but also the ability to provide introspective information about the temporal behaviour. On the other hand, building complex, multi-robot systems requires distribution of such local real-time safe coordination over unreliable networks. As a consequence, coordination must be robust under (event) communication failures or varying latency. For example, reordering or loss of events may never lead to a coordination dead- or live-lock. Structural priorities and time-events facilitate this.

Lastly, statecharts and its variants are widely known and many developers are familiar with these. As this chapter aims at providing a minimal subset of the statechart model, the reuse of existing tools becomes possible.

For our purposes, the main alternatives to statecharts are behavioral trees or hierarchical petri-nets. Behavior trees can be composed in a similar way as statecharts, but offer a richer model and require more rigorous commitment to the approach, compared to this work's approach of superposing a statecharts to coordinate computational components. Moreoever, statecharts are better suited for modeling reactive and preemptable behavior by using non-local state transitions (i.e. a top level transition causing the deactivation of an entire sub-branch) taking into account the proper invocation of associated exit and entry functions. Similar limitations apply to hierarchical petri-nets (HPN) or variations of these as SFCs. Moreoever, the implementation of concurrency in Petri-nets requires making similar assumptions as those which we choose to

avoid by excluding parallel states in rFSM. Furthermore, petri-nets abstract away communication (e.g. for synchronization) between concurrent threads of execution. Thus, especially distributed Petri-Nets would require making additional assumptions on communication properties, which we can avoid for rFSM. Lastly, statecharts are, to some extent, also applicable to modeling of sequences (such as assembly tasks) that are traditionally the domain of petri-nets.

4.4 The rFSM Model

The rFSM model is a minimal subset of UML 2 and Harel statecharts consisting of only three model elements: **states**, **transitions** and **connectors**. Additionally, a virtual model element named *node* is introduced to simplify explanations concerning both states and connectors. Figure 4.3 shows the structural model as an Ecore diagram.

States can be composite or leaf states, depending on whether they contain child nodes or not. For example in Figure 4.4, the outer rounded rectangle (labelled `root`) is a composite state; the rounded rectangles `opening`, `grasping`, and `closing` are leaf states of the tree formed by the hierarchical composition of states; the arrows are transitions; and the filled black circle is an initial connector.

Distinguishing between leaf and composite states is important for modeling additional constraints that must be satisfied by valid rFSM models. For example, an important constraint that simplifies validation and execution semantics is, that, for an active, well-formed rFSM statechart, exactly one leaf state must be active (unless during transitioning, where it may be zero). Moreover, this constraint permits representing the entire active state of a rFSM statechart by means of a single leaf state. This is owned to the constraint of the statechart semantics, requiring that for any active state all parent states are active too.

At the top-level, a rFSM model is always contained within a state; that way any state machine model is inherently composable within other state machines. This composability is essential to enable the reuse of Coordination functionality represented by rFSM models.

Transitions connect nodes in a directed way, and may define a side-effect free, Boolean-valued *guard* function (whose result determines whether the transition is enabled or not), as well as an *effect* function (invoked when transitioning). Note that defining a specific *trigger* language in the core model is avoided by deferring the responsibility of determining the triggering of a transition to the

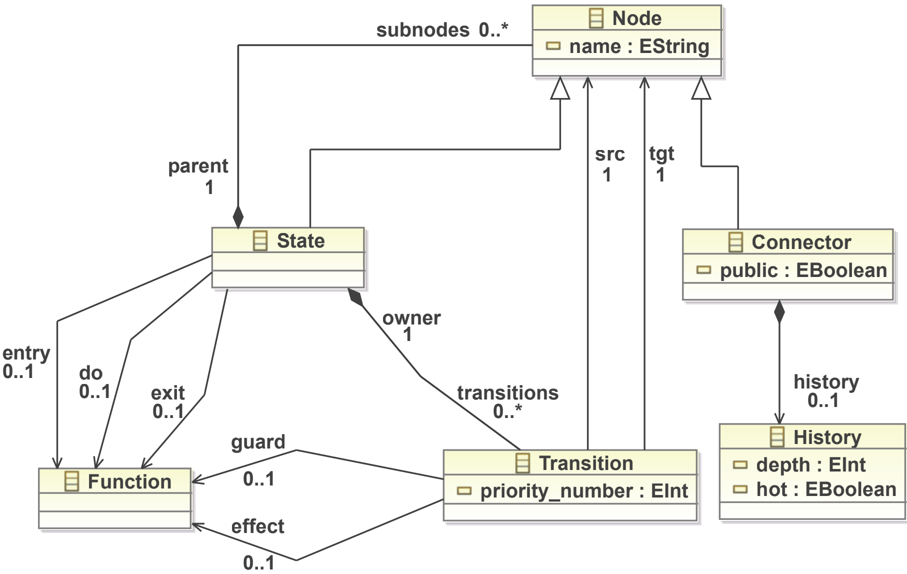

Figure 4.3: Complete rFSM Ecore model.

guard function. The latter can then be extended (in an automated way, e.g., by a plugin) to validate any event specification specified on a transition against the current set of events.[3]

Transitions are owned by composite states and *not*, as often assumed, by the states from which they originate (more explanation is given in Section 4.5.2). As this constraint is not expressed by the Ecore model, the OCL constraint shown in Listing 4.1 is added. The formulation assumes a function `LCA` to compute the least common ancestor and a predicate `ancestor` to check whether the first argument is an ancestor of the second.

```
context: Transition inv:
   let lca : State = LCA(self.src, self.tgt) in
      self.owner=lca or ancestor(self.owner, lca)
```

Listing 4.1: OCL constraint on transition ownership.

Following UML, rFSM permits states to be associated with behaviours that are executed at different points in time. These behaviours are the `entry` and `exit` actions that are executed upon entering and leaving a state respectively,

[3]Throughout the chapter we assume a basic trigger language that permits specifying a list of events of which each triggers the respective transition.

and the in-state `do` activity. While actions are generally short, activities are composed of atomic actions themselves and can run over longer periods, during which they are interruptible at the granularity of actions. Hence, to simplify execution semantics, the `do` activity is restricted to leaf states only. Moreover, rFSM states may define *internal transitions*, that permit reacting to events by executing an effect, though in contrast to regular transitions *without* leaving the state. In the rFSM model, all behavior is modelled as opaque functions.

Connectors can be used for constructing composite transitions by interconnecting two or more elementary ones. When such a transition is taken, the scope of connectors is honoured, permitting to define exactly which states are exited and entered throughout the composite transition. Connectors have multiple uses: for example they can be used to define the interface of a composite state by providing different entry or exit points. The initial connector has special semantics: when a transition ending on the boundary of a composite state is executed, the execution will continue with the transition emanating from the initial connector. To avoid stuck transitions, a constraint is introduced to enforce that each composite state that is target of a transition also defines an initial connector. This is expressed by the first OCL constraint in Listing 4.2; the second enforces that a composite state may define at most one initial connector.

```
context:
   State inv: self.subnodes.size>0 and
      Transition.allInstances()->select(t | t.tgt=self) implies
         (self.subnodes()->select(c | c.isTypeOf(Connector) and c.name="
              initial"))->size() = 1

   State inv: self.subnodes.size>0 implies
      (self.subnodes()->select(c | c.isTypeOf(Connector) and c.name="initial
           "))->size() <= 1
```

Listing 4.2: OCL constraint on existence of initial states.

As with states, multiple transitions may emanate from a connector. This permits to implement *dispatching* transitions that are triggered by two or more events and that dispatch to different states (see Section 4.5.3 for an example). The rFSM connector model element unifies the four very similar UML model elements junction, initial, entry- and exit pseudostates.

A state-machine is entered for the first time by transitioning via the transition emanating from the initial connector of the root state, resulting in the target state of this transition to be entered.

The elementary way to advance a rFSM state machine is to call the *step* function on it. This function retrieves *all* events that accumulated since the last step and attempts to find an enabled transition. This process starts top-down, from the root composite state down to the active leaf state. The rFSM semantics

require that, as soon as an enabled transition is found, the searching terminates and the transition is executed.

This approach of identifying the next transition has the advantage that it assigns explicit priorities to transitions, so-called *structural priorities* (Harel and Naamad 1996). The higher the source state of a transition is located in the state machine's tree, the higher the priority of the transition. The priority is visible in the state graph: given a set of events, the current active states, and the value of the guard predicates, it is immediately visible which transition will be selected. This follows the approach chosen for the STATEMATE semantics. Furthermore, through structural priority conflicts between transitions are largely avoided, leaving only the possibility of local conflicts among transitions exiting the same state. These conflicts can be eliminated either by additional guard conditions, or by a mechanism such as priority numbers

The minimum requirement for events is to carry identity and hence to be comparable between (possibly distributed) connected state machines. The simplest approach that remains comprehensible to humans and that is real-time safe[4] is to use string events.

The rFSM model does *not* include a parallel state element. The reason for this is that this element requires making a large number of fundamental, platform-specific assumptions. These are, for instance, the *order* in which parallel states are entered and exited, which underlying *concurrency mechanisms* such as threads are being used to execute the state machine instances or what the *priorities* of different parallel regions are. Instead of parallelism, a loosely coupled approach of distributed state machines is used, that permits multiple rFSM instances to interact by means of the available communication middleware (Sec. 4.8).

Moreover, rFSM does not adopt the STATEMATE *execution time* requirement which enforces that changes occurring in step n can only be sensed in step $n+1$, because of the complexity involved to manage such delayed processing of effected changes. But also, more importantly, because of the impossibility of delaying changes in open environments, as it is inherently the case in robotics. rFSM also differs with respect to the *greediness property* of transition selection, by choosing a simpler *take first* approach to computing the set of transitions to execute. In contrast to STATEMATE, external and internal events are not distinguished.

The ability to simulate a statechart before execution is a useful feature that can facilitate early detection of errors. However, often subtle corner cases

[4] Obviously assuming that memory is appropriately pre-allocated for the longest possible event.

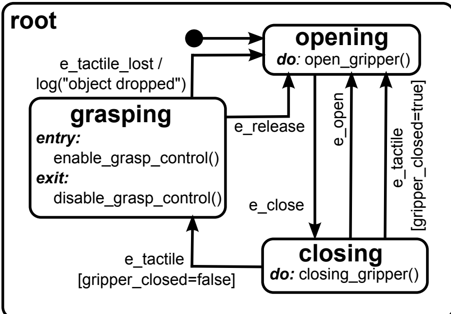

Figure 4.4: Coordinating a gripper.

remain in which simulation and real system behaviour differ, as is the case for STATEMATE. The rFSM reference implementation avoids this problem by unifying simulation and real system in form of an *executable model.* Obviously, this is possible only because the rFSM model targets the domain of coordination (in software) and need not be synthesised to targets such as VHDL (Very High Speed Integrated Circuit Hardware Description Language).

For convenience, we use the graphical notation of UML to visualise rFSM statecharts throughout the chapter. The only deviation is necessary to distinguish between initial and non-initial connectors: the first are (as in UML) shown as filled circles while the latter are depicted as empty circles. Public connectors must be drawn on the border of composite states.

4.5 Structural Semantics

This Section discusses in detail the implications of different semantics of finite state machine model elements. Since the rFSM model is derived from Harel statecharts and UML state machines, the discussion is based upon these standards. Other models are included in the discussion where appropriate.

4.5.1 Fundamental State Machine Elements

The fundamental elements of UML state machines are introduced using an example of a gripper coordination FSM shown in Figure 4.4. The required behavior is the following. Initially the gripper shall be opened. After receiving a request `e_close` the gripper shall start to close. The closing will be aborted and the gripper reopened if the request `e_open` is received. If a successfull

grasp is detected, the grasp control loop shall be activated and remain active until either the object is dropped (represented by `e_tactile_lost`) or releasing the object is requested (`e_release`). If the grasp fails, the gripper shall be reopened in preparation of a new attempt.

The required behavior is realized as follows. Three states `opening`, `closing` and `grasping` are introduced, each representing a distinguishable and exclusive state of the gripper. The filled circle is the *initial pseudo-state* and together with the transition to `opening` ensures that the latter will be the first state entered. Transitions connect states in a directed manner and carry a label of the following form:

```
trigger [guard condition] / transition effect
```

A trigger (typically one or more events) enables a transition between a source and a target state if and only if the optional *guard condition* evaluates to true.

An example of a guard condition can be found in Figure 4.4 on the transition from state `closing` to `grasping` labelled with `e_tactile [gripper_closed = false]`. This expression defines a successful grasp as an event from the tactile sensor (`e_tactile)` *and* the condition that the gripper is not fully closed. Likewise, a failed grasp is detected by a tactile event and a fully closed gripper; this triggers the transition back to `opening`.

Guard conditions constrain transitions and often permit reducing the amount of states necessary. On the other hand, guards also hide state (hidden by not being represented by a state model element) and can thus lead to conflicting transitions if two guards are simultaneously true. Therefore, it is considered a best practice to define these such that the exclusive disjunction of all guard conditions is true at any time.

A *transition effect* is an action that is executed during a transition between states. In Figure 4.4 the transition triggered by event `e_tactile_lost` from state `grasping` to `opening` uses the effect for logging a message that the grasped object has been dropped. Effects are necessary to define actions that are specific to transitions and not to source or target states. For the grasping example, the log message is only required when the transition to `opening` takes place as a result of unintentionally dropping the grasped object (triggered by `e_tactile_lost`), and not when the object is deliberately released (`e_release`).

The Matlab Stateflow model (MathWorks) includes a *condition action*, that is already executed when the guard condition evaluates to true, but before the transition as a whole is enabled. That way, condition actions behave similar

to transitions connected by UML Choice pseudo-states (describes in Section 4.5.3), and are not included in the rFSM model for the same reason.

Actions can be associated with states too. The *entry* and *exit* actions are executed when a state is entered or exited respectively. For instance, when the state `grasping` is entered, force control is enabled and disabled when the state is left. Like transition effects, entry and exit actions are always executed atomically as part of a transition and are never interrupted. In contrast, a state's *do* activity is executed as long as the state is active. In contrast to entry and exit actions, UML defines that (but not how!) the `do` activity can be interrupted if an event occurs that triggers an outgoing transition. For the example given in Figure 4.4, in state `closing` the gripper motors are continuously stepped by the `closing_gripper()` activity until the gripper is fully closed. Executing this operation in the do-activity permits it to be interrupted by a request to re-open the gripper or an event from the tactile sensors.

It is important to note that for robotic coordination this interruptibility can not be achieved by forced preemption, as for instance used to preempt operating system threads. This is because in systems interacting with real hardware some code paths must be treated as atomic in order to avoid non-deterministic and potentially dangerous behaviour.

For achieving safe preemption of the `do` activity, rFSM adopts the idea of GenoM *codels* (Fleury, Herrb, and Chatila 1997). A codel is the smallest non-interruptible unit of execution. Codels can be composed into larger computations that are interruptible at the granularity of the individual codel. By defining the `do` activity in terms of codels, its execution can be safely interrupted. The worst case latency for exiting the `do` program is defined by the execution time of the longest codel. The rFSM reference implementation realises this behaviour by implementing `do` using coroutines (Conway 1963).

The last fundamental primitive is the *internal transition*. The behaviour of this transition type is best explained by contrasting it to a self-transition, which is a regular transition starting and ending on the same state. When the latter transitions, the state is exited and entered, including invocation of the respective `exit` and `entry` actions. In contrast, when an internal transition is taken, the defining state is *not* exited. Consequently, the only action invoked is the internal transitions effect. Syntactically, internal transitions are drawn within a state and otherwise have the same form
`trigger [guard] / effect` as regular ones.

Interestingly, this seemingly simple model element already illustrates the issue of implicit semantic variation points in UML. For instance, it is undefined whether, in case of conflicts, internal or external transitions take priority. Furthermore,

it is unclear if and how an internal transition affects a running `do` activity. Does it cause the `do` activity to restart or does the internal transition effect run interleaved?

According to Simons (Simons 2000), internal transitions are redundant since all behaviour associated with a state can be modelled using a substate. This is however, only true for an internal transition defined within a leaf-state. Internal transitions defined within non-leaf composite states can not be converted to self-transitions, since the execution of the effect action implies exiting and entering of all child states. In practice, a typical use-case for internal transitions is to model *secondary mode switches*. For the grasping FSM of Figure 4.4, this could be to permit runtime switching between different controller configurations, but without otherwise interfering with the nominal grasping task. Such mode switches can be conveniently modelled using internal transitions defined at the root level.

With respect to variation points, the rFSM model assumes the following: regarding priorities, internal transitions are given equal priority to external ones unless made explicit by priority numbers. As to interference with the `do` activity, it is assumed that the internal transition execution takes place interleaved, though in a safe way honouring the atomicity of codels.

In summary, the rFSM model adopts the following fundamental state machine elements from the UML 2 standard: state, transition, internal transitions, event, entry, exit and transition effect actions, do activity and guard condition.

4.5.2 Hierarchical State Machines

Hierarchical state machines were first introduced by Harel (Harel 1987). The key idea is to permit nesting of states within states. This is illustrated by Figure 4.5 showing a model of a robot performing a force-controlled operation on a work piece. The actual operation is modelled using a nested state machine within the `operational` state. Such a state machine containing other states is also called a *composite state*, since it can be understood as the composition of multiple state machines. When the `operational` state is entered via the transition triggered by `e_range_clear`, it immediately continues entering the `approaching` state, since this is connected from the initial connector. As a consequence the work piece is approached until the event `e_contact` is received, that triggers the transition to the `in_contact` state, in which force control is enabled.

At the top level, the two states `safe_mode` and `operational` describe the basic behaviour of the system. The `safe_mode` state stops the robot immediately

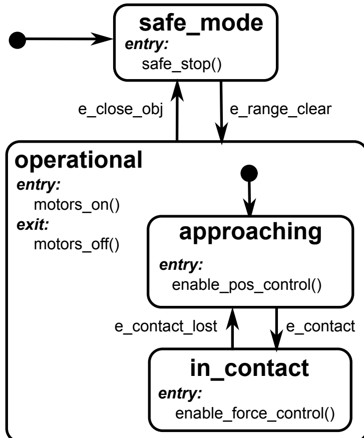

Figure 4.5: Hierarchical state machine.

and is entered when a sensor such as a laser scanner reports an object within the safety range. Note that the execution of this transition will result in exiting the `operational` state, no matter which substate is active. The event `e_range_clear` signals that the safety range is clear again and triggers the transition back to the `operational` state.

A fundamental property of hierarchical state machines is that multiple states can be active at the same time. For example, if the state `approaching` is active, then so is the parent state `operational`. In statecharts, the set of active states, also called the *active state configuration*, is always transitively closed with respect to the parent relation.

It is often helpful to visualise hierarchical state machines as a rooted tree. The root is formed by the top level composite state with nested states as children. This way, the active state configuration of any active hierarchical state machine can be represented by a single state (assuming the transitivity relationship from above).

Moreover, the tree structure facilitates unique identification of states within a hierarchical state machine by making use of the concept of a *fully qualified state name*. For instance `in_contact` becomes `root.operational.in_contact`. This scheme solves the problem of naming conflicts between identically named states in different composite states.

Hierarchical state machines serve multiple purposes. Firstly, composite states can be used as an abstraction mechanism and to modularise systems. For the sample model shown in Figure 4.5, the `operational` composite state can be

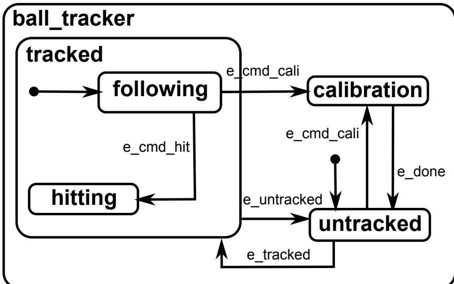

Figure 4.6: Boundary crossing.

developed and tested independently from the rest of the system.

Secondly, hierarchical states serve to express constraints in an efficient way. For the example above, one of the constraints is the following: the motors are only enabled while in the operational state. This is enforced by the entry and exit actions of the `operational` composite state. Of course a non-hierarchical flat state machine can model the same behaviour. However, in that case it would be necessary to add transitions from all substates of `operational` to the state `safe_mode`, thereby preventing the internals of the operational state to be modelled separately from the safety measures.

Lastly, hierarchy is also used to define priority among groups of transitions. This is discussed in Section 4.6.3.

The rFSM model supports hierarchical state machines and requires that these can be composed within each other.

Boundary Crossing Transitions

A boundary crossing transition is defined as a transition crossing a composite state's boundary. An example is shown in Figure 4.6: the transition from state `following` to `calibration` crosses the boundary of the `tracked` state. There exists some controversy in literature about this type of transition. Simons (Simons 2000) strongly suggest to prohibit boundary crossing for the reason that such transitions violate encapsulation of the nested state. On the contrary, UML 2 and Harel statecharts permit and encourage boundary crossing. According to Simons, one motivation for permitting this type of transition in UML was to provide a way to signal multiple accept conditions of a composite state by directly adding transitions from within the composite state to the respective target states. Meanwhile, this issue has been addressed in UML

version 2.1 by introducing exit points. The latter model element provides an explicit way to specify multiple accept conditions of a composite state. The rFSM model supports exit points by means of the generic connector, as described in the following Section.

Nevertheless, an important use of boundary crossing is to support *structural priority*. This concept resolves many cases of conflicting transitions that would otherwise result in non-deterministic behaviour. Structural priority is discussed in more detail in Section 4.6.

Furthermore, taking a closer look at the ownership relation between states and transitions reveals that boundary crossing does not necessarily violate encapsulation as described by Simons. The frequently made, wrong assumption is that transitions are owned by their source state. Under that assumption the boundary crossing transition of the example indeed violates encapsulation of the `tracked` and `following` states, because it introduces a dependency on a state (`calibration`) not part of the composite. However, if this transition is instead owned by the state `ball_tracker`, which is parent of both `tracked` and `calibration`, the problem vanishes. From this perspective transitions are viewed as layers added by surrounding composite states, thereby permitting to extend the enclosed behaviours without breaking their encapsulation. The recent version 2.3 of UML adds a paragraph making a similar observation (Object Management Group 2011, p. 583):

> The owner of a transition is not explicitly constrained [...]. A suggested owner of a transition is the LCA of the source and target vertices.

The least-common ancestor (LCA) of a transition is the deepest nested state which is a parent of both transition source and target. To ensure construction of modular and reusable statecharts the rFSM semantics impose a stronger constraint than UML: it is *required* that transitions are owned by no state less nested than the transition LCA. This ensures that no state can contain a transition with an unresolved reference to a state. As long as this constraint is satisfied, the rFSM model permits boundary crossing transitions.

The LCA concept also plays an important role for describing the execution semantics of hierarchical state machines (Sec. 4.6).

4.5.3 UML Pseudo-States

UML state machines introduce several types of so called Pseudo-States with special semantics.

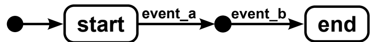

Figure 4.7: UML Junction.

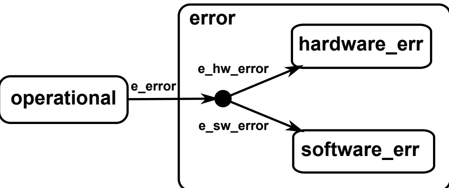

Figure 4.8: UML Junction with split.

Initial State

The UML 2 initial pseudo-state defines the sub-state of a composite state that is entered by default when a transition ending on the composite state is executed. UML permits only one transition to emerge from an initial state, that furthermore may not define a guard condition. Apart from these constraints, the initial state is semantically equivalent to the junction model element described below.

Junction and Choice

The junction (available both in UML and STATEMATE) is used to create *composite transitions* by chaining together multiple elementary transitions, as shown in Figure 4.7.

Composite transitions formed by junctions are evaluated statically, meaning that all elementary transitions up to the next state are checked *before* the composite transition is enabled. Only when the conjunction of all transition elements is enabled[5] the composite transition is too. Junctions permit to create splits and merges, that for instance are used in the *dispatcher* pattern illustrated in Figure 4.8.

This pattern employs a junction as the entry point of a composite state. Internal to the `error` state, further dispatching to specific handler states takes place using multiple outgoing transitions. It is the responsibility of the state machine designer to ensure that only one enabled path is found. Otherwise the state

[5] A transition is enabled when triggered by the current events and the guard condition evaluates to true.

machine will make an arbitrary choice and thus become non-deterministic. Mechanisms to avoid conflicts are discussed in Section 4.6.3.

UML Choices are similar to junctions, differing only by being *dynamically* evaluated by the transition logic while the transition is already being executed. Thus, choices have the dangerous property that composite transitions can get stuck during execution if at some point none of the outgoing transitions are enabled. To avoid this, the UML standard advises to always include an *else* transition, that is automatically enabled if no other transition is true.

Entry and Exit Points

Entry and exit points were introduced in version 2.1 of the UML standard with the goal of permitting to define multiple ways to enter a composite state, or to exit it with different outcomes, respectively.

Examining the semantics of entry and exit points reveals that these are semantically almost identical to the junction element described above. According to (Object Management Group 2011, p. 551), only one outgoing transition is permitted from the entry point to a state, while a junction permits multiple. In our opinion, there is no need for this constraint. Hence the only remaining difference is concerned with the graphical representation of whether the circle is drawn on the border of the enclosing composite state or not.

The rFSM Connector: unifying Initial, Junction, Entry and Exit Pseudostates

Given the large semantical similarities between the UML 2.1 initial, junction, entry and exit pseudo-states, the rFSM model unifies these under the name *Connector*. This unification significantly reduces the amount of concepts, while still providing the same semantic expressivity. The name Connector follows the suggestion of Simons (Simons 2000), who pointed out that the term state (even with the "pseudo" prefix) is misleading. This is because the element in question does not represent a state (defined as a distinguishable condition of the system) but rather constitutes part of a transition.

As entry and exit points represent public interfaces to a state machine, the rFSM adds a Boolean-valued attribute `public` to the connector element. This serves two purposes: (i) it can be used by a model checker to issue warnings if a state machine designer adds transitions to non-public connectors from outside of a composite state, and (ii) this attribute may be used by graphical visualisation tools to draw public connectors according to the UML visual guidelines for entry

and exit points, which is on the border of a composite state. In the following we represent connectors as small circles: initial connectors are filled in black while all others are not filled. This avoids the ambiguity in UML resulting from both initial and junction pseudo-states being graphically depicted as black, filled circles. If this notation were used instead of UML, the middle junction in UML diagram 4.7 (connecting `start` with `end` would be not filled, since it is not an initial connector. The junction in diagram 4.8 would move to the border of `error` state and be likewise not filled, unless it actually is the initial state.

Because of the dangerous property of permitting *stuck* transitions, the UML choice pseudo-states is not included in the rFSM model.

Final State

Transitioning to a UML final state means that the enclosing state has completed. In contrast to the initial pseudo-state, the final state is not a pseudo-state at all, but a specialised regular state. The reason for this is not explained in the standard, however it can be assumed that this is because a well-formed UML state machine can actually *be* in the final state. In contrast, it is not possible to be in the initial state, since this is part of a transition. When a final state is entered, UML requires a *completion event* to be raised that may trigger transitions emanating from a parent state. If no transitions are enabled, the final state remains active.

The rFSM model does not introduce a special final state. Instead an empty, user-defined regular state named *final* can serve this purpose. When no `do` function is defined, the completion event is raised immediately after entry and can hence be used to trigger transitions emanating from parent states.

Major semantic differences exist between UML and rFSM with respect to the *completion event* itself; this is discussed in Section 4.7.4.

Terminate

The UML *Terminate* pseudo-state offers a mechanism to terminate the execution of the entire state machine instance by transitioning to this connector. No actions are executed except those associated with the transition. In our opinion, there is no benefit of using a Terminate model element instead of a regular state called `terminate` that executes the required shutdown procedure. Hence the rFSM model does not include a terminate pseudo-state.

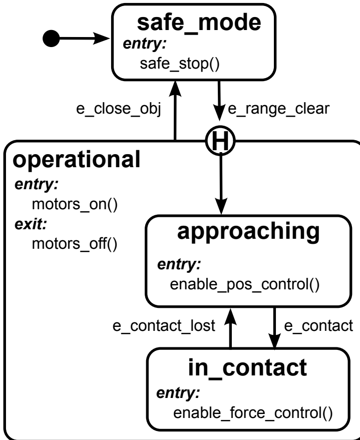

Figure 4.9: UML History states.

History States

The *History state*, graphically denoted by a circle around a capital **H**, is an initial connector with special semantics. Consider the example of Figure 4.5. After returning from the state `safe_mode` (due to an interrupt caused by receiving the `e_close_obj` event) the `operational` state is reentered. In this model the execution of `operational` will be restarted, by re-executing the transition from the initial to `approaching`. However, in some applications it is preferable to *resume* the execution instead of restarting it. For instance this could be the case if the force-controlled operation is to be performed only once and the robot was paused in a position ready to resume. To achieve this behaviour, the initial connector is replaced by a history connector, as illustrated in Figure 4.9.

The target of the `e_range_clear` transition is now a history connector of the `operational` state, from which a second transition is defined to the `approaching` state. The latter defines the *default* state to enter when a composite state is entered via a history connector for the first time. So far, the state machines shown in Figure 4.5 and 4.9 behave identically. However, on subsequent entries of operational through the history connector, the last active state configuration (i.e., the one active at the time when `operational` was exited) will be restored. This effectively results in resuming the previously interrupted task.

UML 2 defines two types of history states: *shallow* and *deep*. Transitions to the former result only in restoring the active configuration for states at the same level as the history connector within the composite state. In contrast,

transitions to the latter type result in restoring the complete state configuration including all substates.

History states permit (possibly recursively) resuming previously preempted activities. This pattern is very common in robotics, both at the lower, hard real-time level (such as for implementing safety mechanisms) and at higher levels to model reactive agent architectures.

The rFSM model includes support for both shallow and deep types of history connectors by including a generalised version offering a configurable depth parameter, that defines up to which substates the active configuration is restored. Additionally, taking advantage of the *Codel* based model of computation described in Section 4.5.1, a Boolean-valued `hot` attribute is introduced. A hot history connector will not only restore the active state configuration but also resume the execution of the `do` codels at the point where it was preempted. In contrast to UML, where a history connector always functions as an initial connector, rFSM permits defining these attributes separately.

Fork and Join

The UML fork and join pseudo-states are used to create and merge concurrent transitions in the context of parallel states: implicitly when entering and exiting parallel regions, and explicitly to synchronise different parallel regions. As discussed in detail in Section 4.8, the rFSM model does not include parallel states, and hence also does not require fork and join model elements.

4.5.4 State Machine Extension

UML 2 supports the concept of inheritance of state machines. A derived state machine may override different aspects of its super state machine. This way, inheritance permits defining *abstract* state machines[6] that function as a common interface for specialised versions. Inheritance may also improve reusability by facilitating redefinition of existing models to support new use-cases that otherwise would have required developing a new model.

In contrast to the previously discussed model elements, extension is a meta-level feature that provides an alternate way to specify state machines. It does not influence the semantics of concrete models. Therefore, rFSM does not include primitives to support the definition of derived state machines. This is because adding inheritance would pollute the core model with a feature that can be

[6] Not to be confused the Abstract State Machine (ASM) formal method.

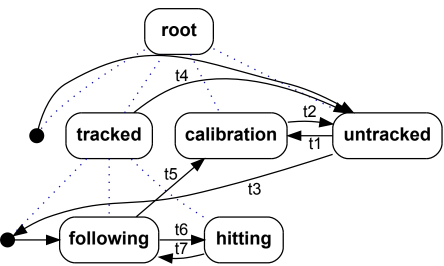

Figure 4.10: Scope of transitions.

easily added as an extension, for instance, as a *development tool* that generates a derived model from a super state machine and a list of overridden properties. Since the rFSM reference implementation uses the fundamental Lua table data-structure for specifying models, redefining, removing or adding elements is easily possible.

4.6 Execution Semantics

While the previous Section predominantly deals with *structural* aspects of state machine semantics, this Section focuses on the *behavioural* aspects, i.e., the execution semantics. To this end, some terminology is introduced. As explained in Section 4.5.2, the *active configuration* is the set of states that is active at given point in time. A *full transition* (Harel and Naamad 1996, p. 302), is a transition starting and ending on a state. A full transition can be either *simple* or *composite*, the latter consisting of a sequence of simple transitions joined together by connectors. The *scope of a transition* (Harel and Naamad 1996, p. 309) is defined as the lowest common ancestor of source and target vertices of a transition.[7] This is best illustrated using a state machine visualised as a tree as exemplified in Figure 4.10.

This Figure shows a similar FSM as in Figure 4.6. The dashed lines represent the containment relationship between composite and contained states, the arrows represent regular transitions. For instance the scope of the transition `t4` from `tracked` to `untracked` is `root`, as both source and target states are direct children. The scope of transition `t5` is also `root`, because the least common ancestor state is root.

[7]For state machines with parallel states the definition is extended to: lowest common non-parallel ancestor.

The scope of a transition is essential for statechart execution semantics because it determines how the active configuration changes when a transition is taken.

4.6.1 Fundamental Execution Semantics

The basic mode of operation of a state machine is to transition in response to events received. UML 2 describes the *run-to-completion* assumption that defines that a new event will only be processed *after* the processing of the current one has completed. This conforms to the STATEMATE semantics that require the following: *"Reactions to external and internal events [...] can be sensed only after completion of the step"* (Harel and Naamad 1996, p. 298). For both semantics, this implies that in order to react to new events, the state machine must be in a stable state; a stable state being defined as the execution of exit action, transition effect and entry action having completed.[8]

Beside these similarities, major differences exist with respect to how and which events are selected for computing a transition, how conflicts between simultaneously enabled transitions are avoided or resolved, and what the semantics of events are.

4.6.2 Event Selection

In UML 2 events are stored in a global queue from which they are removed one at a time. The standard does not specify the exact procedure (Object Management Group 2011, p. 574), in order to permit modeling of different priority-based schemes. After an event has been selected, it is used to compute the set of enabled transitions. If none are triggered, the event is discarded.

The classical statechart semantics differ from this by taking *all* events that occurred since the last transition into account for computing the set of enabled transitions. These events are discarded after the step.

4.6.3 Computing the Enabled Transition Set

Next, the selected event(s) are used to compute the set of enabled transitions given all transitions emanating from the current active configuration. With both UML and Harel semantics it is possible that the set of enabled transitions

[8] It is theoretically impossible to encounter a STATEMATE statechart during transition, since transitions are assumed to take zero time. This is the so-called *perfect synchronicity hypothesis*, which seldom holds in modern robotic systems.

contains more than one transition. Apart from certain special cases,[9] this condition signifies a conflict: executing both transitions would result in an invalid configuration of the state machine as multiple non-parallel states would become active simultaneously.

Because conflicts are likely to happen, it is necessary to resolve these by applying rules. The concept of *structural priority*, introduced by Harel (Harel and Naamad 1996, p. 328) largely resolves conflicts by assigning priorities to transitions based on their location in the state machine tree. The initial approach was to assign higher priorities the higher the position of the transition LCA was in the hierarchy of states. Later versions of STATEMATE simplified this by using the transition source state instead of the LCA. This priority rule solves most conflicts; the remaining can be explicitly solved by assigning so-called priority numbers to transitions or by using a domain specific mechanism realized as a guard condition.

UML state machines adopt Harel's approach of structural priority although the order of priorities is reversed. This means that transitions with deeper nested source states are assigned higher priority than less nested states. The reason for this deviation is not explained in the UML standard but presumably has its cause in the inherent object oriented focus of UML. Assigning higher priority to deeper nested states permits substates to be interpreted as specialisations that refine the behaviour of parent states.

4.6.4 Discussion

The different focus of UML state machines and Harel statecharts become apparent by examining both execution semantics. The UML semantics, being part of an Object Oriented modeling standard are more focused on introducing object oriented concepts in state machines. In contrast, the original Harel semantics are targeted towards building complex, reactive systems. Hence, it comes as no surprise that the classical STATEMATE semantics are often better suited to model complex, multi-robot systems.

The rFSM model adopts the classical statechart semantics of both event selection and computation of the enabled event set, because of the reasons discussed below.

Firstly, the behaviour of a classical statechart is largely predictable from the graphical model since the next transition to be executed depends only on the active configuration, the set of input events and the state of the guard conditions.

[9] Multiple transitions entering a parallel state are simultaneously enabled *without* being in conflict.

This supports developers that can rely on the graphical model to understand and predict the statechart behaviour. This is not as simple for an UML model, that requires a developer to have additional knowledge about the particular dequeuing implementation and about the currently deferred events (discussed in Section 4.7.7).

Secondly, taking all events into account for determining which transitions to execute has the advantage that the highest-priority events take effect while lower-priority events are ignored. Moreover, the queue of events is less likely to overflow by being emptied on each step. This is important for complex systems for which floods of events are possible.

Thirdly, the classical statechart approach of conflict resolution supports *compositional robustness*. The example in Figure 4.5 illustrates this. For instance, assume the developer of the nested state machine `in_contact` adds a self-transition for the event `e_close_obj`. This transition conflicts with the top-level transition from `operational` to `safe_mode`. In the Harel semantics this does not affect the behaviour of the state machine as a whole, since the (safety relevant) transition from `operational` to `safe_mode` has higher structural priority. The system is said to behave *compositionally robust* to a minor change (*minor* being defined in terms of depth of the changed state) because a small change results in little or no change of the system as a whole. However, in the UML semantics this change will alter the behaviour of the entire state machine because the new transition will take precedence and prevent transitioning to `safe_mode`. Hence a minor local change results in a large change at system level. For modeling complex and modular systems this is undesirable.

rFSM follows the STATEMATE approach of using the transition source depth to define priorities. Furthermore, the mechanism of priority numbers is adopted to resolve conflicts among transitions emanating from a single state.

4.6.5 Evaluating Composite Transitions

One important issue that is scarcely discussed for UML and STATEMATE semantics is the *extent* to which composite transitions are checked prior to being executed. Starting from the example shown in Figure 4.5 and assuming that `safe_mode` is active and the event `e_range_clear` is in the queue, the transition from `safe_mode` to `operational` is enabled. The question posed by this scenario is whether the state machine logic may already start executing the transition at this point, or if further checking is necessary. The imminent danger of starting execution is that after the first part of the transition has been executed, the second part from the initial connector to the `approaching` state is *not* enabled, hence resulting in the FSM getting stuck. UML 2.1 circumvents

this problem by prohibiting initial transitions to define guard conditions, and by introducing a semantic variation point (Object Management Group 2011, p. 560). The latter leaves open how to interpret transitions to composite states without initial connectors. It is then up to tool-implementers to decide if this is to be treated as an ill-formed state machine or as the intention to enter a composite state but none of its substates.

Regarding this issue, there are differences in STATEMATE between simulated and generated code (Harel and Naamad 1996, p. 303): while the simulator will not begin executing the transition, the generated code will, and hence get stuck when the continuation transition is not enabled.

The rFSM model deals with the above-mentioned UML variation point by defining a transition to a composite state without an initial connector as ill-formed. Secondly, *deep transition checking* is required, meaning that for a transition to be enabled the complete path until reaching a leaf state must be enabled, thus including zero to many transitions from initial connectors to states. This eliminates the possibility of transitions getting stuck during execution. Because the rFSM model treats initial connectors and junctions identically, no special logic is required. Moreover, the UML constraint that forbids initial transitions to define guard conditions becomes unnecessary and can be dropped.

4.6.6 Transition Execution

Once a transition is enabled, it is executed in the following sequence. Firstly, all source states up to, but excluding, the LCA are exited by invoking their exit functions. For each composite state of these states, the last active sub-state is stored in preparation of a potential re-entry via a history connector. As an example, for a transition between two states within the same composite state only the source state is exited, as the LCA *is* the composite state. In contrast, for transitions contained by different composite states, multiple states are exited.

Secondly, the transition effect is executed, followed by the third part of the transition execution in which the transition target state including its parent states are entered. If this target is a leaf state, the transition execution is completed. Otherwise, the transition execution is continued at the initial connector of the composite state until a leaf state is reached. In case of composite transitions this procedure is executed for each individual transition.

As an example, assume that the active state of the statechart in Figure 4.6 is `following`, and the event `e_cmd_cali` occurs. The LCA of this transition

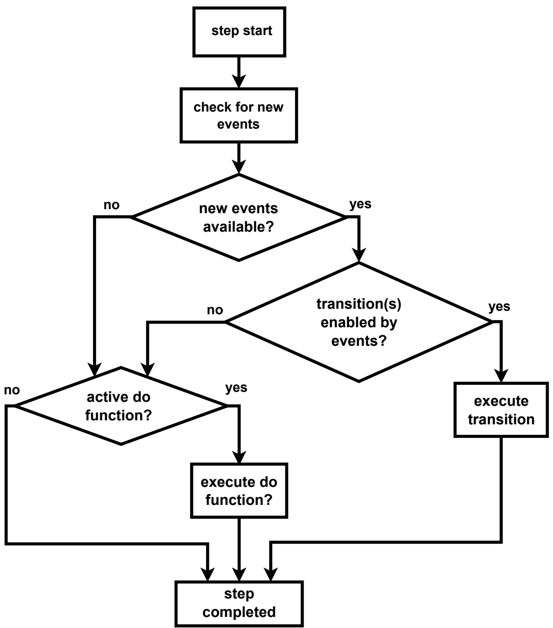

Figure 4.11: Flowchart of the step procedure.

is the root state (`ball_tracker`), hence first `following` and `tracked` will be exited. Next, the transition effect (not used in the example) will be executed. Lastly, the target states—for this example, only `calibration`—are entered.

4.6.7 rFSM Transition Semantics

Figure 4.11 summarises the process of executing a rFSM step. Executing a step will advance the state machine in an atomic way, leading to, at most, the execution of one composite transition.

Each step begins with the retrieval of all events that have accumulated since the last step. Next, if new events are available, it is checked whether these trigger any transitions. If yes, the one with the highest structural priority is selected and executed. If no events exist, the rFSM core checks if the currently active state has an enabled `do` function to execute. If yes, it is run. Otherwise the step is completed.

To complement `step`, a mechanism called `run` is provided: when a state machine is *run*, the step procedure is invoked either until no new events are available

and the active state has no enabled `do` function, *or* a maximum number of steps (given as an argument to run) have been executed. The practical application of step and run is described in detail in Section 4.10.

An active `do` function can be configured to be in one of two modes: idle and non-idle. Which one is appropriate is strongly influenced by the model of state machine progression (Sec. 4.10.1). If a `do` function is non-idle, the state machine engine will recall it immediately, provided that no events enabling outgoing transitions were received. This permits using the `do` function to carry out a computation as fast as possible while at the same time remaining reactive to incoming events. On the other hand, an idle `do` function is not recalled immediately, but only upon the next `step` invocation. This mode is usually appropriate when the `do` function is to be used to periodically carry out an action while the current state is active.

4.7 Event Semantics

Previous Sections described how events enable transitions and what are the ramifications thereof. This Section examines the semantics of events themselves, their structure and specialisations.

In classical statecharts, such as the STATEMATE semantics, events can be understood as messages which represent the occurrence of something. Events can be compared to events specified on transitions and are hence required to have identity. UML includes this basic event type using the name *SignalEvent*. However, there exist other, more specialised types of events which are discussed below.

4.7.1 UML ChangeEvent

A *ChangeEvent* is raised when a Boolean-valued condition evaluates to true. For this, the syntax `when <condition>` on a transition is used. In principle, such an event type seems useful for robotics coordination. The problem however, is that the UML specification leaves open when and how the condition is evaluated (Object Management Group 2011, p. 452). Obviously, for a hard real-time robotic system this is not acceptable, as the condition checking will directly influence the worst-case latency of the event generation and hence the transition execution. This situation can be remedied in two ways. Either the ChangeEvent is extended to take into account the additional semantics of evaluation frequency, timing precision etc., or this event type is excluded from the core semantics

and realized outside of the statechart. In the latter approach the validation is carried out by a separate component that raises the respective events. For rFSM the latter approach was chosen, as it does not require extensions to the core semantics and makes the involved assumptions explicit. Moreover, many computational components such as robot driver components, control components or estimators can easily provide such validation of internal conditions and the respective event generation with little computational overhead. Therefore, it is considered a best practice to extend computational components to raise events based on configurable constraints on their internal state.

4.7.2 UML Time Event

UML defines a *TimeEvent* as an event which is raised at a relative or absolute point in time. The syntax used is `after <duration>` or `at <time>`. For robotic applications TimeEvents can be useful in various circumstances, yet they suffer from a similar limitation as ChangeEvents, as no assumptions can be made on the qualitative properties of the time used. Hence, the variation point can be resolved likewise, by either extending the TimeEvent or by implementing it outside of the scope of the core semantics. rFSM takes the latter approach and does not include TimeEvents in the core semantics. Instead, TimeEvents are realized as a plugin that must be explicitely configured with a source of time. This way, the state machine developer is forced to consciously select a time source that provides the necessary quality of service.

It is worth pointing out the danger of misusing time events for modeling flowchart-like execution flow (Simons 2000). A state should reflect a distinguished condition of the system or a part thereof, not a time-bounded computation. The need for the latter is an indication to use a Flowchart formalism such as UML Activity Diagrams. Conversely, TimeEvents have important use-cases, most notably to model timeouts that trigger transitions to states for dealing with the absence of the nominally expected event.

4.7.3 UML Call Event

A CallEvent is an event defined for state machines used in the context of Object Oriented software systems. This event represents a request to invoke a certain method on an object. The CallEvent is generated after the operation has been invoked, thereby permitting the FSM to track the methods that are invoked. As the rFSM semantics do not make any assumptions about the underlying programming paradigm, no CallEvent is included.

4.7.4 UML Completion Event and Final State

According to UML, a *completion event* is raised either when the `do` behaviour of a simple state completes or a final state is entered. Graphically it is depicted by an *unlabelled transition* (called completion transition) emanating from a simple state or from a composite state that contains a final state.

The UML completion event has special semantics and is *implicitly* assigned highest priority among all events. For the UML semantics this is necessary, as a CompletionEvent does not carry any information about which state completed. Therefore, this event is only valid for the active configuration of the FSM at its time of generation. Queuing a completion event would permit the triggering of different unlabelled completion transitions from other states at a later time, which would be undesirable.

The rFSM model avoids this problem by adding identity in form of the fully qualified state name (Sec. 4.5.2) to the completion event. For instance, the completion event of the `in_contact` state of Figure 4.5 would be `e_done@root.operational.in_contact`. This way no special treatment of the completion event is necessary. Moreover, by *not* assigning special priority to this event, priorities are respected as a completion event must compete with all other events stored in the event queue. Finally, the readability of completion transitions is improved by labelling these explicitly. The semantics of unlabelled transitions are discussed below.

As in UML, the rFSM model requires that the completion event is raised either when the `do` function of a leaf state has completed, or immediately if no `do` is defined. The use-case of UML final states is covered in rFSM by using an empty, regular leaf state named *final*. Once this state is entered, the completion event `e_done@final` will be raised. Transitions can then be defined to react to this event.

4.7.5 UML AnyReceiveEvent and unlabeled Transitions

As the rFSM model (unlike UML) chooses to label completion transitions explicitly, the question of the semantics of unlabelled transition arises. There exist at least two natural interpretations of such a transition. The first is a transition that is never enabled, because no events are specified that could trigger it. The second, more useful interpretation is a transition that is enabled by any event. Harel statecharts opt for this interpretation because such *true-transitions* can be useful: when chaining multiple transitions by means of Connectors, true-transitions permit to avoid repeating the enumeration of the events specified

on the previous transitions for all subsequent transitions, thereby reducing redundancy. A second use-case is to avoid having to exhaustively enumerate all possible events for a transition that is only constrained by a guard condition. For UML state machines this interpretation can be realised by specifying the *AnyReceiveEvent* which is denoted by the keyword `any`. Because simpler and arguably more intuitive, the rFSM model adopts the Harel interpretation of the unlabelled transition as a transition being triggered by any event.

4.7.6 Edge- and Level-triggered Events

In the simplest case, an event represents the one-time occurrence of something. Based on all events that occurred since the last step, the FSM core reasons to find and execute a transition and afterwards drops these events. However, in some cases events are not only valid at a particular time instant, but persist during a period of time and/or until some action takes place. These two types of events are called edge and level triggered events, respectively.[10] The first are only valid at one point in time (on the rising or falling edge) while the second persist for some time, i.e., as long as the level remains *high*.

Level triggered events are often useful to signal a condition requiring some form of response. For example this could be a slight over-temperature condition of a motor during a manipulation task. Eventually, it is necessary to react to this event and reduce the overall speed, however this can take place after the current manipulation has completed. Moreoever, level-triggered events permit expressing the use case of deferred events, which are discussed below.

Level triggered events are quite similar to STATEMATE conditions (Harel and Naamad 1996). The main difference is that in STATEMATE a condition value is cached before each step while the rFSM semantics make no assumptions in this regard. The recommended way to realize a level triggered event in rFSM is by means of a guard function checking the condition in question. The details of how this condition is realized strongly depends on the underlying software framework in use.

4.7.7 Deferred Events

The UML standard permits a state to define a list of *deferred events*. When such a state is active, receiving a deferred event will not trigger any outgoing transitions. Instead this event remains in the queue until a state not deferring it becomes active. Since deeper nested states deferring events take priority

[10] In analogy to edge-triggered interrupts in operating systems.

over less nested states (Object Management Group 2011, p. 576), deferred events can be interpreted as an object-oriented specialisation mechanism which permits refining the behaviour of less nested states. Because rFSM adopts STATEMATE's priority approach to resolve conflicts, a deferred event mechanism would not violate the principle of compositional robustness. Nevertheless, the rFSM model does not include deferred events because (i) this primitive obscures the hierarchical conflict resolution, and (ii) the use-case of deferred events can be accommodated by a *level triggered* event. This is because a deferred event is effectively a level triggered event in disguise; it remains active (stored in the queue) until it is convenient to be processed. By using level triggered events, no special defer primitive is required and no additional, hidden state is introduced in form of a queue of deferred events.

4.8 Concurrency Semantics

State machine parallelism was introduced by Harel and is currently supported by most specifications, including UML, SCXML and Simulink Stateflow. This feature permits more than one substate or state machine contained in so called orthogonal regions to be active, and therefore the concurrent execution of their associated behaviors.

As robotic applications are inherently concurrent, the concept of parallel states is generally suitable for modeling these systems. Nevertheless, the weakly specified semantics of the parallel state element introduce several ambiguities. For instance, according to the UML standard (Object Management Group 2011, p. 575), no assumptions are made about the underlying thread environment, even though this greatly influences both performance and real-time properties of the executed model. Furthermore, the level at which concurrency is implemented is left undefined. In the simplest case, parallelism may only mean simultaneous execution of `do` behaviours, whereas more elaborate implementations might choose to implement concurrent entry of individual orthogonal regions. The latter would require introducing more assumptions: for instance it has to be decided which of the different regions is entered first and whether the entry is executed interleaved or sequentially.[11]

The analysis of existing coordination state machines shows that robotic coordination, unlike computation, rarely requires such tightly coupled concurrency. This is due to the fact that coordinative actions generally consist of issuing commands to lower level, concurrently running computation

[11] The latter example only holds true if multiple composite transitions emanate from the same fork.

components. Therefore, coordination does not benefit from parallelization in terms of performance as intensive computations do.

On the other hand, supporting distributed state machines is more important to robotics than the previously mentioned internal concurrency. This is particularly true for multi-robot applications. Distributed state machine instances must be able to observe each others' state and share some events. This approach has several advantages compared to parallel state machines: essentially, the complexity of parallel states and the associated assumptions on the threading environment can be avoided. Furthermore, the aspect of communication is moved out of the implementation and thereby made explicit, which is a best practice advocated in the 4Cs design paradigm. If, for instance, the communication between two FSMs breaks, an explicit event can be raised. The system architect can select a communication middleware suitable for the particular purpose. The downside is that additional effort is required to deploy such a distributed state machine. However, this process is generally suited to be automated by deployment *tools*.

A generic pattern of distributed sub-states is described in Section 4.10.5.

4.9 Reference Implementation

To illustrate the applicability and the exact semantics of the described model, a complete reference implementation of rFSM statecharts has been developed(Klotzbuecher 2011). As of today, this execution engine provides support for all mechanisms described in this chapter, apart from internal events and history connectors, which are expected to be added soon.

This rFSM engine is implemented as an *internal domain specific language* (DSL) (Fowler 2005) in the Lua programming language (Ierusalimschy, de Figueiredo, and Filho 1996). Internal DSL are built on top of an existing programming language, while external DSL are developed from scratch. By reusing existing infrastructure, internal DSLs are significantly easier to create and maintain than external ones. Moreover, internal DSLs can very easily be combined with programs of the host language or even with other DSLs. This extensibility is an important requirement for a state machine implementation, whose major purpose is to execute user defined actions according to the specified state machine model. The only disadvantage of an internal DSL is that its syntax is constrained by the host language. In practice this limitation is often acceptable given the reduction in development time and the simplicity gained for combining DSLs together.

Besides being suitable for building DSLs, the Lua language was chosen for the following reasons. Firstly, the language is designed to be both embeddable and extensible, which is reflected by the small memory footprint and straightforward foreign function interface. These properties are important since distributed coordination implies multiple Lua instances executing state machine instances embedded within components of a robotic software framework. Secondly, Lua offers a simple syntax that facilitates less experienced programmers not familiar with C++ or Java to build state machines; nevertheless, the language is mature and semantically well grounded by being strongly influenced by the Scheme language (Ierusalimschy, de Figueiredo, and Celes 2007). Moreover, the use of a scripting language contributes to the robustness of a system, because scripts, in contrast to C/C++, can not easily crash a process and thereby bring down unrelated computations executed in sibling threads. This property is essential for the aspect of coordination, that, as a system level concern, has higher robustness requirements than regular functional computations.

The following Listing shows the textual input model corresponding to the model of Figure 4.2.

```
return state{
   following = state{},

   paused = state{
      entry=function() ArmController:stopArm() end,
      exit=function() ArmController:startArm() end
   },

   transition{ src='initial', tgt='following' },

   transition{ src='following', tgt='paused',
               events={ 'e_untracked' } },

   transition{ src='paused', tgt='following',
               events={ 'e_tracked' } },
}
```

Listing 4.3: Textual model of Ball-tracking FSM.

This model is a valid Lua program representing a tree of states constructed using the `table` data type. When loading this file, the Lua interpreter performs basic syntax checking and instantiates the data structures. Note that this requires no effort by the DSL developer; it comes for free for an internal DSL. Next, the rFSM engine carries out basic validation and transformation of the tree structure to a graph ready for efficient execution. The validation is very limited and only concerned with detecting common structural errors. Formal verification of Statecharts has been treated in literature (Borland 2003; Mikk, Lakhnech, Siegel, and Holzmann 1998; Zhao and Krogh 2006) and outside the scope of this work.

Composition of states is supported through the `rfsm.load` primitive. For instance, to reuse an existing `following` state machine defined in a file, line 2 of Listing 4.3 could be defined as follows:

```
following = rfsm.load("following_fsm.rfsm")
```

As illustrated by example 4.3, a state machine developer will use Lua functions to implement behaviour in form of `entry`, `do`,[12] `exit` and `effect` programs. Calls to low-level C/C++ are easily integrated by means of the foreign function interface, as described below for the OROCOS/RTT framework.

Hooks are the key mechanism to extend and embed rFSM statecharts by means of custom, user defined functions. The most important hook is `getevents`, that allows customizing where events are retrieved from. A `getevents` function is expected to return a list of events that occurred since the last invocation and will be called by the rFSM engine one or more times during the execution of a step. The example in Listing 4.9 shows a sample getevents hook to retrieve all new events from a port `events_in` in the context of the Orocos RTT framework. The second part illustrates how the FSM is customized with this hook.

```
function rtt_getevents()
   local ret = {}
   while true do
      local fs, event = events_in:read()
      if fs ~= 'NewData' then break end
      ret[#ret+1] = event
   end
   return ret
end

return state {
   getevents = rtt_getevents,
   following = state{},
   ...
}
```

Since retrieving events from ports is very common when using rFSM with Orocos RTT, the auxillary function `gen_read_events(port1, port2,...` is provided in the module `rfsm_rtt` to automatically generate a getevents hook to read all events from the given ports.

Other hooks include `pre_step_hook` and `post_step_hook` that are called before and after a step is executed respectively. These are lowlevel hooks mainly used by extensions, such the time-event plugin that checks for expired timers or the event memory plugin described in Section 4.10.4 to keep track of occurred events. The rfsm module-level `preproc` hook allows registering functions that

[12] In the reference implementation `do` is renamed to `doo` to avoid conflicts with the homonymous Lua keyword.

will be called at initalization time and can be used to preprocess or validate the rFSM model prior to execution. An example use is for transforming platform independent task models to rFSM hooks, as explained in Section 4.10.2.

4.9.1 Software Framework Integration

The reference implementation is implemented in pure Lua and has no dependencies whatsoever. This permits standalone use which is convenient for testing and debugging of statecharts. To use the implementation in the context of robotic software frameworks requires a plugin to make the primitives of these frameworks available within Lua. We have developed such bindings for the OROCOS Real Time Toolkit (RTT) (Soetens 2006) that permit interacting with Components, Services, Operations, Input- and Output Ports and data types. A RTT LuaComponent is an initially empty container into which Lua programs can be loaded. This approach permits treating Coordination components just as regular computational components that are configured to load Coordination statecharts. Event driven input ports can be used to trigger dormant components upon receiving events and output ports are used to emit events. Using the standard communication primitives of a framework avoids duplicating these mechanisms for the sake of Coordination.

4.9.2 Considerations for Hard Real-Time Execution

The RTT framework provides a hard real-time safe execution environment for components. Naturally, real-time requirements exists also at the Coordination level. Satisfying hard real-time constraints for interpreted, garbage collected languages poses several challenges. Firstly, the allocation of memory must take place in a temporally deterministic way, which is typically not guaranteed to be the case for the default memory allocators of general purpose operating systems such as Linux. Secondly, the recuperation of unused memory must take place in a way that does not interfere with the nominal execution.

Our approach achieves deterministic allocation by extending the Lua interpreter to use an O(1) memory allocator (Masmano, Ripoll, Balbastre, and Crespo 2008). To achieve deterministic recuperation in critical real-time paths, the Lua garbage collector is stopped and manually controlled. Further details can be found in (Klotzbuecher, Soetens, and Bruyninckx 2010) (Klotzbuecher and Bruyninckx 2011).

4.9.3 Representing Events

In the majority of examples and real applications we have represented events using strings. While the rFSM implementation permits any comparable type to be used, the string representation is convenient for humans and still reasonably fast. Provided that memory is preallocated correctly, string events can even be used in hard real-time. Nevertheless, depending on the application, significant overhead could be avoided by denoting events using numbers. To take advantage of the performance of the numeric representation and the readability of string events, the latter could be transformed to the former at load time using an rFSM pre-processing hook defined as a rFSM plugin.

4.10 Patterns and Best Practices

This Section discusses reoccurring patterns and best practices in robotic coordination and their realization using rFSM.

4.10.1 Models of State Machine Progression

An important decision a statechart designer needs to take is to define how and when a statechart is advanced. Harel (Harel and Naamad 1996) describes two basic approaches: the asynchronous and synchronous model (see Section 9, "Two models of time"). The synchronous model assumes that one step is executed every time step, thereby causing the state machine to react to events and changes that occurred since completion of the previous step. In contrast, the asynchronous model advances the state machine only upon receiving events and is typically configured to execute as long as events are available. These two models correspond to the paradigms of event-triggered (ET) and time-triggered (TT) systems (Kopetz 1993; Kopetz and Bauer 2003) and have received thorough treatment in literature. In summary, the TT architecture offers several advantages over the ET architecture, including exact predictability of temporal behavior and allowing for systematic formal verification of temporal properties. In contrast, ET-systems generally require substantial testing to ensure that deadlines are met. Unfortunately, open and uncertain environments common in robotics require significant effort to determine the necessary granulation of observation lattice and maximum execution times (Kopetz 1993).

While the rFSM model and reference implementation support both models, each model has different use-cases for which it is appropriate. The asynchronous model is best suited for coordination scenarios in which multiple components,

possibly running at different frequencies, are coordinated. To this end, a dedicated Coordination component with its own activity is introduced. When new events are received, the state machine component is woken up and run (via the `run` function) and permitted to execute until it goes idle (which happens if there are no events in the queue and there is no active `do` function). Input events can originate from various sources: user commands, error events from computational components or filtered and processed raw events. Outputs may consist of performing actions such as starting or stopping components, creating or destroying connections, invoking component services, or configuration of parameters.

The synchronous execution model is suitable for coordination that takes place at a fixed frequency. The typical use case is coordination of a single component. For each cycle of the computation, the `step` function of the FSM is invoked to advance the state machine. This results in at most one transition being executed, taking into account all events that occurred since the last step. In contrast to the asynchronous model, the coordination can be executed conveniently within the activity of the coordinated host component. For instance, a PID controller component can be decorated with a discrete task-aware coordinator that monitors and adjusts control parameters according to the current task state.

In practice, hybrid models combining both asynchronous and synchronous advancing of state machines have often proven to be useful. This behaviour can be achieved by connecting a periodic timer component in addition to other asynchronous event sources to the incoming event port of a coordination component. This way, the coordinator is guaranteed to wake up for processing at a minimum rate defined by the timer component. However, if events arrive in between timer events, the coordinator will react instantly.

Note that such timer events should be understood in the broadest possible sense of time: the events do not have to be emitted at fixed intervals of real-time, but may be raised according to a virtual task-specific clock. For instance, a virtual time could be derived from physical system properties such as the current velocity.

4.10.2 Defining Platform and Robot independent Coordination Models

For reusing coordination models on different robots and with different software frameworks, it is necessary to avoid introducing dependencies on these aspects. A simple way to achieve this is to encapsulate the platform specific functions

used by the FSM in modules. A supported platform must then provide implementations satisfying the required FSM API.

The downside of this approach is that platform independence is only achieved for the FSM, but not for the behavior hidden in the opaque functions. Platform independence *including* this aspect becomes possible if the behavior of states can be formally modelled in a platform independent way. To this end, the statechart model is specified such that states *reference* platform independent task models. By means of a plugin realized using a rFSM pre-processing hook, the platform independent task models are dynamically transformed to platform specific rFSM hook functions at initalization time. The following example illustrates this transformation for a statechart coordinating end-effector motions. The motion specifications are expressed in the task frame formalism, a hybrid force-velocity control robot programming formalism (Mason 1981; Bruyninckx and De Schutter 1996).

Listing 4.4 shows a rFSM statechart modeling a robot arm moving down using velocity control and aligning upon entering in contact. Motion models are specified in an external module `tff_motions` and not further described here. Examples of the TFF motion DSL can be found in (Klotzbuecher and Bruyninckx 2012). States reference task models using a keyword `task`. Introducing this is legal since rFSM is implemented as an *open model*, signifying that keywords not part of the rFSM model are ignored instead of treated as errors.

```
require "rfsm_tff"
require "tff_motions"

return state {
  move_down = state {
    task = tff_motions["move_down"],
  },
  push_down = state {
    task = tff_motions["push_down"],
  },
  transition{ src="initial", tgt="move_down" },
  transition{ src="move_down", tgt="push_down",
              events={"e_contact"} }
}
```

Listing 4.4: rFSM model referencing TFF motions models.

Prior to executing the statechart model, the referenced task models need to be transformed to standard rFSM hook functions. This is achieved using the `rfsm_tff` plugin, shown in Listing 4.5. This plugin installs `transform_tff` as a rfsm pre-processing hook for carrying out the transformation at initalization time (line 13). This function in turn uses the rFSM `mapfsm` higher-order function to invoke `tff2hooks` on all states of the FSM (line 10). For each state defining a task, `tff2hooks` generates a function (using `tff_rtt.gen_apply` in line 5), that

when called commands the TFF controller to apply the given TFF motion using the respective platform specific mechanism. This function is set as an entry function such that this motion will be executed upon entering the state. In this way, the abstract task model rFSM statechart is transformed to an executable, platform specific rFSM instance. Though functional, the example is intended to be illustrative; for instance the Orocos RTT specific `tff_rtt.gen_apply` function should not be hardcoded but become a parameter of the `rfsm_tff` module.

```
1   module("rfsm_tff")
2
3   function tff2hooks(s)
4      if tff.is_TFFMotion(s.task) then
5         s.entry  = tff_rtt.gen_apply(s.task)
6      end
7   end
8
9   function transform_tff(fsm)
10     rfsm.mapfsm(tff2hooks, fsm, rfsm.is_state)
11  end
12
13  rfsm.preproc[#rfsm.preproc+1] = transform_tff
```

Listing 4.5: Dynamical transformation of TFF models to rFSM hook functions using plugin.

The plugin approach permits easily adding further preprocessing or validation steps. For instance, an additional robot specific plugin could be used to ensure that the used forces and velocities are within the limits of the actual capabilities of the robot.

4.10.3 Best practice *Pure Coordination*

Most coordination models do not restrict the primitives that can be used in actions, as for instance rFSM does not constrain the `entry`, `do`, `exit`, and `effect` functions in any way. In contrast, a pure coordinator limits its side-effects to exclusively raising events and has the following advantages. Firstly, the reusability of coordination models is increased by drastically limiting dependencies on platform specific actions. Secondly, the blocking invocation of operations on functional computations is avoided, thereby improving the determinism of the Coordinator. Lastly, Coordinator robustness is increased by avoiding operations that might block indefinitely or crash, either of which may effectively render the coordinator inoperative.

To that end we propose splitting the *rich* coordinator component that executes actions itself into a *Pure Coordinator* and a *Configurator*. Although the

coordinator remains in charge of commanding and reacting, the execution of actions is deferred to the Configurator. The Configurator is configured with a set of configurations that it will apply upon receiving the corresponding event. This pattern, called *Coordinator–Configurator*, has been implemented as a Configurator domain specific language for the Orocos RTT framework. More detail on pattern and DSL can be found here (Klotzbuecher, Biggs, and Bruyninckx 2012).

A pure coordinator requires to be informed via events about relevant changes in system state. This can be achieved in two ways: on the one hand by introducing an explicit monitor component that is configured with constraint-event pairs. When a constraint is violated, the corresponding event is raised to inform the coordinator. The other approach is to extend computational components themselves to raise events when (configurable) constraints on their internal state are violated. In general, the latter approach should be preferred if the constraint is specific to the computation, as it avoids the need to communicate state to a monitor. However, if the constraint is application specific, the monitor component is preferrable, since it preserves reusability of the computational component. This trade-off is further elaborated in the step by step example given in Section 4.11.

4.10.4 Event Memory

The default behaviour of statecharts is to avoid any state apart from the currently active configuration; all events are discarded after the execution of a step. One the one hand, this improves the deterministic nature of statechart execution by avoiding hidden state (e.g. in the form of deferred events). Yet, on the other hand, event-only coordination is complicated, in particular for distributed statecharts. Consider the example shown in Figure 4.12. This coordinator must wait in state `init_subfsms` until it receives an event from each sub-FSM, signalling that initialisation has completed. The problem with this implementation is that the transition will not be enabled, unless both events are received at exactly the same step.

The proposed solution to this problem is to extend each state with memory of the events that were received while this state was active. This permits to check if (and how often) an event has occurred while the transition source state is active. The rFSM reference implementation provides an *event memory* extension in form of a plugin, that when loaded keeps track of this information. That way, the problem can be solved by reformulating the transition using a guard as show in Listing 4.6. Using the recorded event history, the guard

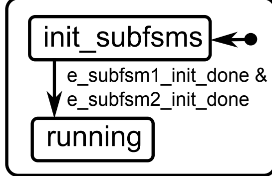

Figure 4.12: Dealing with inter-step history.

condition will inhibit the transition until both required sub-FSM events have been observed *while* residing in the source state of the transition.

```
guard=function (tr)
        return tr.src.emem.e_subfsm1_init_done > 1 and
               tr.src.emem.e_subfsm2_init_done > 1
      end
```

Listing 4.6: Event memory based guard condition.

Moreover, event memory can support detection of erroneous FSM behaviour, such as reception of too many of a particular event within a time step. A trigger to detect such conditions can check whether the number of a certain error event divided by the number of virtual time-events (Sec. 4.10.1) does not exceed a threshold.

4.10.5 Distributed Substates

The following paragraphs describe a generic pattern for implementing distributed statecharts. The approach is generic since it permits expressing concurrency at different levels of distribution, ranging from states distributed over a network to the traditional, closely coupled, thread-level parallelism.

The pattern is illustrated by the state machines in Figures 4.13 and 4.14, which are subsequently called the top and bottom half of the generic distribution mechanism.[13]

The first Figure shows a *container*, the top half of a distributed state. Its purpose is to activate and deactivate the (semantically) contained, concurrent substates on entry and exit respectively. Depending on the type of parallel state the entry function will carry out different actions: for a local thread-level distributed state it might spawn a new thread for each substate. For a parallel

[13]Apart from the name, the concept of top and bottom half has no relationship to the (obsolete) interrupt handling mechanism of the Linux Kernel.

distributed_state
entry:
activate_substates()
exit:
deactivate_substates()

Figure 4.13: Distributed state.

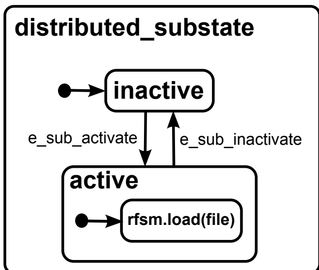

Figure 4.14: Distributed substate.

substate distributed over the network it might connect to a running and waiting instance. Substate activation and deactivation is achieved simply by sending the corresponding events to the bottom half.

Each distributed substate, the bottom half, is formed by a generic substate as shown in Figure 4.14. This state contains two states `active` and `inactive` that are connected by transitions. The active state *includes* the application specific state machine, that is entered per default via its initial connector once the `active` state is entered. This way, the top half can control the execution of the bottom half by sending it the events `e_sub_activate` and `e_sub_inactivate`. Likewise, by observing the active state of the bottom half the top half can determine when all substates have successfully been entered or exited.

Using this pattern formed by two FSM, any hierarchical state machine can be distributed independently of the form of distribution and obeying the rules of hierarchical statecharts. The only requirement is that events can be communicated between distributed instances.

Communication and Deployment By distributing state machines as described above, the communication between these instances is made explicit. To implement this, an asynchronous message passing mechanism (to communicate events) is most suitable. Communication is required for notifying a distributed state to activate or deactivate its sub-state machines, for propagating events between the two halves and for notifying the top half about state changes in

the bottom half. The latter is necessary, for example, in the exit function of the top half, which must wait for all substates to enter the `inactive` state after sending `e_sub_inactivate` event. This mechanism is similar to the `in(<state>)` *conditions* available in STATEMATE.

An important question is *which* events are communicated between event sources and sinks (statecharts, but also computational components). Generally, statecharts deal robustly with unused events by simply dropping these. Nevertheless, it is a good practice to limit the communicated events to the set of events that can trigger a peer statechart. Besides reducing the communication load, this avoids accidentally triggering unlabelled transitions and reduces the likeliness of event buffer overflows. In practice, this can be straightforwardly achieved by introducing two (or more) functions `raise_local(eventB)` and `raise(eventA)` that raise events locally or globally to a statechart, respectively.

Moreover, depending on the system it might be necessary to introduce buffers for storing events until they are retrieved by the statechart engine. Whether this is necessary or not depends on the worst-case number of events that can be raised simultaneously by all peer event sources, and the model of state machine progression (see Section 4.10.1). Complementary to this, *Complex Event Processing* techniques can help to reduce the number of events prior to being fed to a statechart.

A further, important issue concerns the reliability of the communication channel. In case of unreliable communication (such as UDP), event messages might get lost or be duplicated. Robust statecharts *can* deal with these issues by different means: for instance time triggered `timeout` transitions can re-raise events if the expected conditions are not met; likewise the for the receiving statechart non-nominal behaviour should be dealt with by *explicitly* defining transitions that trigger when expectations are violated.

4.10.6 Serialised Locally Distributed States

The most common motivation to use parallel (and hence also distributed) substates is to improve performance by parallelization. A less common use-case, nevertheless more relevant to the domain of coordination, is for combining behaviours that must be executed concurrently but that shall be kept separated for reasons of reusability (of each of the behaviours individually).

An example of this can be found in the coordination FSM of the iTaSC framework (De Schutter, De Laet, Rutgeerts, Decré, Smits, Aertbeliën, Claes, and Bruyninckx 2007), that requires composition of a generic (to the iTaSC framework) state machine with another state machine defined by the application

developer. The generic state machine first triggers all task-level FSMs to run at the right time. The second, user-defined FSM implements the overall task execution. Although both parts must always be executed after each other (first the generic, then the user defined) it is desirable to keep them specified separately to be able to reuse the generic part. To achieve this composition, a state named *serialised-locally-distributed state* (SLDS) was introduced. This name originates from the fact that the sub-states are advanced one by one in a serialised way in the same activity (thread) as the parent state machine. More concretely, a SLDS state can conveniently be realised as a leaf state that triggers all substates in its `do` function. This way, both FSMs can be composed from separate models at a late stage, during the loading of the SLDS state.

To concisely specify the behaviour of this type of state, additional properties must be defined: in which (partial) order shall the substates be advanced, shall this take place by invoking `step` (and how often?) or `run`? Shall non-idle sub-states result in the `do` of the SLDS to become non-idle too? As a side note, the considerable number of parameters required by the implementation of this most simple form of parallel state illustrates the complexity introduced by this family of model elements and confirms our approach to exclude these as primitives.

4.10.7 Discrete Preview Coordination

Discrete Preview Coordination is inspired by the concept of *preview control* (Sheridan 1992), but also by compiler branch-prediction techniques such as gcc's (GNU Compiler Collection) `__builtin_expect`. The basic idea is to exploit knowledge about the future behaviour of a system to optimise the current actions. In order to apply preview techniques to discrete coordination, we extend the core rFSM model as follows: transitions are extended with an optional Boolean `likely` attribute and states with an additional `prepare` action. While checking for enabled transitions from the currently active states, the preview mechanism will additionally check for transitions that are likely. If such a transition is found, the transition target state's prepare action is executed. This way, the prepare action can be used to prepare the activity of the respective state in expectation that it might be entered next.

The use of preview coordination is illustrated by the example statechart shown in Figure 4.15. This statechart models a mobile robot skill for retrieving an object. Object retrieval consists of three nominal substates of first approaching the target location, then moving the arm close to the object to be grasped and lastly grasping the object. Additionally, the situation of unexpected

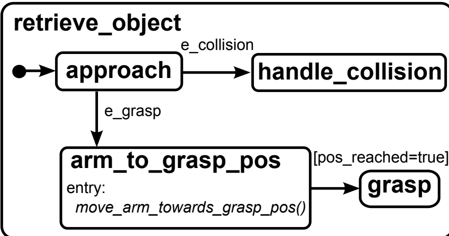

Figure 4.15: Object retrieval without Preview Coordination.

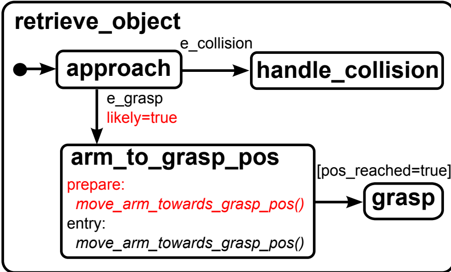

Figure 4.16: Object retrieval with Preview Coordination.

collisions is dealt with. For simplicity, details on the realization of `approach` and `handle_collision` are omitted.

The key observations are that firstly, the nominal sequence from `approach` to `arm_to_grasp_pos` will take place most of the time, and secondly, that the behaviors of `approach` and `arm_to_grasp_pos` are neither logically nor kinematically in conflict and can hence be executed in parallel. To exploit this knowledge using preview coordination, the statechart from Figure 4.15 is extended as shown in Figure 4.16.

Firstly, the transition from `approach` to `arm_to_grasp_pos` is marked as likely (`likely=true`). Secondly, a `prepare` function is added to `arm_to_grasp_pos` for preparing the states behavior. In this case, the prepare function is the same as the entry function `move_arm_towards_grasp_pos()`. This assumes that this function can be invoked more than once and will immediately return if the arm is already at the desired position.

As a consequence of these extensions, the `prepare` function of the `arm_to_grasp_pos` state is invoked even though the robot is still approaching the grasp position. This in turn causes the robot arm to be moved to a suitable grasp position, thereby reducing or even eliminating the time spent in `arm_to_grasp_pos` after

the approaching phase has completed.

It should be noted that the described Discrete Preview Coordination mechanism does not improve the expressiveness of the rFSM model; the same result could be obtained by calling `move_arm_towards_grasp_pos()` from within the `approach` state. The main advantage is, however, that the prepare action can be placed in the context of the semantically related state, namely `arm_to_grasp_pos`. Only this way, both `approach` and `arm_to_grasp_pos` states can be reused independently of each other.

The described preview coordination mechanism has been implemented as a rFSM plugin that extends the core execution semantics. This preview implementation permits the `likely` attribute to be defined as a function that returns true or false. This way, the likelihood of transitions can change over time and is not limited to static load time specification.

The described approach currently only considers a *preview horizon* of one state; however the rFSM preview plugin could be easily extended to two or more. The major challenge for applying such multi-state horizons in practice will be to detect potentially conflicting `prepare` actions. Such reasoning will require more formal representation of actions (and thus robot tasks) and is outside the scope of this chapter.

4.11 Step by step example: constructing coordination for a dual-robot haptic coupling

The following describes approach and methodology to construct the coordination of a dual robot haptic coupling.[14] Two KUKA youbots are to be coupled in a bidirectional manner in cartesian space using an impedance controller. That way, either youbot can be used to move the other and forces applied on one side can be felt on the other (see figure 4.17).

Furthermore, the coupling shall satisfy the following requirements. Initially, both arms are decoupled and compensate for gravity, thus they can be moved freely around by operators. The force coupling between the two robots is only established once two constraints are satisfied. Firstly, the communication quality between the two robots must be sufficiently good (here defined in terms of round-trip latency). Secondly, the end-effector forces that would result from the coupling must not exceed a certain threshold. In other words, if the end-effectors of the robots are too far apart (relative to their bases), then the resulting force

[14] This demo was shown at the Automatica 2012 tradefair in Munich.

Figure 4.17: The youbot coupling demo at the Automatica trade fair. The forces of pulling on one robot arm are felt on the other side. Once the forces rise above a threshold, the coupling is disabled and both robot arms are put into floating mode.

that would pull them together would be too high too, thus preventing the coupling. Moreover, it shall be possible to manually switch between a five and eight degrees of freedom mode in which either only the arm or arm and omnidirectional base are used. This switching shall only be possible when the coupling is established.

The first step to model coordination is generally to examine the architecture of computational components. In most cases this architecture is more or less fixed as a consequence of reusing existing components. Figure 4.18 depicts the component architecture to control each robot. The coupling is achieved by connecting each impedance controllers desired position to the peer robots measured position. The resulting cartesian space force is locally communicated to the dynamics component that computes the inverse dynamics; the resulting desired joint-space forces are then sent to the driver for execution.

How many statecharts shall be introduced to coordinate this system? A best practice facilitating this decision is the following: any subsystem connected via unreliable or temporally non-deterministic communication should be coordinated by a separate, loosely connected coordinator. This rules out the approach of making one robot the master that supervises the slave robot. A further motivation for introducing two coordinatiors is reuse. Since the same computational architecture is used on each robot, it should be feasible to achieve the same symmetry and hence reuse for coordination.

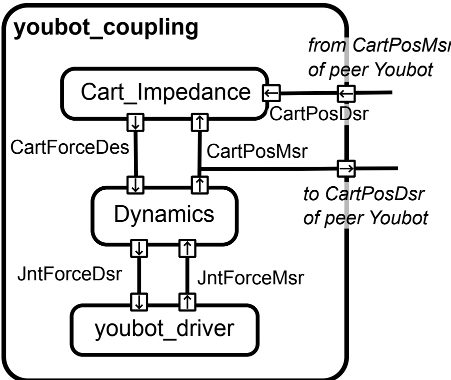

Figure 4.18: Computational component architecture of the dual youbot haptic coupling.

Next it must be considered how an individual coordination statechart can obtain the necessary state information from the system. For reactive models like statecharts, this information is usually best represented by events. For this application we are interested in the communication quality and the impedance controller force output `CartForceDes`. Yet, which component shall raise these events? Generally, there are two fundamental strategies for this: *embedding* the logic of event raising within a computational component or by introducing a *separate monitor* component for that purpose. By avoiding additional communication, the embedding strategy allows for lower latencies than with the external monitor. On the other hand, the external monitor component avoids polluting the computational component with application specific details.

For the coupling application one could consider extending the impedance controller component to raise communication quality events, since it is receiving the cartesian position from the peer (for instance by determining the communication latency using the creation timestamp of the received measured position). This is obviously a bad choice, as it would clutter the controller component with application specific information. Thus, we introduce a separate, external communication monitor that determines the communication latency based on timestamped heartbeat messages exchanged with the remote side. Based on a configurable quality level, the events `e_QoS_OK` and `e_QoS_NOTOK` are emitted.

In contrast, for raising the force threshold events, the embedding approach was chosen and the impedance controller extended to raise two events `e_force_thres_exceeded` `e_force_thres_below` upon exceeding respectively falling below a configurable threshold. Unlike with the communication quality,

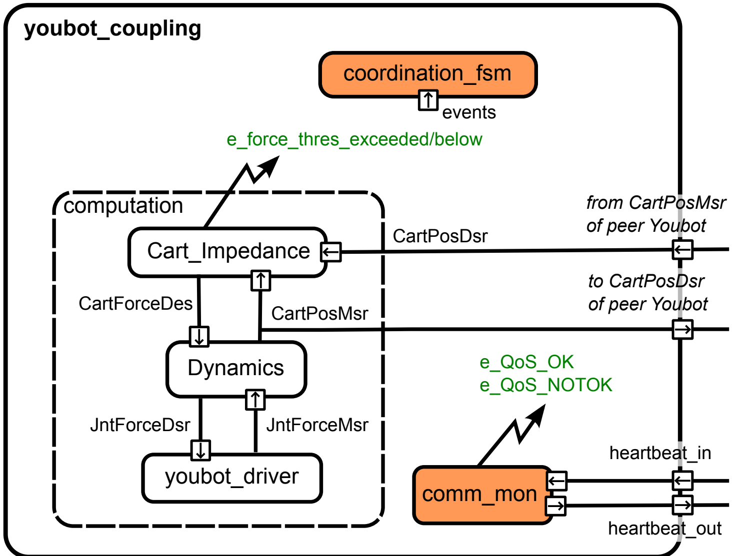

Figure 4.19: Component architecture with extensions to support coordination.

the desired output is an essential quantity of a controller and hence this event is likely to be useful in other circumstances. Secondly, to permit simple switching between gravity compensation and coupling mode, the impedance controller was extended with a Boolean mode *external reference mode* (`ext_ref_mode`). When true (the nominal case), the external desired position (`CartPosDsr`) is taken into account to compute the output force. If false, the external input is ignore and instead zero forces are output. A functionally equivalent and less intrusive solution to achieve emitting a zero force would have been to (on-the-fly) remove the external `CartPosDsr` connection and instead connect the measured positions `CartPosMsr`. However, since rewiring connections is not a real-time safe operation in Orocos, we opted for the former.

Figure 4.19 shows the extended architecture including a coordination component containing a (yet to be defined) rFSM statechart, the external monitor component raising the QoS events and the extended impedance controller component.

Defining coordination statecharts is best carried out by top-down refinement. Since statechart priorities are decreasing with depth, this corresponds to starting with the highest priority states and transitions. For this application this is the requirement that the communication is established and sufficiently good. Without communication coupling the robots is impossible. Thus, we introduce

two states synchronized and unsynchronized that model this requirement (Figure 4.20).

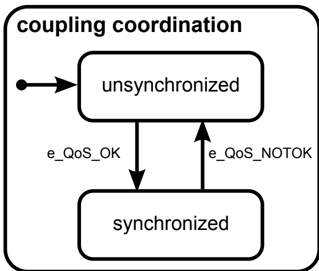

Figure 4.20: Constructing the coordinator, step 1: modeling the communication quality constraint.

Next we refine the systems behavior by extending the synchronized state (entered when the communication is good enough) by adding two substates gravity_comp and copying (Figure 4.21). gravity_comp represents the state in which the impedance controller output force CartForceDes is too high and hence gravity compensation is enabled. In copying the actual coupling is established by enabling external reference mode. Conversely, external reference mode is disabled upon exiting the copying state, since only that way it is guaranteed that this mode is disabled no matter how copying is exited. For instance, if external reference mode were only disabled in the entry function of gravity_comp, the transition to unsynchronized triggered by an e_QoS_NOTOK when in copying would result in entering unsynchronized with external reference mode still enabled.

Note that in contrast to the toplevel transitions between unsynchronized and synchronized, the transitions between gravity_comp and copying make use of a guard condition above_force_thres() instead of being triggered by events. The reason for this is that upon entering the synchronized state the impedance controller forces will already be either too high or not and the corresponding events already raised. Thus, if these transitions were triggered only by events, the gravity_comp state might erronously remain active, unless by chance this condition just changes after entering gravity_comp.

```
function above_force_thres()
   local flow_status, value = force_thres_ex:read()
   if flow_status == 'NoData' then return false end
   else return value end
end
```

Listing 4.7: Implementation of above_force_thres guard condition.

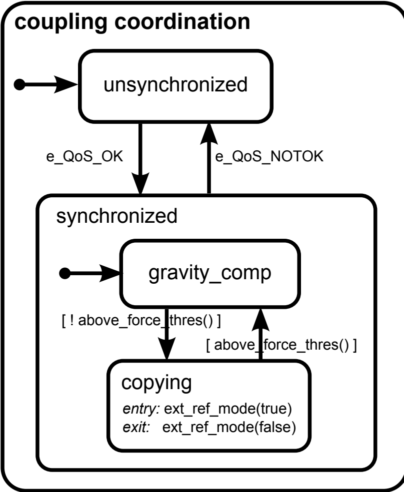

Figure 4.21: Constructing the coordinator, step 2: modeling the force threshold constraint.

To realize this guard, the edge triggered events emitted by the impedance controller can easily be transformed into the level triggered events required by the guard condition. Listing 4.7 shows how this can be achieved using the Orocos RTT framework. The latest received event is cached in the coordination component using a regular data-port (`force_thres_ex`), that then can be read and evaluated by the guard condition.

What remains now is only the functionality to switch between five and eight DOF mode. Once again this is achieved by top-down state refinement, in this case by extending `copying` with the two modes, as shown in figure 4.22. The operator can switch between both modes by sending the `e_8DOF` and `e_5DOF` events respectively. As an optimization, a history connector could be used instead of a plain initial connector. That way the previously selected mode would be resumed after an interruption of the coupling.

Using states to model modes works nicely when modes are mutually exclusive. However consider the requirement to additionally support two switchable modes *low-* and *high force threshold* in which the force threshold is reconfigured accordingly. Extending copying to support these modes would result in a combinatorial explosion of states. A good way to solve this is to use an SLDS state (discussed in Section 4.10.6) to compose the orthogonal modes as separate, concurrent substates.

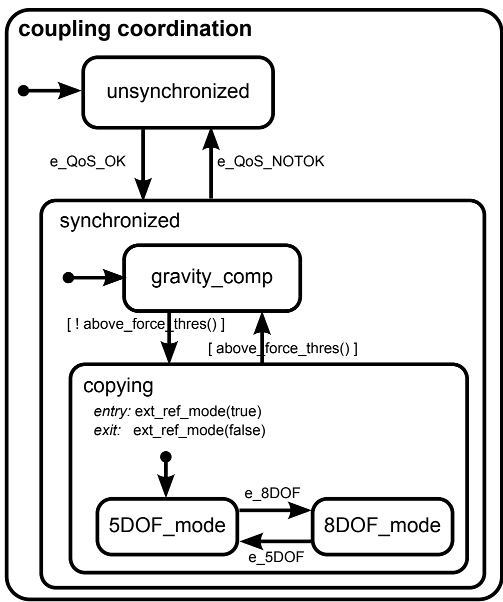

Figure 4.22: Constructing the coordinator, step 3: adding modes

4.12 Discussion

This chapter describes a statechart model consisting of a *minimal* number of semantic primitives necessary for constructing *practical* robotic coordination. This minimality approach has the advantage of reducing implementation complexity, avoiding introducing unnecessary assumptions (as for instance the aforementioned communication properties) and being simple to understand and use.

Nonetheless, a minimality approach also has some disadvantages. For one it places a higher burden on developers to construct the required *composites*. We intend to address this by introducing a *standard library* of coordination extensions. First mechanisms included are event memory, the serialised locally-distributed state and the preview coordination extension.

Additional effort is also required for deploying a distributed statechart, as connections and communication must be configured between the different FSM. This involves considering communication reliability, defining buffer sizes, triggering and buffering policies. Little if any tooling exists to support this process; tools for deploying *computations* might be of some help, yet the different characteristics of *coordination* will most likely require dedicated tools. As an

example, having multiple simultaneously active writers communicating data to a single reader is often an erroneous situation for Computations (e.g. only one controller may command a robot at a time). Conversely, it is common to connect multiple, active event sources to one Coordination statechart.

For using the rFSM model in the context of complex distributed robotic systems, we propose to make exclusive use of *pure coordination*. Pure coordination can be achieved by rigorously separating the concerns of Computations from Coordination; more concretely by limiting Coordination actions to exclusively raising events. The benefits of adhering to this best-practice are increased robustness and reusability of Coordination. The drawback, however is that as of today few components provide sufficiently expressive constraint configuration interfaces; one reason for this being presumably the lack of a generic mechanism to specify condition-event pairs on internal state of computations.

One motivation for choosing graphical models (as statecharts) to model complex systems is their apparent ease to be understood by humans. Yet surprisingly few tools exist to visualise graphical models (online or offline) by performing automatic layout and rendering of the textual representation. The tool most widely used for this purpose is probably graphviz (Gansner and North 2000), that is also used by the rFSM reference implementation for visualising statechart models. Unfortunately, the generated representation, especially when involving hierarchical states, is often suboptimal[15]; significant fine tuning of layout parameters only resulted in a marginal improvement. Nevertheless, this simple tool has proven invaluable to validate that a model specified in textual form does indeed correspond to the graphical model the developer has in mind. We intend to address this topic in future work.

4.13 Conclusion and Future Work

We have presented a lightweight Coordination statechart model that is derived by analysing and extracting a minimal subset of model elements from existing formalism. The proposed rFSM model is graphically a subset of UML with simplified execution semantics derived from the well known STATEMATE statecharts. By selective inclusion of model elements, many of the corner cases and additional rules required by existing formalism can be avoided, thus simplifying both implementation and coordination models themselves.

Instead of providing a rich set of built-in features for all possible use-cases, the rFSM model advocates dealing with complexity by composition.

[15]Optimality defined by how a human would draw a statechart, as is the case with all figures (apart from Figure 4.10) in this chapter.

Composition means both local hierarchical composition of Statecharts or distributed composition as described in Section 4.10.5 as well as composition of core execution semantics with run-time extensions such as event memory.

To back up these claims we implemented an extensible, framework independent, real-time safe reference implementation based on the Lua scripting language. The featurewise almost complete rFSM core engine currently amounts to less than 830 lines of code[16]. This reference implementation has successfully been integrated into the OROCOS RTT software framework and applied to a wide range of use-cases.

Moreover, we describe several best-practice patterns of robotic task and system coordination that were discovered during our work. These patterns serve to highlight frequent coordination design issues together with best practice solutions, with the goal of fostering adoption of the rFSM model.

Up to now, the described rFSM model has been sufficient to describe all robotic task and system coordination we encountered, hence there currently seems no need for extensions to the core model. On the other hand we observe a severe lack of tools to support creating, deploying and visualising rFSM statecharts and Coordination models in general. We intend to address this in future work.

[16]Calculated using the cloc(1) tool.

Chapter 5

Reusable hybrid force-velocity controlled motion specifications with executable domain specific languages*

5.1 Abstract

Most of today's robotic task descriptions are designed for a single software and hardware platform and thus can not be reused without modifications. This work follows the *meta-model* approach of *Model Driven Engineering* (MDE) to introduce the concepts of *Domain Specific Languages* (DSL) and of *Model Transformations* to the domain of hybrid force-velocity controlled robot tasks, as expressed in (i) the Task Frame formalism (TFF), and (ii) a Statechart model representing the discrete coordination between TFF tasks. The result is a representation in MDE's M0, M1, M2 and M3 form, with increasingly robot and software independent representations, that do remain instantaneously executable, except obviously for the M3 metametamodel. The *Platform Specific Model* information can be added in three steps: (i) the *type* of the hybrid force-velocity controlled task, (ii) the *hardware properties* of the robot, tool and

*This chapter is based on: Klotzbücher, M., Smits, R., Bruyninckx, H., and De Schutter, J. *Reusable hybrid force-velocity controlled motion specifications with executable domain specific languages.* In Proc. IEEE/RSJ Int. Conf. Int. Robots and Systems (San Francisco, California, 2011), pp. 4684–4689.

sensor, and (iii) the *software properties* of the applied execution framework. We demonstrate the presented approach by means of an *alignment task* executed on a Willow Garage PR2 and a KUKA Light Weight Robot (LWR) arm.

5.2 Introduction

The Task Frame formalism (TFF) was introduced by Mason (Bruyninckx and De Schutter 1996; De Schutter and Van Brussel 1988; Mason 1981), to specify hybrid force position/velocity control operations. This formalism identifies six force- and velocity-controlled directions, coinciding with the axes of a single frame, the *Task Frame* or *Compliance Frame*. The TFF abstracts from the exact robotics hardware (manipulator, sensor, tool) and software (hybrid force/velocity controller) that are used to execute the task, but these "platform-dependencies" are introduced implicitly via the force and/or velocity *setpoints* that are required in each degree of freedom. Although being used in practice by many labs and in many applications, the TFF *representation* has never been standardized, which would be the first step towards *reuse* of the same task specification over multiple applications and robot platforms. This chapter makes a first attempt towards such "standardization", in the established context of the *Model Driven Engineering* (MDE) framework[1], more in particular, by using the concepts of (i) a *Domain Specific Language* (DSL), and (ii) *metamodeling*, (Atkinson and Kühne 2003; Flatscher 2002). The added value of this work lies in the structured way to deal with various platform-dependencies: (i) the *type* of the hybrid force-velocity controlled task, (ii) the *hardware properties* of the robot, tool and sensor, and (iii) the *software properties* of the applied execution framework.

Domain Specific Languages are lightweight programming languages designed to concisely express the concepts of a particular domain. Commonly two types are distinguished: internal and external DSL. The first are built on top of an existing language while the second are developed from scratch. By reusing existing infrastructure, internal DSL are much easier to create and maintain than external ones. The only disadvantage of an internal DSL is that its syntax is constrained by the host language. In practice this limitation is often acceptable given the reduction in development time and the ease of combining DSLs together. In this work we define multiple internal DSL to facilitate programming with the TFF. The goal of this is threefold: firstly to support the programmer in constructing complex behaviors in an intuitive way (via

[1]The Object Management Group (Object Management Group b) is the main driver of standardization in the context of Model Driven Engineering, for which it uses the trademarked name *Model Driven Architecture*.

the familiar concept of a *finite state machine*, secondly by facilitating reuse of these behaviors or parts thereof in new applications (via the introduction of a TFF-centric DSL), and thirdly to cleanly separate the generic behavior from robot hardware and software framework dependent aspects (via MDE's four levels of modeling, as depicted in Figure 5.1).

Figure 5.1) sketches a systematic approach to model a certain domain in four *levels of abstraction*; in this chapter's context of TFF robot tasks, these four levels have the following meaning:

M0: the level of the concrete implementations, for example, using particular C++ libraries for execution of TFF specifications, using the Orocos and/or ROS framework.

M1: the level of a *platform-specific* TFF specification DSL, using the force and velocity setpoints that are appropriate for, for example, the KUKA LWR, or the PR2.

M2: the level of the *platform-independent* TFF specification DSL, which provides a *parameterized template* for each sub-task, in which the platform-specific setpoints have to be filled in later.

M3: the highest level of abstraction, that is, the model that represents all the constraints that a DSL has to satisfy in order to be a valid TFF DSL.

This chapter gives DSL implementations on the M1 and M2 levels, and assumes that an implementation for the M0 level is available. For the M3 level the Ecore language is used.[2]

To illustrate our approach, we modeled and implemented an object alignment task, and executed it on different robot platforms (see Figures 5.2–5.3). This task is one of the simplest ones that still shows all relevant concepts introduced in this chapter. (**The focus of this chapter is on the concepts, and not on the complexity or innovation of this particular task**.) Figure 5.2 shows the steps of the alignment task executed by a KUKA LWR. The top-left image shows the unaligned initial position of the manipulator, from which the arm holding the object moves down with a constant velocity until contact with a surface is detected (top right). This triggers a transition to a force-controlled, pushing-down motion which causes the end effector to align perpendicular to the surface. After the alignment has completed, a sliding motion (velocity control in the direction towards the right-hand side fixture, and force control perpendicular to the surface) is executed (bottom-left image). When contact is

[2]Ecore is the M3-level metametamodel language standardized by the Eclipse EMF consortium, *www.eclipse.org/emf*.

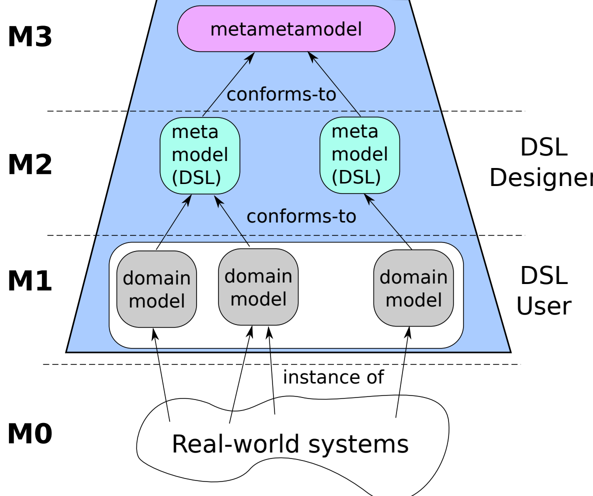

Figure 5.1: The four levels in OMG's standard for the role of Domain Specific Languages in Model Driven Engineering.

established with the fixture, a sideways pushing motion is activated, resulting in alignment against the fixture (bottom right image). Figure 5.3 shows the same alignment task being executed by one arm of a PR2 robot.

5.2.1 Related work

The first integrated programming and simulation environment for the Task Frame Formalism was COMRADE (Bruyninckx and De Schutter 1996; Van de Poel, Witvrouw, Bruyninckx, and De Schutter 1993). COMRADE reduced the task specification into three steps: (i) initialize the model by specifying the Task Frame and selecting the force- and velocity-controlled directions, (ii) choose the desired values for each force and velocity and (iii) specify the termination conditions for the motion. Coordination of such TFF tasks into possibly complex sets of tasks and subtasks, was introduced by (Finkemeyer, Kröger, and Wahl 2005), using finite state automata. This chapter's DSLs encompass both aspects (task execution as well as the coordination of the execution of various tasks), but decouples them by providing two different DSLs for each of the two responsibilities. This approach is less coupled to a specific behavioral model, and permits easier integration with any formalism suitable for modeling coordination. In addition, we describe a concrete approach for integrating these behavioral models with different robots and software frameworks.

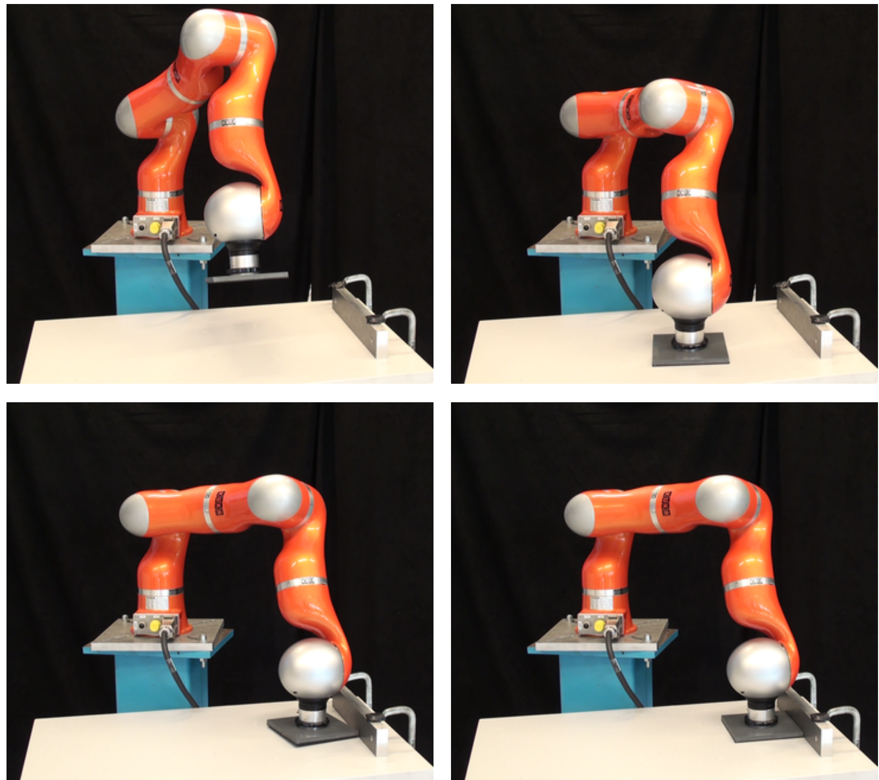

Figure 5.2: The alignment task executed by the KUKA LWR, with an implementation based on Orocos/RTT.

5.3 Domain Specific Languages for M1, M2, M3

For the M1 and M2 level, this chapter presents DSLs specified using the Lua extension language (Ierusalimschy, de Figueiredo, and Filho 1996). In principle, any scripting language could serve the same purpose, but Lua is one of the few that can be used with real-time performance (Klotzbuecher, Soetens, and Bruyninckx 2010). Moreover, Lua has already been integrated into several robotic software frameworks, including Orocos/RTT and ROS. Sections 5.3.1 and 5.3.4 introduce the DSLs (at the M1 and M2 levels), to represent one single TFF task and the discrete coordination of multiple TFF tasks, respectively. The M3 level is briefly touched upon in Section 5.3.2, and Section 5.3.3 explains the M0-level implementation.

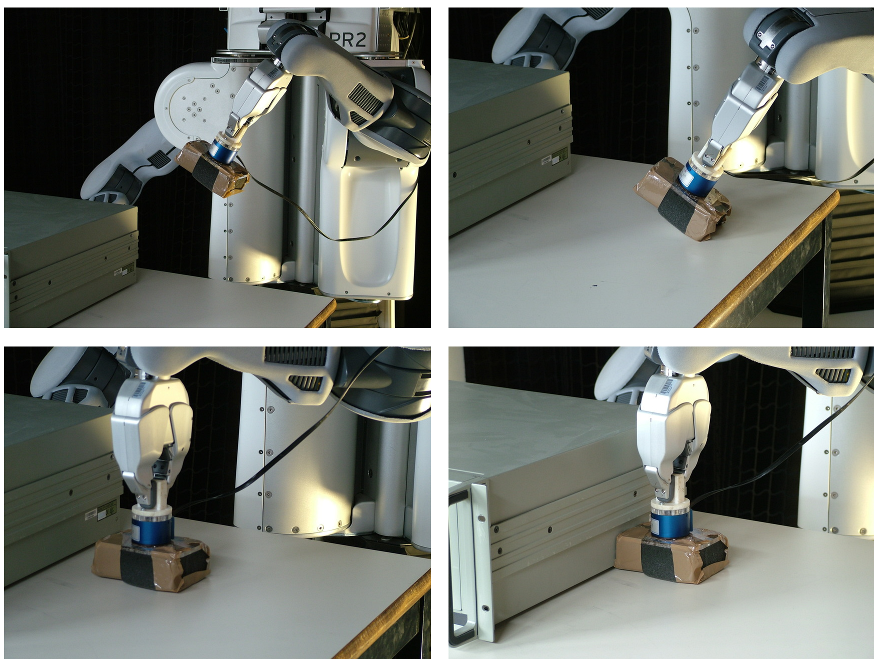

Figure 5.3: The alignment task executed by the PR2, with an implementation based on Orocos/RTT and ROS.

5.3.1 M1- and M2-level TFF-DSLs

At the M2 level, the TFF-DSL defines a *template* for each *type* of hybrid force/velocity controlled TFF specification. For example, `move_down` moves the end effector downwards with a constant velocity, or `push_down` applies a force downwards with the other directions force-controlled and set to zero. `compliant_slide_x` controls the arm to move in the `x`-direction while applying a force downwards. The DSL follows very straightforwardly from the descriptions in (Bruyninckx and De Schutter 1996); the following listing shows three sample TFF-DSL specifications:

```
push_down = motion_spec:new {
  xt=ax_spec:new{ type='force', value=0, unit='N' },
  yt=ax_spec:new{ type='force', value=0, unit='N' },
  zt=ax_spec:new{ type='force', value=10, unit='N' },
  axt=ax_spec:new{ type='force', value=0, unit='Nm' },
  ayt=ax_spec:new{ type='force', value=0, unit='Nm' },
  azt=ax_spec:new{ type='force', value=0, unit='Nm' }
}
```

```
move_down = motion_spec:new {
  xt=ax_spec:new{ type='vel', value=0, unit='m/s' },
  yt=ax_spec:new{ type='vel', value=0, unit='m/s' },
  zt=ax_spec:new{ type='vel', value=0.1, unit='m/s' },
  axt=ax_spec:new{ type='vel', value=0, unit='rad/s' },
  ayt=ax_spec:new{ type='vel', value=0, unit='rad/s' },
  azt=ax_spec:new{ type='vel', value=0, unit='rad/s' }
}

compliant_slide_x = motion_spec:new {
  xt=ax_spec:new{ type='vel', value=0.03, unit='m/s' },
  yt=ax_spec:new{ type='force', value=0, unit='N' },
  zt=ax_spec:new{ type='force', value=1, unit='N' },
  axt=ax_spec:new{ type='force', value=0, unit='Nm' },
  ayt=ax_spec:new{ type='force', value=0, unit='Nm' },
  azt=ax_spec:new{ type='force', value=0, unit='Nm' }
}
```

Listing 5.1: TFF-DSL specification.

Each motion specification consists of six specifications for the three translational and rotational degrees of freedom of the Task Frame. Each axis specification consists of a type which is either 'velocity' (abbr. 'vel') or 'force', a value and a unit. It is important to note that the `value` is only a *placeholder default*, to be redefined when the *platform-specific* information is introduced (which gives the M1 level TFF-DSL). Indeed, the presented templates hold for *any* robot-tool-sensor combination[3], but the specific *magnitudes* of the force and/or velocity setpoints depend on the selected platform.

The M1 and M2 levels of the presented TFF-DSLs are extremely similar, which is considered an advantage, since this similarity implies a smoother learning curve, and simpler software support. But despite the large similarity, the *transformation* between both levels, however, can be very involved: although the transformation is "only" about filling in the right setpoint magnitudes, it is in this transformation that one has to introduce all the platform-specific knowledge, about the particular robot, sensor and tool. Our experiences with the two examples presented later in this chapter (KUKA LWR and PR2) showed that the "optimal" parameter settings for both platforms are very different, because of the difference in performance between both platforms, which is about one order of magnitude.

5.3.2 M3 model: Ecore

At the M3 level of Figure 5.1 the Ecore metametamodel is used. Figure 5.4 shows the M2 TFF-DSLs of the previous section specified using Ecore. Currently,

[3]At least, in so far as it provides a hybrid force/velocity control implementation!

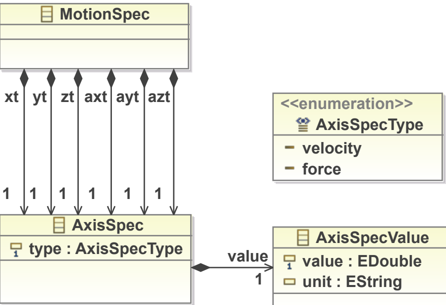

Figure 5.4: The Ecore metamodel (M2 level in Figure. 5.1) that represents the formal model that all DSLs for hybrid force/velocity control should conform-to.

the Ecore model serve visualization and documentation purposes only, and the transformation into the executable DSL has still to be performed manually. Future work will address the automatic generation of DSL from Ecore models.

5.3.3 Software Framework integration: from M1 to M0

The TFF-DSL permits specifying motions; for *executing* these a specification must be transformed into an appropriate hybrid controller *implementation*. For performance reasons, these controllers are commonly implemented in C/C++; for this chapter we have chosen the Orocos and ROS frameworks. (No further details are given, since the implementations are not the focus of this chapter's contributions.) The loose coupling between the TFF-DSL specifications and the Orocos/ROS controller implementations is realized via a *transformation* function `apply`, which accepts as arguments a motion specification and optionally a list of redefined values, as shown in Listing 5.2.

```
apply(compliant_slide_x, {xt=0.2, zt=1})
```

Listing 5.2: Example usage of `apply` transformation function.

An implementation of this `apply` function was developed in Lua. This function first transforms the `compliant_slide_x` motion with redefined values for the `xt` and `zt` placeholders, then converts the motion specification to the Orocos platform-specific representation, and lastly sends it to the platform's TFF controller. Since these steps are always the same for any TFF-DSL, the apply function can be automatically generated. For example for the Orocos/RTT (Soetens and Bruyninckx 2005), (Soetens 2006) platform, `apply` is generated by

the following code:

```
function gen_apply(port)
   local rttms = rtt.Variable("MotionSpec")
   return function (ms, overrides)
             set_rtt_ms(rttms, ms, overrides)
             port:write(ms)
          end
end
```

Listing 5.3: Generation of apply transformation function.

The gen_apply function generates a function which accepts a motion specification and a list of default template values to override. This motion specification is then written to the communication port which was passed as an argument to gen_apply.

5.3.4 Composing individual TFF-DSL motions into skills using the rFSM statechart DSL

After defining elementary motions, a rFSM Statechart (Chapter 4) is used to compose these basic blocks into a robust skill (Smits 2010). The rFSM model used is a lightweight Statechart variant designed for real-time coordination of robotic systems, and is as such suitable for this task. However, if required in different scenarios, any other behavioral model could be used instead. The robustness is achieved by explicitly validating the outcome of motion executions in the respective states and raising error events in case invalid conditions are detected.[4] The graphical model of this Statechart is shown in Figure 5.5.

The nominal behavior begins at the initial connector at the top left of the figure and runs through the move_down, pre_align_z, align_z, pre_slide, compliant_slide_x and align_x states to the final connector at the bottom-left. Validation is performed in each state, and in case anomalies are detected, respective events are emitted. These events then trigger the transition to the error connector, to which further error handling is connected. The pre_align_z and pre_slide states serve to mitigate unstable behavior of the system during transitions. For instance, instead of immediately pushing down with 20 N when entering in contact, pre_align_z prepares this transition for half a second by pushing down with 5 N.

In addition, the align skill in Figure 5.5 illustrates an advantage of using hierarchical Statecharts instead of flat state automata: conflicts between

[4] The rFSM design, and the robustness aspects of its implementation, are beyond the scope of this chapter.

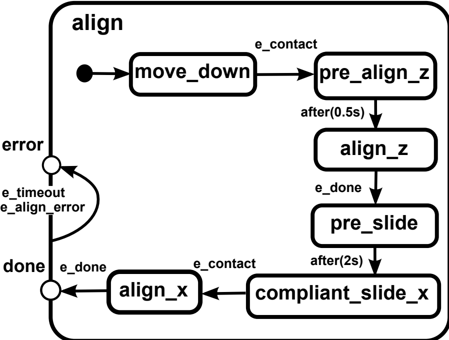

Figure 5.5: rFSM Statechart for the alignment skill.

transitions are resolved by the Statechart structural priority rules (Harel and Naamad 1996). As the error transition originates from the top-level `align` state, it has higher priority than the deeper nested motion states. Consequently, in case of simultaneous error and non-error events, the error transition will be executed.

An excerpt of the textual rFSM DSL corresponding to Figure 5.5 is shown below.

```
align = state {
   move_down = state {
      entry = function () apply(move_down, {zt=0.01}) end
   },
   pre_align_z = state {
      entry = function () apply(push_down, {zt=5}) end
   },
   align_z = state {
      entry = function () apply(push_down, {zt=20}) end
   },
   transition { src='initial', tgt='move_down' },
   transition {
      src='move_down', tgt='pre_align_z',
      guard = function ()
                 return get_force_TF().force.z > 4
              end },
   transition {
      src='pre_align_z', tgt='align_z',
      guard = function ()
                 return get_move_duration() > 0.5
              end
   },
}
```

Listing 5.4: Textual model of align rFSM statechart.

The first line defines the top level state which encapsulates the `align` skill. Next the states `move_down`, `pre_align_z` and `align_z` are defined. In the `entry`

program of these states the use of the `apply` function for executing the next motion can be seen. After the state definition, transitions are defined to link states together. The `guard` function is used to inhibit the transition until the respective condition holds true. For instance in the second transition between `move_down` and `pre_align_z` the condition of entering in contact is defined by a force larger than 4N in z-direction.

5.3.5 Dealing with robot dependencies

As with the TFF-DSLs of Section 5.3.1, the rFSM DSL as presented above covers both the M2 and M1 levels. In the rFSM case, the platform dependencies are slightly more complex than in the TFF case: not only the concrete parameter values have to be filled in, but, in general, every robot platform could require extra platform-specific states.

An example of the latter are the Fast Research Interface (FRI) *command-* and *monitor mode* states of the KUKA Light Weight Robot (Schreiber, Stemmer, and Bischoff 2010). Depending on which of these states is active, the robot can either be actively commanded or only monitored; switches to monitor mode can occur at any time as a result of bad communication quality.

Another example is the introduction of "transition dynamics" sub-states, which are especially relevant for low-performance dynamic control systems such as the PR2's standard joint space controller: whenever a TFF motion is started, or a transition to the next TFF motion is triggered by a contact formation change, the robot arm tends to vibrate for some time. Hence, it may be necessary to introduce a state for the sole purpose of letting these vibrations damp out.

To achieve the harmonization necessary to execute the align skill (an M2-level requirement), a robot specific configuration file was introduced (in the M2-to-M1 transformation) that encapsulates the required parameters and performs initialization. For example, to deal with the KUKA FRI states, an additional, robot-specific rFSM Statechart was defined, which is shown in Figure 5.6.

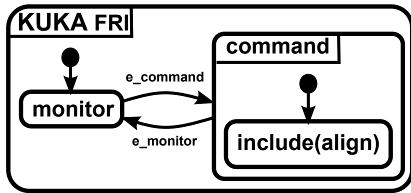

Figure 5.6: Harmonizing the KUKA FRI operational modes.

The Statechart models both FRI states and the connecting transitions which are triggered by events raised by the low-level FRI controller. The robot-independent `align` Statechart is *included* into the FRI `command` state. This way it is guaranteed that the align statechart is only executed when the FRI is in command mode. If an unexpected switch to monitor mode occurs the align skill will be exited cleanly. More importantly, the robot-independent behavior remains agnostic of the FRI statechart and is not polluted with robot specific information.

As similar configuration file is used for the Willow Garage PR2. Since the PR2 does not require specific operational states, the align statechart is executed directly.

5.4 Experiments on PR2 and KUKA LWR

The PR2's two 7 DOF arms are passively backdrivable, making them very suitable for manipulations tasks in contact with an environment. Due to the lack of a built-in force sensor, a JR3 force/torque sensor was added in the gripper of the PR2. The LWR is not passively backdrivable, but provides a controlled compliance mode and force sensing via its internal joint torque sensors[5].

Figure 5.7 shows measured forces, desired Cartesian velocities and gripper tool frame position and orientation of the alignment task over time for the PR2. The vertical black lines mark when a new statechart state was entered.

The plot of the measured forces clearly shows the benefit of the `pre_align` and `pre_slide` states in reducing unstable, oscillating behavior during switches of motion specifications. Although simple and effective, it is intended to replace these (manually introduced) intermediate states by (automatically generated) transformation states, that blend two motion specifications within a given time frame.

A plot of the task execution over time on the KUKA LWR is very similar to that on the PR2 and is hence omitted. A notable difference is that the measured forces are much smoother on the LWR. On the one hand, this is explained by the task being executed slower than on the PR2 (just to show the influence of different platform-specific parameters). On the other hand, the LWR's controller is much more performant that the PR2's.

[5]The source code of these experiments is available here:
https://github.com/kmarkus/tff_dsl_tests

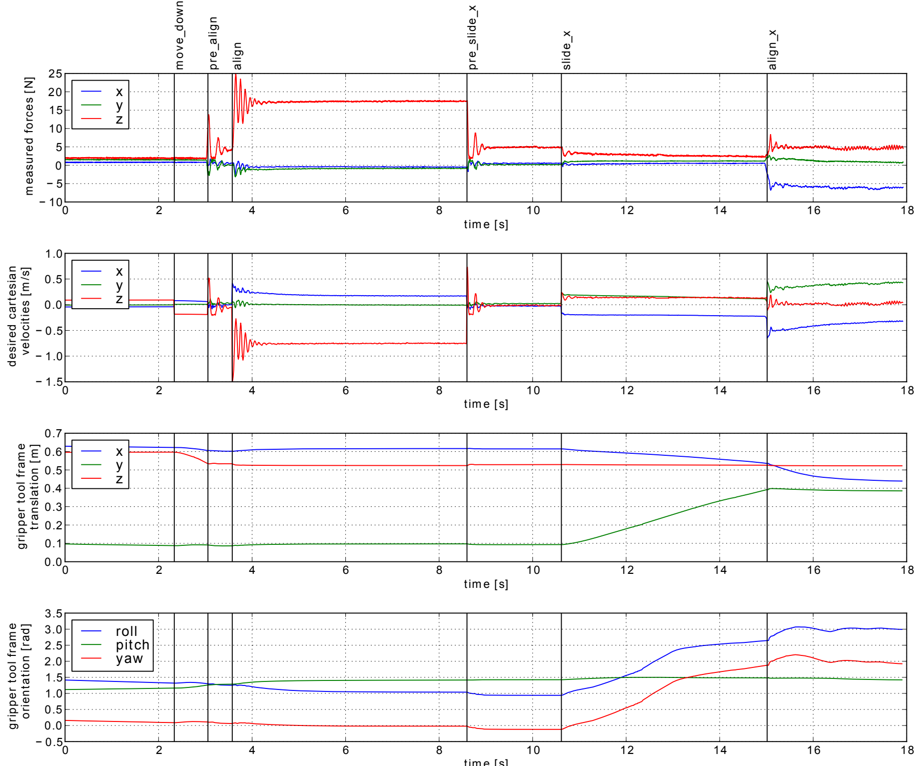

Figure 5.7: Forces, velocities and positions of the PR2 robot.

Source code	Lines of code
Elementary TFF motions	60
Alignment Skill rFSM Statechart	33
Total	93

Table 5.1: Platform-independent code

Tables 5.1 and 5.2 summarize the lines of code for the various models. The platform-independent models required for the alignment skill consist roughly of 100 lines of code. In order to execute these on one of either robots requires approximately an additional 200 lines, of which one half consists of software framework integration and the other of robot specific parameter or operational mode related code. Although it would be desirable, it is unlikely that the software framework code can be much reduced. In contrast it should be possible to significantly reduce the manually developed robot specific code by generating

Source code	Lines of code
KUKA LWR config and initialization	96
KUKA Fast Research Interface rFSM Statechart	13
PR2 config and initialization	102
TFF Orocos/RTT framework integration	86
Total only LWR	195
Total only PR2	188
Total for LWR and PR2	291

Table 5.2: Robot and Framework specific code.

it from a model representing these characteristics. This is the subject of future work.

5.5 Conclusions

We have illustrated the *Model Driven Engineering* concepts of *Domain Specific Languages* and *model transformation* to the case of hybrid force/velocity control. The major contribution of this chapter is proving that DSLs have the potential to bring more user-friendly programming and more standardization in the field of sensor-based robotics. However, the presented approach and the given examples are just scratching the surface. Much more work has to be done, not only in software support (which is probably the easiest task), but certainly also in the domain of standardizing the robotics controller terminology. Without the latter, every particular DSL will not lead to real reusability beyond the confines of the research group that developed it, and will not lead to portability beyond the set of robots and controllers applied in that group.

The presented DSLs facilitate the specification of robot tasks, by composing elementary motion specifications in a behavioral statechart model. They were implemented with an emphasis on loose coupling with the software frameworks underneath, so that complete separation of generic task specification, software framework dependencies and robot hardware requirements is facilitated.

The analysis of lines of program code of the DSL reveals that approximately twice the amount of platform-specific code is necessary to support a generic and reusable task description.

Future work needs to address, at least, the automatic generation of (i) transition dynamics damping states, and (ii) basic executable domain specific languages such as the TFF-DSL from formal descriptions, e.g., Ecore models.

Chapter 6

A Lightweight, Composable Metamodeling Language for Specification and Validation of Structural Constraints on Internal Domain Specific Languages*

6.1 Abstract

This chapter describes a declarative and lightweight metamodeling framework called uMF for modeling and validating structural constraints on programming language data-structures. By depending solely on the Lua language, uMF is embeddable even in very constrained embedded systems. That way, models can be instantiated and checked close to or even within the final run-time environment, thus permitting validation to take the current state of the system into account. In contrast to classical metamodeling languages such as Ecore,

*This chapter is based on Klotzbücher, M., and Bruyninckx, H. A lightweight, composable metamodeling language for specification and validation of internal domain specific languages. In *Proceedings of the 8th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES'12)*, Germany, LNCS, 2012.

uMF supports *open world* models that are only partially constrained. We illustrate the use of uMF with a real-world example of a robot control formalism and compare it with a previously hand-written implementation. The results indicate that strict formalization was facilitated while flexibility and readability of the constraints were improved.

6.2 Introduction

A domain specific language (DSL) is built specifically to express the concepts of a particular domain. Generally, two types of DSL can be distinguished according to the underlying implementation strategy. An *external DSL* is constructed from scratch, starting from definition of the desired syntax to implementation of the associated parser for reading it. In contrast, *internal* or *embedded DSL* are constructed on top of an existing host language (Fowler 2005) (Spinellis 2001). While this technique has been used for decades, in particular by the Lisp and Scheme communities, it has recently gained popularity through dynamic languages such as Ruby. Internal DSL offer some advantages over external ones. Firstly, existing infrastructure provided by the host language can be reused. Secondly, embedded DSL models can be typically loaded/instantiated within the host language without any parsing effort on the side of the DSL developer. Thirdly, when internal DSL are specified in a data-oriented manner (as proposed in this chapter) models can easily be introspected and interpreted at runtime. On the downside, as models are specified using the host language (or a subset thereof), the syntax of an internal DSL is always constrained by the syntax of the host language.

As an internal DSL reuses the parsing infrastructure of its host language, basic syntax checking and error reporting is taken care of automatically. Using OMG MDA terminology, the host language can thus be seen as the M2 (meta-) model to which the M1 model must *conform-to* (Bézivin 2005).

Obviously, this conformance relationship constrains any real-world DSL too little, hence additional checking is necessary to validate conformity with the concepts of the domain. For internal DSL, this is mostly achieved by manual programming the validation, which is a tedious and error prone task.

Consider a DSL for modeling 3D vectors. If object oriented concepts are supported by the host language, the obvious choice would be to define a class and use its constructor to set and validate the attributes. We would like to avoid this approach for two reasons. Firstly, it still requires implementing manual validation in the constructor. Secondly, most object oriented programming (OOP) models typically *close the world* for the model, meaning that additional

attributes are prohibited. Our goal is to avoid this implicit closing for two reasons: firstly, explicitly permitting additional attributes in addition to the known enables implementing plugins that extend the behavior of the default model. As an example, the vector could be extended with physical units to be used for validation in arithmetic operations. Secondly, open models permit validating a single model against multiple specifications, of which each validates one aspect of the model. An example for this is given in Section 6.4.1.

Lastly, open models enable development of DSL in a more natural order: instead of starting by specifying the metamodel and only then defining and testing models, this approach permits to start by defining concrete models, and only later to add structural constraints when the underlying metamodel is becoming apparent.

As an alternative to the class based specification of 3D vectors, Listing 6.1 shows the use of a simple associative array. This alternative seems likewise complex to validate, as it requires asserting that the fields `x`, `y` and `z` are defined and of correct type. If the type is defined to be *closed*, the absence of any other fields must be checked. Moreover, if *optionality* of entries is supported, it must be asserted that all non-optional fields are defined.

```
v1 = { x=1, y=0, z=0 }
```

Listing 6.1: 3D vector model specified using a Lua table

To overcome these limitations we propose a declarative, constraint specification DSL, that is part of a minimalistic Lua (Ierusalimschy, de Figueiredo, and Filho 1996) framework called uMF (micro modeling framework). uMF permits specifying and validating the structure of Lua data types and is complemented by a mechanism for declaring typed arrays. While in principle most dynamic programming languages could have been used, Lua has some benefits for building embedded DSL that are described below.

6.2.1 Internal DSL in Lua

The Lua language is an extremly lightweight[1], yet multi-paradigm scripting language whose semantics are strongly influenced by the Scheme dialect of Lisp. Amongst others, this is revealed in the composite *table* data structure (the analog to Lisp's list), that apart from behaving as a list (`{1,2,3}`) can also function as an associative array (`{x=1, y=2}`) or even as both simultaneously (`{x=1, y=2, 1, 2, 3}`. Tables are the key mechanism for constructing embedded Lua DSL that yield an intuitive syntax as shown in the examples.

[1] approximately 250K for the complete library.

In Lua, OOP is typically implemented using `metatables`. In a nutshell, metatables permit refining the behavior of regular tables by redefining its standard operations such as indexing of fields, assigning new values to fields or invocation of a table. Using this mechanism, sophisticated OOP models have been realized. As uMF is primarily used to model constraints on data-structures, it does not strictly require OOP concepts. Nevertheless we use a minimalistic class model for introducing the notion of *typed Lua tables*, but which are by default agnostic about the data they contain. Further information on Lua can be found in the language reference manual (Ierusalimschy, de Figueiredo, and Celes 2012).

6.2.2 Related work

Most metamodeling today is focused on so called *language workbenches* that provide tooling for development of external DSL. An overview of these can be found here (Merkle 2010).

MOF (Object Management Group 2006) is the OMG modeling language for specifying metamodels, which can further be annotated with OCL constraints (Object Management Group a). The most widely used implementation of MOF is Ecore of the Eclipse EMF project (Eclipse Foundation b). Xtext is a language framework integrated with Ecore that supports generating external DSL and associative editors for their specification. Although these tools are only loosely coupled with Eclipse, they still inherently depend on the Java language, which constrains their application for small embedded systems.

The idea of describing entities of a programming language is not new. Bezivin (Bézivin 2005) points out the importance of distinguishing between the two relationships *instance-of* of OOP and *conforms-to* of MDE. Magritte (Renggli, Ducasse, and Kuhn 2007) is a meta-description framework that is tightly integrated into the Smalltalk object model. Magritte permits annotating Smalltalk object with metadata than can then be used for various purposes such as for creating views or editors, validation, and persistence. Like uMF the host language itself is used to specify additional constraints. The main difference is that uMF constraints are defined on data (opposed to objects) and that uMF is stronger focused on embeddability in contrast to dynamic web development.

XSD (XML Schema Definition) (W3C 2001) is a language for defining the structure of XML documents to permit their validation. Unlike uMF's Lua syntax, XSD is XML based and thus not particularly well suited for textual modeling by humans. Moreover, our approach is easily embedded by only depending on the minimal Lua language opposed to the rather heavyweight XML host language. An alternative approach is suggested by Renggli et al.

(Renggli, Ducasse, Gîrba, and Nierstrasz 2010) who observe the insufficiency of traditional static code analysis tools when applied to internal DSL. To take into account the specifics of the latter, the authors describe an approach of adding additional rules to check for common errors.

6.3 The uMF micro-modeling framework

This section gives an overview of uMF, which can be divided into two parts: types and structural specifications. The uMF source code is available here (Klotzbuecher 2012).

6.3.1 uMF Types

To attach type to Lua tables, uMF provides a minimalistic OOP model. Using typed tables facilitates both modeling and validation; The first because it permits a more explicit syntax to be used (e.g. `Vector{x=...}` instead of only `{x=...}`) and the latter because a type mismatch error is typically much clearer than an error reporting structural discrepancies. As a side effect of introducing an OOP model, single inheritance and virtual methods can be used to define pretty printing methods or constructors. The whole OO system consists only of approximately 60 lines of Lua[2].

6.3.2 uMF Specs

The core of uMF consists of the specification infrastructure which amounts to approximately 230 LOC. Using this, the (desired) structure of data can be defined by using so called specs. Specs describe primitive data (strings, numbers, booleans, enums, etc) or composite data (tables or extended types). Specs reflect the compositionality of the data they describe, hence a `TableSpec` may, analog to tables, contain primitive or composite data while primitive specs may not contain any sub-specs. Definition of new specs is straightforward and encouraged.

As Lua tables can function as both list and associative array (consequently referred to as array and dictionary), both parts are defined independently.

The spec for modeling a 3D vector using `x`, `y`, `z` fields is show in Listing 6.2.

[2] Code lines only. Calculated using the `cloc(1) tool`.

```
Vector3DSpec = umf.TableSpec{
   name = "Vector3D",
   sealed = 'both',
   dict = {
      x = umf.NumberSpec{},
      y = umf.NumberSpec{},
      z = umf.NumberSpec{},
   },
   optional={},
}
```

Listing 6.2: uMF Spec defining a 3D vector

The `Vector3DSpec` is defined as a specification of a table data-structure, hence a TableSpec. The `name` field is not strictly necessary, yet is essential for providing meaningful, localised error reporting. This becomes especially important when specs are nested or defined recursively. The `sealed`[3] field defines whether and where unspecified fields or values are permitted in the table. Legal values are `'array'`, `'dict'` or `'both'` meaning that the array, dictionary or both parts may not contain fields or values that have not been specified by the related `dict` or `array` spec fields.

The `dict` table specifies three fields `x`, `y` and `z` that must conform to the `NumberSpec`. The `Vector3DSpec` does not define an `array` field, hence this defaults to an empty table implying that no data is permitted in the array part.

Optionality of entries in the dictionary part can be specified by adding these to the `optional` list. For the vector all fields are mandatory, hence this list is empty (and the entry could be omitted altogether).

Specs can be composed into each other. Listing 6.3 shows how `Vector3DSpec` is used within a Spec describing a path. A 3D path can be defined as an ordered list of 3D points. Hence, for this spec no dictionary entries are required and only the array part is used. The `array` keyword contains of a list of entries that are legal, in our example only types conforming to `Vector3DSpec`.

```
Path3DSpec = umf.TableSpec{
   name = "Path3D",
   sealed = 'both',
   array = { Vector3DSpec },
}
```

Listing 6.3: uMF Spec defining a 3D Path

Using a spec, input data can be now be validated using the `umf.check` function. The following Listing 6.4 illustrates the checking of an invalid vector (`y` is a string, `a` is undefined in a sealed spec and `z` is missing) against the `Vector3DSpec`. The last argument `true` requests the validation results to be printed.

[3] The term sealed is inspired by srfi-76: http://srfi.schemers.org/srfi-76/

```
> v1 = { x=1, y="three", a=2 }
> umf.check(v1, Vector3DSpec, true)

checking spec Vector3D
err @ Vector3D.y: not a number but a string
err @ Vector3D.a: key 'a' has illegal value '2'
err @ Vector3D: non-optional field 'z' missing
3 errors, 0 warnings, 0 informational messages.
```

Listing 6.4: Validating models against uMF Specs

6.4 Real World DSL

6.4.1 The Task Frame Formalism DSL

The *Task Frame Formalism* (Mason 1979) (Bruyninckx and De Schutter 1996) is a hybrid force position/velocity control formalism that permits specifying robot tasks in a programmer intuitive way. Typical use cases are assembly tasks involving force-controlled operations in contact situations. The first TFF DSL developed at our lab was an external DSL called COMRADE (Witvrouw, Van de Poel, and De Schutter 1995) that was built using Lex and Yacc.

More recently, a new version was implemented as an internal Lua DSL, which simplified the implementation by removing the need for parsing and enabled new applications by permitting the TFF DSL to be composed within behavioral models such as Statecharts (Klotzbuecher, Smits, Bruyninckx, and De Schutter 2011). Yet, input models were still validated by manual programming, and that in an insufficient way: certain invalid models[4] would not be rejected and passed to the underlying control components, leading to run-time faults.

A sample TFF motion model is given in Listing 6.5.

```
compliant_slide_x = TFFMotion {
    xt = TFFAxis { value=0.1, type='velocity', unit='m/s' },
    yt = TFFAxis { value=0,  type='force', unit='N' },
    zt = TFFAxis { value=1,  type='force', unit='N' },
    axt = TFFAxis { value=0, type='force', unit='Nm' },
    ayt = TFFAxis { value=0, type='force', unit='Nm' },
    azt = TFFAxis { value=0, type='force', unit='Nm' }
}
```

Listing 6.5: TFF motion model

A TFFMotion defines how the robot end-effector is controlled while the motion is executed. A motion specification consists of a desired axis specification for

[4] e.g. declaring the unit 'Nm' for a translational force.

each of the three translational and three rotational degrees of freedom. An axis specification consists of a `type` field that defines if the degree is controlled in force or velocity control mode and a `value` field defining the quantitative desired force or velocity. Moreover a `unit` field is included to facilitate developers to use quantities appropriate for the task. The given example 6.5 defines a motion that will cause the robot end-effector to push in z-direction (down) with a force of 1N while moving (sideways) in x-direction with 0.1 m/sec.

In the following the third generation TFF DSL that uses uMF is described. To facilitate the specification of motions, two types TFFAxis and TFFMotion are introduced in the listing below.

```
TFFAxis = umf.class("Axis")
TFFMotion = umf.class("TFFMotion")
```

Listing 6.6: Type definitions

Next, the two specs for translational and rotational axis are defined in Listing 6.7. In contrast to previous examples an `ObjectSpec` is used, which is essentially a TableSpec that additionally requires the table to be an instance of the class specified in the `type` field, in our case of `Axis`.

```
rot_axis_spec = umf.ObjectSpec{
   name="rotational_axis",
   postcheck=axis_check_type_unit,
   type=TFFAxis,
   sealed='both',
   dict={
      value=umf.NumberSpec{},
      type=umf.EnumSpec{ 'velocity', 'force' },
      unit=umf.EnumSpec{ 'Nm', 'rad/s' }
   },
}

trans_axis_spec = umf.ObjectSpec{
   name="translational_axis",
   postcheck=axis_check_type_unit,
   type=TFFAxis,
   sealed='both',
   dict={
      value=umf.NumberSpec{},
      type=umf.EnumSpec{ 'velocity', 'force' },
      unit=umf.EnumSpec{ 'N', 'm/s' }
   },
}
```

Listing 6.7: Translational and Rotational axis specs

In principle one spec could be used to model both types, however by separating them the constraint between the axis type and units (e.g. a rotational axis must have the unit Nm or rad/s and a translational axis N or m/s) are expressed. This

does not however express the constraint between the control type (velocity or force) and the respective unit. This could be solved by separating the two types into four. Instead, we use a `postcheck` function, which is shown in Listing 6.8. Pre- and postcheck functions can be used to extend a TableSpec by adding further checks before or after the validation of the regular spec. These functions seamlessly integrate into the existing framework by being able to reuse the contextual error reporting infrastructure.

It should be noted that these functions can not only be used to define static checks, but also facilitate checking run-time constraints of systems. For instance, a postcheck hook could be used to ensure that the current robot is powerful enough to generate the requested force, or a too fast movement could be rejected based on the current battery level.

```
function axis_check_type_unit(class, obj, vres)
 local ret=true
 if obj.type == 'velocity' and
        not (obj.unit=='rad/s' or obj.unit=='m/s') then
   ret=false
   add_msg(vres, "err", "type velocity implies unit rad/s or m/s")
 elseif obj.type == 'force' and not (obj.unit=='N' or obj.unit=='Nm') then
   ret=false
   add_msg(vres, "err", "type force implies unit N or Nm")
  end
  return ret
end
```

Listing 6.8: Type definitions

Lastly, Listing 6.9 illustrates the composition of translational and rotational specs into the TFFMotion spec. Note that the `sealed` field is set to `'array'`, thereby permitting additional entries in the dictionary part. As an example, a TFFMotion might also carry information about specific controller settings such as stiffness or damping.

```
motion_spec = umf.ObjectSpec{
   name="TFFMotion",
   type=TFFMotion,
   sealed='array',
   dict = {
      xt=trans_axis_spec, yt=trans_axis_spec, zt=trans_axis_spec,
      axt=rot_axis_spec, ayt=rot_axis_spec, azt=rot_axis_spec
   },
}
```

Listing 6.9: Motion spec definition

A related use for `open` specs is to validate a single data-structure using multiple specs. For example, consider a spec to validate the previously mentioned `stiffness` and `damping` parameters for a KUKA Lightweight Robot, as shown in Listing 6.10.

```
kuka_lwr_motion_spec = umf.ObjectSpec{
   name="KUKA_LWR_TFFMotion",
   type=TFFMotion,
   sealed='array',
   dict = {
      stiffness=NumberSpec{ min=0.01, max=5000 },
      damping=NumberSpec{ min=0.1, max=1.0 },
   },
   optional={ 'stiffness', 'damping' },
}
```

Listing 6.10: Adding secondary constraints

By checking a TFFMotion model against both specifications, its conformance with both the generic and robot-specific constraints can be asserted. Nevertheless, both metamodels are kept separate and can be reused and recombined individually.

6.5 Evaluation and Results

Comparing both the previous hand-written and uMF based TFF DSL (excluding the KUKA LWR specific spec) reveals insignificantly less lines of code for the uMF version (46 vs. 53 LOC), obviously not counting the size of uMF itself. More importantly though, the uMF based version faithfully models all required constraints on the data, thus providing higher robustness against invalid input from users. In case of validation errors these are reported in a consistent and well-localized form. Lastly, the mostly declarative uMF based version is arguably much more readable and hence maintainable than the purely procedural original version.

Comparing the number of LOC of the uMF version with the initial Lex and Yacc based TFF DSL implementation does not make much sense, since then the internal Lua parsing code would have to be taken into account too. However one can make the following qualitative observations: firstly, development and maintenance of the complex grammer specifications for Lex and Yacc are avoided. Secondly, composition with other Lua based models (Klotzbuecher, Smits, Bruyninckx, and De Schutter 2011) becomes possible in a straightforward manner. Thirdly, the availability of a scripting language permits loose integration with further, platforms specific processing. For the TFF DSL example this is the transformation to a platform specific C++ object ready for execution by the TFF controller. Previously, this coupling took place statically at compile time.

6.5.1 The limits of untyped conforms-to

The following limitation was observed regarding the use of purely structural constraints, e.g. untyped tables in uMF. Consider the example in Listing 6.11 that extends the previously shown `3DPathSpec` to permit a path to be formed by both 3D and 6D vectors, that are described using untyped TableSpecs.

```
PathSpec = TableSpec{
   name = "Path",
   sealed = 'both',
   array = { Vector3DSpec, Vector6DSpec },
}
```

Listing 6.11: Example of difficult to validate purely structural model

To validate a vector entry of a concrete path, this entry needs to be checked against each array spec (`Vector3DSpec` and `Vector6DSpec`) until a spec that successfully validates the entry is found. The problem is that if all checks fail, it becomes impossible to provide a concise error message, since uMF can not know the TableSpec against which validation should have succeeded. Since this is an inherent problem of the pure, untyped conforms-to relationship, the only solution is to print the errors of all checks, which is not user friendly and verbose. The problem is worsened if the PathSpec array is unsealed, as in that case it is impossible to know if this is an error at all. On the other hand these problems can be avoided altogether by using typed tables when disjunctive matching of multiple specs takes place.

6.6 Conclusion and Outlook

We have presented a lightweight, declarative DSL for specifying structural and composable constraints on data-structure of the Lua language. This meta-DSL can be easily extended and integrated into a runtime environment by using additional hooks. Using the Lua language yields a syntax that is reasonably understandable to humans and moreover permits the DSL to be deployed into very constrained embedded systems, such as robot or machine tool control panels. Since open uMF specs can be used to specify partial *conformance* constraints on data, it becomes possible to use multiple specs to validate different aspects of a single model, thereby essentially permitting virtual constraint composition.

Currently, the main use of uMF specs is to validate user models and provide indicative error messages, however further uses such as view generation or model persistence are conceivable. This could be achieved by extending specs to carry additional meta-data, such as which fields shall be displayed or persisted.

Furthermore, we plan to loosen the currently tight coupling to the Lua language by providing a horizontal, heterogeneous transformation (Mens and Van Gorp 2006) of spec and model files to the JSON (Crockford 2006) format. An example of this using the Vector3DSpec showed earlier is given in Listing 6.12.

```
{
   "name": "Vector3D"
   "sealed": "both",
   "dict": {
      "y": "NumberSpec",
      "x": "NumberSpec",
      "z": "NumberSpec",
    },
    "optional": {},
}
```

Listing 6.12: Vector3DSpec represented in JSON

Assuming the absence of functions in the model (such as pre- and postcheck hooks) the transformation to JSON can be trivially implemented in a bi-directional manner. When permitting functions, the transformation from JSON to Lua is still feasible by representing functions as strings that are evaluated after the transformation. Providing an alternative, Lua independent input syntax would permit uMF to be used as a lightweight, standalone model checker.

Chapter 7

Pure Coordination using the Coordinator–Configurator Pattern*

7.1 Abstract

We report on our efforts to improve different aspects of *coordination* in complex, component-based robotic systems. Coordination is a system level aspect concerned with commanding, configuring and monitoring functional computations such that the system as a whole behaves as desired. To that end a variety of models such as Petri-nets or Finite State Machines may be utilized. These models specify actions to be executed, such as *invoking* operations or configuring components to achieve a certain goal.

This traditional approach has several disadvantages related to loss of reusability of coordination models due to coupling with platform-specific functionality, non-deterministic temporal behavior and limited robustness as a result of executing platform operations *within* the context of the coordinator.

To avoid these shortcomings, we propose to split this "rich" coordinator into a *Pure Coordinator* and a *Configurator*. Although the coordinator remains in

*This chapter is based on Klotzbücher, M., Biggs, G., and Bruyninckx, H. Pure Coordination using the Coordinator–Configurator Pattern. In *Proceedings of the 3rd International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob-12)*. SIMPAR 2012. Tsukuba, Japan (November 2012).

charge of commanding and reacting, the execution of actions is deferred to the Configurator. This pattern, called *Coordinator–Configurator*, is implemented as a novel Configurator domain specific language that can be used together with any model of coordination. We illustrate the approach by refactoring an existing application that realizes a safe haptic coupling of two youBot mobile manipulators.

7.2 Introduction

The context of this work (and hence also for the described pattern) is complex, component-based robotics and machine tool systems operating under real-time constraints. For building such systems, an increasingly acknowledged best practice is to separate the concerns of Coordination, Computation, Configuration and Communication (Radestock and Eisenbach 1996; Prassler, Bruyninckx, Nilsson, and Shakhimardanov 2009).

Computation defines the basic, functional building blocks from which a system is constructed. Communication defines how and with whom the individual elements of a system communicate. Configuration defines the properties of a system . Lastly, Coordination is concerned with supervising and monitoring the computations in way that the system as a whole behaves as intended.

Classical coordination models that have been used in robotics are Petri-Nets (Rosell 2004), Finite State Machines (FSM) (Finkemeyer, Kröger, and Wahl 2005) and Statecharts (Marty, Sahraoui, and Sartor 1998; Klotzbuecher, Smits, Bruyninckx, and De Schutter 2011). These models are used to define when certain behaviors shall be executed. For instance, UML state machines (Object Management Group 2011) allow execution of a behavior upon entering or exiting a state. The exact actions available depend on the primitives of the underlying framework, and may include *invoking* operations or modifying the configuration of a component.

This traditional approach has three major disadvantages. Firstly, reusability of coordination models is reduced because the model is polluted with platform specific information. In other words, reusing the same model on a different robot or software framework requires intrusive refactoring to replace the platform specific operations used in the coordinator. Secondly, the blocking invocation of operations on functional computations can severely degrade the determinism of the Coordinator. Lastly, Coordinator robustness is reduced since an invocation might block indefinitely or crash, either of which may effectively render the coordinator inoperative.

Figure 7.1: Bidirectional youBot coupling demonstration: each robot copies the cartesian position of its peer robot.

This chapter proposes the *Coordinator–Configurator* pattern to overcome these challenges. The Configurator has been implemented as a Lua (Ierusalimschy, Celes, and de Figueiredo 2012) based internal domain specific language (DSL) for the Orocos Real Time Toolkit (RTT) framework (Soetens 2006), and is to be complemented by a Coordination model such as rFSM (Klotzbuecher 2011). We have applied a preliminary version to a moderately complex application[1] consisting of a haptic, force-controlled coupling of two KUKA youBots in which multiple constraints are monitored and their violation is reacted to by the coordinator.

7.2.1 Prior usage

From an object oriented software engineering perspective, the classical *command* design pattern (Gamma, Helm, Johnson, and Vlissides 1995) comes close to our suggestion by permitting a *client* to request an *invoker* to execute a given *command* on a *recipient*.

The ROS (Quigley, Conley, Gerkey, Faust, Foote, Leibs, Wheeler, and Ng 2009) framework provides the actionlib library as a standardized protocol to define commands that can be executed, monitored, aborted, etc. Hence, it is not a

[1] This demo was shown at the Automatica trade fair 2012 in Munich.

form of Configurator, but rather a mechanism that could be used to implement one.

At higher and non-realtime abstraction levels, the described pattern has been used before in the domain of task or plan management systems (Botelho and Alami 1999; Lesser, Decker, Wagner, Carver, Garvey, Horling, Neiman, Podorozhny, Prasad, Raja, Vincent, Xuan, and Zhang 2004; Gancet, Hattenberger, Alami, and Lacroix 2005; Joyeux, Philippsen, Alami, and Lacroix 2009), though generally little detail is provided on this aspect. Joyeux et al. (Joyeux, Philippsen, Alami, and Lacroix 2009) have implemented a Configurator as a plugin for the Ruby based Roby DSL in the context of the ROCK project (DFKI 2011).

Since this work is concerned with constructing modular subsystems, the work of the Ptolemy project (Eker, Janneck, Lee, Liu, Liu, Ludvig, Neuendorffer, Sachs, and Xiong 2003) is relevant, although that is more focused on composition of heterogeneous systems.

7.2.2 Outline

The remainder of this chapter is structured as follows. The following section describes the Coordinator–Configurator pattern in detail and introduces the configuration DSL that underlies the Configurator. Section 7.6 critically examines the solution and discusses further potential uses of the DSL. Section 7.7 concludes and describes future work.

7.3 Approach

To overcome the described shortcomings, we propose to split the “rich” coordinator into a *Pure Coordinator* and a *Configurator*, named the *Coordinator–Configurator* pattern. Although the coordinator remains in charge of commanding and monitoring, the execution of actions is deferred to the Configurator. The Configurator is realized as a software entity (typically a component), that is configured with a *set of configurations*. Each configuration describes one possible state of the system and is identified by a unique name. Furthermore, a Configurator has a mechanism to receive events. When an event that matches the ID of a configuration is received, the Configurator *applies* the respective configuration (details on this application follow below). Success or failure is reported back via a status event (`e_<id>_OK` or `e_<id>_ERR`

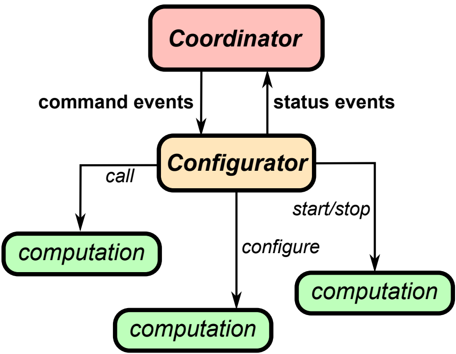

Figure 7.2: The relationship between the Coordinator, the Configurator and Computational components.

respectively), permitting the Coordinator to react appropriately. Figure 7.2 illustrates this mechanism.

Note that Figure 7.2 omits the important and complementary concept of a monitor, which is responsible for observing the system and generating events when certain conditions are met or violated.

7.4 Example

Figure 7.3 shows the coordination statechart that is executed on each of the two youBots of the coupling application. Figure 7.4 shows a (slightly simplified) component architecture; straight lines represent data-flow communication and zigzag lines represent status events emitted by computational components and received by the coordinator. This application is used as a running example throughout this article. It has been converted to the Coordinator–Configurator pattern.

The basic behavior of the demo is immediately visible from the statechart. It should be noted that the handling of error events has been omitted for simplicity; in the simplest case safe behavior can be realized by adding all possible Configurator error events to the transition from `copying` to `harmonizing`, that way ensuring that the coupling will be undone in the event of Configurator errors. At the toplevel, the `unsync` (unsynchronized) state is entered by default and signifies that communication with the peer robot is not functional. After communication is established and its QoS is sufficient, a transition to the `sync` (synchronized) state takes place, and from there to the `harmonizing` state. The

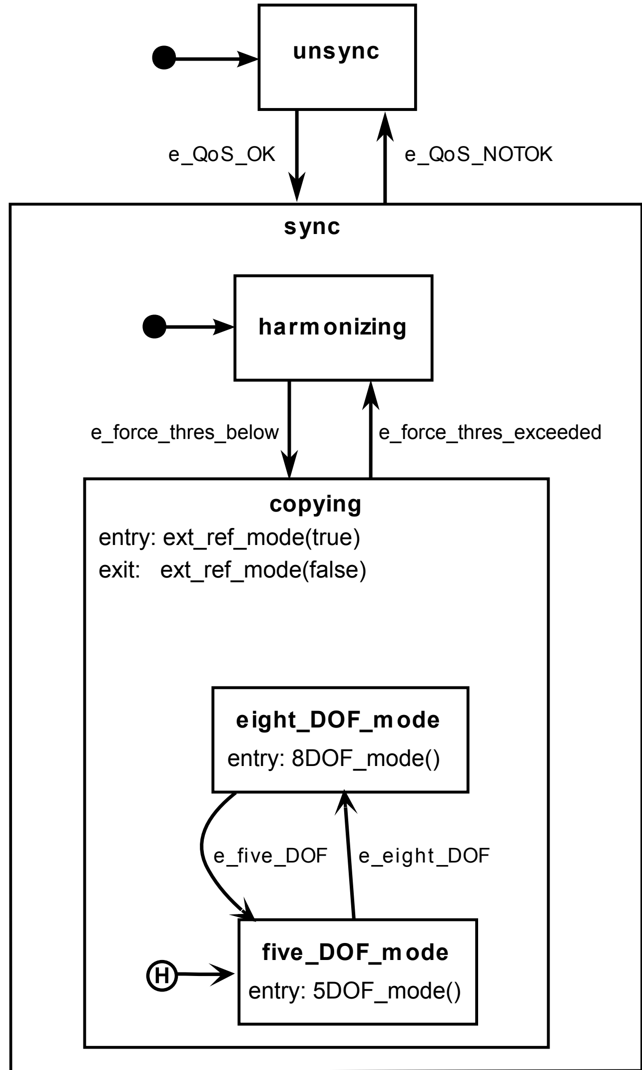

Figure 7.3: rFSM Coordination Statechart for the youBot coupling demo.

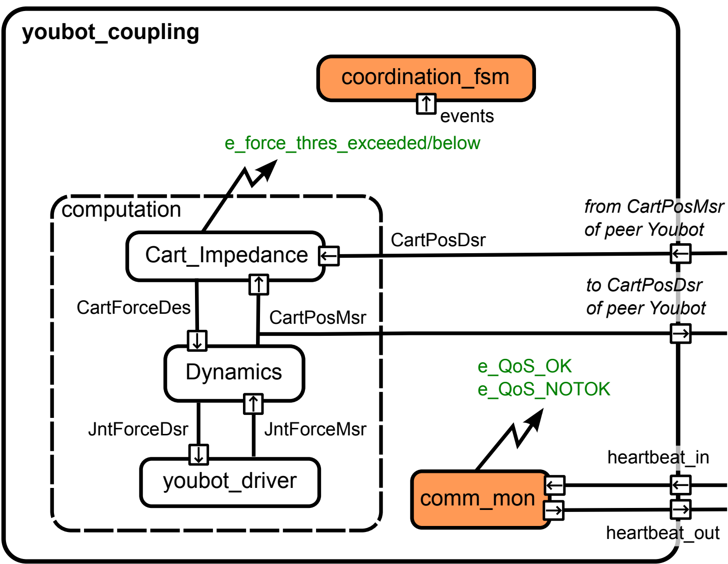

Figure 7.4: Component diagram illustrating the components and connections running on each of the youBots in the example.

latter means that the control loop responsible for moving the end-effector of the robot towards the position of the peer robot is operational and the impedance controller `Cart_Impedance` (see Figure 7.4) is computing a desired-force control signal. However, since this force is too high, this controller is configured to not output the signal but instead output a desired force of zero in all directions (`ext_ref_mode(false)`). As a result, the robot arm remains controlled in a compliant way, merely compensating for gravity. Only when both arms are (more or less) manually aligned by the operator does the desired force fall beneath the threshold, and thus trigger the transition to the `copying` state and from there (depending on the configuration) to the `eight_DOF_mode` or `five_DOF_mode`. In either case, the coupling is put into effect by requesting the controller to output the control signal `ext_ref_mode(true)`. Switching between the `eight_DOF_mode` and `five_DOF_mode` states is controlled by the human operator, and reconfigures the `Dynamics` component to use the holonomic base as additional degrees of freedom or not. It should be noted that support for history connectors has not yet been implemented in rFSM; the identical behavior is realized by using additional transitions and guard conditions.

7.5 Modeling configuration and its application

An important question is how to model a configuration and what the semantics of applying it are. Obviously, a configuration has to be able to express the necessary platform-specific changes required for runtime coordination. For the youBot coupling example, the following Orocos RTT-specific primitives are sufficient:

- Changing the state of a component (e.g. from `running` to `stopped`)
- Modifying a property of a component
- Writing a value on a port.

One of the most fundamental design choices is whether to choose a declarative or procedural model to express the behavior of applying these constraints. In a classical coordination model like FSM, the configuration applied when entering one Coordinator state is typically defined in a procedural fashion by using a function that executes several statements. However, in most cases this approach constrains the execution more than is necessary, as there often only exists a partial ordering requirement of the execution of these statements. This accidental introduction of constraints is undesirable, since it obscures the true requirements of the system and hinders maintenance. Thus, in our approach we opted for a purely declarative model of a configuration (apart from a minor deviation described below). This purely declarative approach becomes possible because, outside of the scope of the Configurator, the Coordinator can express ordering requirements by coordinating the Configurator to apply a series of configurations.

Listing 7.1 shows a single sample configuration written in the Lua based Configurator DSL. A configuration consists of a `pre_conf_state` and a `post_conf_state` specification and a list of configuration changes. The first two define to which (runtime) state components shall be brought before and after the actual configuration takes place, while the latter list defines the exact (platform specific) changes to be applied. Note that with respect to ordering of configuration application, the only guarantee made is the following: the run-time states of the components mentioned in *pre_conf_state* and *post_conf_state* are set accordingly and in the defined order before resp. after the list of changes is applied. However, no assumptions can be made about the order of applying the individual changes themselves. The `_default` keyword permits changing the state of all components that have not been mentioned otherwise. If no `_default` statement is provided, the state of the unmentioned components is not changed.

```
Configuration {
    pre_conf_state = { 'compA:running', 'compB:configure',
                       '_default:stopped' },

    post_conf_state = { _default='running' },

    property_set("compA.prop1", { 2.3, 3.4, 5.34 } ),
    port_write("compB.portX", 33.4),
    operation_call("compG.op1", arg1, arg2,...),
}
```

Listing 7.1: A sample configuration.

```
ConfiguratorConf {
   disable_copying = Configuration{
       port_write("Cart_Impedance.ext_ref_mode", false)
   },

   enable_copying = Configuration {
       port_write("Cart_Impedance.ext_ref_mode", true)
   },

   eight_DOF = Configuration {
      property_set("Dynamics.force_gain", {0.1, 0.1, 0.1})
   },

   five_DOF = Configuration {
      property_set("Dynamics.force_gain", {0, 0, 0})
   },
}
```

Listing 7.2: Named configurations for the youBot sample.

Using this DSL, the system configurations required by the coordinator of Figure 7.3 can be modeled as shown in Listing 7.2.

Since state changes from `unsync` to `harmonizing` do not involve any actions, but merely model the constraints that must be satisfied before the coupling can be activated, these are not visible in the Configurator configuration. The only configurations necessary are for enabling and disabling the coupling (hence to be applied in `entry` and `exit` of copying respectively) and for switching between eight and five degrees of freedom.

7.6 Discussion

We have implemented the described DSL and the associated configurator for the Orocos RTT framework. The existing youBot coupling coordination has been

refactored to make use of the new Configurator. This approach has solved the shortcomings of the traditional approach: firstly, the coordination model remains free of any software platform-specific actions and can be reused with any other framework, assuming a Configurator and corresponding configuration. Secondly, the actual changes are applied by the Configurator while the Coordinator remains reactive and free to deal with any other situation that may arise. Lastly, failures within the Configurator are isolated from the Coordinator, permitting it to react to the absence of a status event and ultimately improving its robustness. Naturally, this robustness depends on the run-time context of both entities and the worst-case failures possible.

Since the Configurator constitutes an additional level of indirection, the question of the overhead introduced is justified. Obviously, this cannot be answered in general but will mostly depend on the level of distribution between Coordinator, Configurator and configured components. Nevertheless, the Coordinator–Configurator pattern offers an advantage with respect to benchmarking and profiling the Coordination behavior: since the execution of configuration application is localized within a single component, the respective measurements need likewise only to be added once. This avoids the scattering of profiling code across the Coordination model.

Lastly, having a flat map of configurations might be suboptimal in some cases, as for instance when it is necessary to undo a configuration to return to a previous one. With the current model it is necessary to manually specify the *inverse* configuration, as for instance is the case for enabling and disabling the coupling in Listing 7.2. One solution to this could be to use a stack of configurations onto which changes can be pushed (applied) and popped (undone) again.

7.6.1 Deployment

Interestingly, the Coordinator–Configurator pattern allows dealing with deployment as a special case of Coordination and Configuration. To that end, the only requirement is to extend the set of configuration actions with the following primitives:

- Creating components
- Destructing components
- Creating connections between components
- Removing connections between components.

That way, deployment can be viewed as coordinating the system through a series of configurations that culminates with the system having reached an initial operational state. Likewise, shutting down the system can be defined as applying a configuration that stops all components followed by one resulting in their destruction. A system, including its rules for deployment, starting up, runtime changes, and shutdown can thus be specified in terms of a single coordination model (which can, itself, be composed from multiple coordination models) and a platform specific Configurator configuration.

7.6.2 Composition

The approach promises to greatly facilitate composition of systems from systems. Any valid pair of Coordinator and platform-specific Configurator configuration can be treated as a subsystem (sometimes called a composite component) that can be used as a building block in a larger system. Nevertheless, for this to work, several questions need to be answered, including: what is the interface a subsystem offers, how can the contained coordination model be controlled from the "outside", and how can the controllable transitions be specified. Answering these questions is outside the scope of this chapter.

7.7 Conclusions

We have described the Coordinator–Configurator pattern that is applicable to complex component-based robot systems. The pattern's goal is to balance the forces between increased reusability, temporal determinism and robustness on the one hand and simplicity of the Coordinator on the other. The key idea is to separate the responsibility for commanding actions from the responsibility for executing them. While the first remains with the Coordinator, the latter is assigned to a new entity called the Configurator. The idea has been implemented as a Configurator DSL for the Orocos RTT framework.

In future work we intend to focus on adding a complementary DSL to describe the monitor, whose description was omitted in this work, to further explore the outlined relationship between Coordination/Configuration and deployment, and to validate the hypothesis that this pattern greatly facilitates specifying platform-independent composite components.

Chapter 8

Conclusions

This chapter concludes the thesis by summarizing and discussing the main contributions from Chapters 3-7 and making suggestions for future work.

8.1 Contributions

The goal of this thesis is to show how robustness, reliability and reusability of robotic systems can be improved through application of domain specific languages. To that end **an approach of constructing executable, internal domain specific languages is presented, which offers the advantages of rapid DSL construction and evolution, facilitating run-time composition and transformation of DSL while improving robustness through execution in a virtualized environment.**

Hard real-time safe scripting and DSL execution

Chapter 3 describes how the Lua programming language was extended in order to satisfy real-time constraints. The principal challenges are dealing with safe allocation and recuperation of memory. The allocation problem was resolved by employing a dedicated, constant time allocation algorithm (Ogasawara 1995; Masmano, Ripoll, Balbastre, and Crespo 2008). Coping with the challenge of real-time garbage collection has proven more difficult, and despite of being researched for years (Pizlo and Vitek 2008; Kalibera, Pizlo, Hosking, and Vitek 2011), no generic solution has been found. Instead, **the presented approach**

exploits domain knowledge to schedule garbage collections such as to avoid interfering with critical paths.

Models of coordination, methodology and patterns

Chapter 4 introduces the rFSM statechart model, which was developed to **model robust and reusable coordination of complex robotic systems with a minimal and extensible formalism** and which is **suitable for integration with legacy systems**. The rFSM semantics are derived by analyzing state-of-the-art discrete coordination models and extracting a subset with well-defined semantics for the purpose of Coordination. **A real-time safe reference implementation as an internal Lua DSL is provided**, and its integration into the Orocos RTT framework described. **Furthermore, a methodology to derive coordination for a given architecture of functional components is presented**. **Several usage patterns and extensions are described that illustrate how to integrate rFSM with existing platforms and how the execution semantics can be extended**. An example for the latter is the **preview coordination** mechanism.

First experiments with rFSM based coordination confirm the expected reusability (Chapter 5) **and the feasibility of modeling multi-robot coordination using distributed rFSM instances** (see 4.11).

Reusable Task Specifications

Chapter 5 investigates how industrial robot tasks can be specified robot and software framework independently. For that purpose **a simple alignment skill is realized on two different robots**, a Willow Garage PR2 and a KUKA LWR. The required reusability is achieved by separating the following aspects: the platform independent rFSM coordination and motion control models, the robot specific parameters and the software framework specific aspects. Upon loading, these aspects are dynamically recomposed into a single, executable rFSM instance (see also Section 4.10.2 for an extended version). **Thus, the goal of platform independent task specification was achieved**. The analysis of the resulting lines of code shows that approximately twice the amount of platform specific code is necessary to support a reusable, platform independent model.

Moreover, this chapter has demonstrated the suitability of the approach to **integrate legacy systems**. This was achieved by showing how a robot agnostic

task FSM was composed at run-time within the FSM for coordinating the KUKA LWR FRI platform in a way to enforce the constraints of the latter.

Constraining internal DSL

A further outcome of the work on the Task Frame DSL (Chapter 5) was the insight that internal DSL models are often insufficiently validated, which may result in invalid behavior or run-time failures. Chapter 6 introduces **the uMF constraint language which addresses the lack of rigorous formalization of internal DSL**. uMF is itself a DSL for modeling constraints on internal Lua DSL. Although similar to existing constraint languages such as XML Schema (W3C 2001), uMF differs in several ways. Firstly, **the language is lightweight and as such suitable to be embedded** within a single software component or a small embedded system such as a robot control panel. Secondly, the language can be **extended using hooks, which permits taking the actual system state into account during the validation**. For example, this could be used to reject the execution of a motion due to a low battery level or the current joint configuration. Lastly, **uMF introduces the notion of open models, which are realized by defining constraints that only enforce partial conformance**. This way a model can be checked using multiple constraints, of which each validates a certain property of the model. For example, a task model could be first checked for conformance with the generic task meta-model, and then secondly if it can be executed by a particular robot.

Coordinator-Configurator pattern

In contrast to the patterns introduced in Chapter 4, which illustrate solutions to problems of specifying and integrating coordination, **Chapter 7 presents the Coordinator-Configurator architectural pattern**. This pattern suggests decoupling the coordination specification from the concrete execution of actions. A reference implementation realizes the pattern as follows: the Coordinator emits events to command a Configurator to bring the system to a specific state, and receives status events from the Configurator and the rest of the system. The Configurator is configured with event-configuration pairs; upon receiving an event the corresponding configuration is applied to the system. **This separation offers the following advantages. The robustness of the coordinator is increased, since the execution of actions and associated failures are separated from the coordination logic. The temporal determinism of the coordinator is improved, since the**

actions executed are restricted to sending event messages. Moreover, the approach also facilitates specification of reusable coordination, since software framework specific operations are contained within the coordinator.

8.2 Discussion

The following critically discusses the extent, to which the initially formulated requirements of improving reusability, reliability and robustness of complex and real-time robotics systems is achieved.

Reusability

Task reusability is achieved by run-time composition of multiple concerns (task model, coordination model and software framework and robot dependencies) into a single model, that way demonstrating the possibility to reuse each concern separately. In the first experiments related to reusability (see Chapter 5), only the coordination statechart was formally modeled, while the activity of each state was added as an opaque function. The limitation of this approach is that validating the composition is virtually impossible, since the model underlying the opaque is not available. Section 4.10.2 shows how this shortcoming can be overcome by validating and transforming a model-only composition to executable code. Nevertheless, while this approach technically permits composing arbitrary DSL models, ensuring logical correctness still depends on the developer to select models that *can* be composed. Overcoming this limitation will not only require richer models, but more importantly standardization to ground different DSLs in a way that a common interpretation of primitives is achieved. For example, the work on geometric relations by De Laet et al. (De Laet, Bellens, Smits, Aertbeliën, Bruyninckx, and De Schutter 2012; De Laet, Bellens, Bruyninckx, and De Schutter 2012; De Laet, Bellens, and Bruyninckx 2012) is an important step in that direction. With a broader scope, the European FP7 projects Rosetta (ROSETTA) and RoboHow (RoboHow) are addressing these knowledge engineering challenges by developing robotics ontologies.

Reliability and Robustness

The presented approach improves reliability and robustness of complex robot systems in several ways. By formulating problems and solutions using the terminology of the domain, DSL offer more concise semantics than general

purpose languages. This supports domain experts to maintain an understanding of the specified behavior while avoiding distraction by programming language syntax and semantics. Through its reduced and more concise language, rFSM avoids several pitfalls and variation points found in UML state machines (see 4.5 and 4.6), such as `Choice` elements without `else` clauses (4.5.3). Moreover, making the concern of coordination explicit often reveals hidden assumptions in system architectures (see Section 4.2.1).

The Coordinator–Configurator pattern is an architectural pattern which enforces explicit coordination (see Chapter 7) and improves reliability by deferring the potentially unsafe and non-deterministic execution of actions to a dedicated Configurator. This pattern is complemented by the approach of introducing constraint monitoring as dedicated monitors or as extensions to computational components (see Section 4.11).

For many internal DSL, input validation is implemented manually and insufficiently. Consequently, invalid models may pass validation and cause run-time failures. The uMF DSL permits to avoid this through rigorous checking of structural models (see Chapter 6), that way increasing the reliability of internal DSL.

rFSM adopts the concept of structural priority (Section 4.3.2) and illustrates how this concept can be employed to improve reliability and robustness for different robotic use-cases (see Section 3.5.4 for an example of memory monitoring in real-time scripting, Section 4.6.4 for an example of a prioritized safety transition and Section 5.3.4 for an example of enforcing robotic platform constraints). The limitation of structural priority for the rFSM implementation is that the transition latency depends on the execution time of `entry`, `do` and `exit` functions. In other words, an erroneous sub-statechart may still impair reliability of coordination. However, if deterministic behavior of these functions can not be ensured, the alternative is to split both safety and task FSM according to the distributed sub-state pattern (see Section 4.10.5). That way, negative effects on transition latency caused by a nested statechart are reduced, though at the price of additional communication overhead.

In addition, this approach of hierarchically decomposing distributed, loosely interacting, single threaded rFSM statecharts contributes to reliability, since failures are contained to individual instances. This way, a system can continue to function gracefully in spite of local failures. Moreover, the suggested event-only communication between coordinators behaves robustly in case of communication failures, since i) no blocking operations are involved that might deadlock and ii) this condition can be easily detected by communication monitoring (see Section 4.11).

A current limitation is the lack of tools to specify, validate, launch and monitor compositions of distributed rFSM. Tools for deploying component based systems may be of some use, however the subtle differences of coordination (see the multi-writer example in Section 4.12) will required coordination specific or at least coordination aware tools.

Currently no formal methods are employed to verify rFSM based coordination. One reason for this is that formal methods tend to require rigorous application, which may not be feasible or costly, if existing subsystems must be reused. For this reason, rFSM based coordination is designed to be easily integrated with legacy systems and subsystems. On the other hand, statecharts have been verified using formal methods (Borland 2003; Mikk, Lakhnech, Siegel, and Holzmann 1998; Zhao and Krogh 2006), thus it can be assumed that much of this work is applicable to the rFSM model.

Interpreted Real-Time DSL

A limitation of the presented approach is that the responsibility of scheduling the garbage collection is placed on the developer. To that end it is not only necessary to have an understanding of the required time constraints, but also to *develop* an understanding of the actual timing by means of appropriate profiling and to ensure that the chosen schedule is sufficient to keep up with memory allocations. On the one hand, this requires additional effort. On the other hand, one could argue that such insights are essential for building a hard real-time system anyway, and that the use of a scripting language merely facilitates this process by supporting on-the-fly instrumentation and self-monitoring of memory use. Thus, choosing between real-time scripting and traditional compiled languages involves a trade-off between improved reliability and flexibility at the price of requiring some manual profiling and tuning, versus less flexibility and reliability but with higher performance.

For future work it may be considered to automate the scheduling of the garbage collection entirely. This would become possible with a sufficiently rich task model, from which the task's real-time constraints can be derived. As an example, coordinating a free space motion will impose weaker timing constraints than when coordinating a switch of controllers upon making contact with the environment.

Implementation Strategy

An important consideration during early stages of this work was whether to follow the approach of external DSL and to *generate* code from models or to employ internal DSL whose models can be *executed* instantaneously (the terminology is introduced in Section 2.2). This choice involved the following trade-off. The former approach is supported by a wide range of existing modeling tools (mostly developed in the context of the Eclipse project (Eclipse Foundation a)), permits full control over DSL syntax and is applicable to hard real-time systems. On the other hand, the approach of modeling using internal DSL facilitates composing, transforming, extending and executing models, though at the price of less freedom to define the DSL syntax and of requiring additional effort to achieve real-time safety. In retrospect, opting for internal DSL has been the right choice, since it has greatly facilitated illustrating concepts *and* their practical implementations without spending much time to develop dedicated tooling. Realizing non-intrusive, optional extensions such as rFSM preview coordination (see section 4.10.7) would have required significantly more effort with the external DSL approach.

Furthermore, the internal DSL approach has opened the possibility for run-time adaptation of models, such as replacing a state of a running rFSM instance. So far this has only been little exploited, however for systems with high availability requirements this may be an important prerequisite.

Language-wise, the author personally would have preferred the Scheme language as a DSL host language. However, for the purpose at hand Lua was selected since it combines several of the characteristics of Scheme relevant to building DSL with a more friendly syntax suitable for non-expert programmers, which are not uncommon among roboticists. Further reasons included the high maturity of the language and the large and supportive community surrounding the project.

Minimality

A further point worth discussing is related to the minimality procured in the rFSM semantics. While the problem of adding too many primitives or features is a well known and notorious stumbling block (“feature creep”, (Sullivan 2005)), there is also the more subtle danger to over-reduce complexity. In “Epigrams on Programming” (Perlis 1982) Alan Perlis warns of minimality: *“Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy”*. For example, it is worth noting that the rFSM model and reference implementation contain no robotic specific features. This may seem surprising for a DSL targeting robotic systems, as the inclusion of robotics specific features

might appear to encourage adoption by robotics system engineers. Here the opposite position is taken: including such features in rFSM would hinder the adoption, since any selection of features would be inevitably biased towards certain use-cases. The solution to this is to provide a simple and extensible model that adopters can mold to their particular use-cases. Patterns and best-practices are intended to support this process. For example, the notion of *time* is intentionally not included into the rFSM core semantics, since its quality (accuracy, resolution) is highly dependent on the underlying operating system platform. Consequently when time is required (e.g. for TimeEvents), it must be *explicitly* configured. This approach has permitted rFSM to be applied to a broad range of use-cases, ranging from communication monitoring to assembly task specification.

All in all, the experience gained so far within the Orocos community and at several BRICS research camps (most importantly Camp III on Control and Coordination in Bertinoro, Italy) indicates that even developers not familiar with coordination can easily get started with rFSM, while at the same time avoiding common UML pitfalls. Nevertheless, designing a modeling language remains to some extent inevitably a matter of taste.

Relevance for Machine Tools Applications

This work refers infrequently to machine tools, given that this domain is mentioned in the title. On the one hand, this is because in terms of software in the context of this thesis, both machine tool software systems as well as advanced industrial robot systems are very similar: both system often exhibit hard real-time constraints and both have strong requirements with respect to reliability and safety. Furthermore, the connection becomes visible in Section 4.3.5, where IEC 61131-3 Sequential Function Charts, which are commonly used in programmable logic controllers, are discussed. A first step in the direction of demonstrating applicability of this approach to the domain of machine tools has been converting a shock absorber testing simulation framework to rFSM based coordination together with the Flanders' Mechatronics Technology Center. One interesting possibility for future work would be realizing one or more classical PLC programming languages as real-time Lua DSL.

With respect to certification according to IEC61508 (International Electrotechnical Commission 2010), it is important to distinguish this thesis' contributions with respect to models and patterns on the one hand, and reference implementation on the other. While the latter aims to cover the broadest range of use-cases, the scripted implementation may prove difficult or impossible to

certify. In contrast, no such limitation applies to the models and patterns presented in this work.

Reproducibility

Each main chapter of this work is backed up with a software implementation. To encourage reproducibility of the presented results, these implementations were released together with the respective papers (corresponding to the chapters), apart from Chapter 7 for which the implementation was released shortly after the publication. For convenience, the following summarizes the software corresponding to each chapter:

- Chapter 3: Real-time scripting benchmarks
  http://people.mech.kuleuven.be/~mklotzbucher/2011-09-19-rtlws2011/source.tar.bz2
- Chapter 4: rFSM Statecharts
  https://github.com/kmarkus/rFSM
- Chapter 5: TFF DSL, Platform and Coordination Models
  https://github.com/kmarkus/tff_dsl_tests
- Chapter 6: uMF Constraint DSL
  https://github.com/kmarkus/uMF
- Chapter 7: Coordinator–Configurator DSL
  https://bitbucket.org/kmarkus/dng

The RTT-Lua integration has been merged into the official Orocos toolchain project (see http://www.orocos.org). Furthermore, a prototype of a development tool has been released (see Appendix A).

8.3 Impact

rFSM and RTT-Lua were successfully used at several of the BRICS research camps, most notably at the third research camp on Control and Coordination 2011, in Bertinoro, Italy.

The current, second generation iTaSC framework is based on rFSM/RTT-Lua for modeling and executing coordination at system and task level (Vanthienen, De Laet, Smits, and Bruyninckx 2011). To facilitate specifying iTaSC based systems, a textual DSL based on the uMF language is currently being developed.

As of today, the rFSM implementation and the Orocos RTT-Lua integration are widely used throughout the Orocos community, judging by the number of different users asking questions and providing feedback in the forum. In response to a query on the Orocos mailing list on December 21 2012, members of several research facilities reported using rFSM and/or RTT-Lua, including the ARD Team[1], the Engineering Department of the University of Ferrara, the Systems Control and Flight Dynamics Department of the French Aerospace Lab ONERA (more specifically in the research described in Chanel et al. (Chanel, Teichteil-Königsbuch, and Lesire 2012) and Gateau et al. (Gateau, Lesire, and Barbier 2012)) and the Robonaut 2 team from the NASA Johnson Space Center in Houston.

The successful ContainerBot ECHORD[2] project carried out by KU Leuven and Intermodalics BVBA[3] made use of rFSM to coordinate the bin picking system. Besides this, Intermodalics has realized several industrial automation systems using rFSM coordination and RTT-Lua real-time scripting (reported in private communication).

8.4 Suggestions for Future Work

Standardization

To reuse robotic models and meta-models on a wider scale, significant standardization efforts will be necessary to define common terminology and to formalize the semantics of primitives in a way that permits different DSL to unambiguously use these. As mentioned in Section 8.2, several efforts, mainly driven by academia, are currently ongoing. Unfortunately, as of today the robotics industry is still showing little interest in standardization of both software and hardware interfaces. As a consequence, the exchange and reuse of robotic models, software and hardware components on a broader scale remains an unsolved challenge.

Formal Verification of Distributed Statecharts

A related topic for future work is to explore how formal approaches can be applied to validate and verify compositions of rFSM statecharts. For example, it may be useful to determine when the loss of an event message may lead to a

[1] http://team-ard.com
[2] http://www.echord.info
[3] http://www.intermodalics.eu/

logical dead-lock. The main challenges to be solved are expected to be related to the unreliability and non-determinism of the interconnecting communication channels, as well as unmodeled behavior of computational components.

Tool Development

The efficient application of many techniques described in this thesis will require further tool support. For example, efficiently realizing discrete preview coordination with an horizon of multiple states (see Section 4.10.7) will require tools to reason about whether two task models are kinematically in conflict or not. Only that way the preview horizon can be automatically determined and the developer freed from the error prone task of manually specifying this information.

The deployment, monitoring and validation of distributed rFSM statecharts could be greatly facilitated by tools for specifying and visualizing the composition. Two basic prototypes (see Appendix A) have been developed, yet these only deal with individual rFSM instances, not compositions.

Appendix A

Tools

This appendix describes two tool prototypes, which are being developed with the goal of supporting development and debugging of rFSM based coordination.

A.1 rFSM online visualization

This tool connects to a running rFSM instance, which is specified by host address and port. For this to work, the rFSM protocol plugin (`rfsm_proto`) must be loaded in the respective rFSM instance. This plugin realizes a server that offers a message oriented interface to the running FSM. Clients register as subscribers that (when accepted) initially are sent a rFSM model message. After that, a continuous stream of state update messages is sent.

Upon receiving the rFSM model, the visualization tool will compute a layout and visualize the statechart. From then on the currently active state is highlighted, based on the stream of messages reporting changes of the active configuration. A screenshot is shown in Figure A.1.

Although functional, this tool has not been released because the current layout algorithm is an ad-hoc implementation that does not scale to larger statechart models. Unfortunately, most graph drawing libraries currently only support cluster graphs, for which edges can only connect to leaf vertices but not to cluster vertices. The latter is frequently the case in graphs representing hierarchical statecharts.

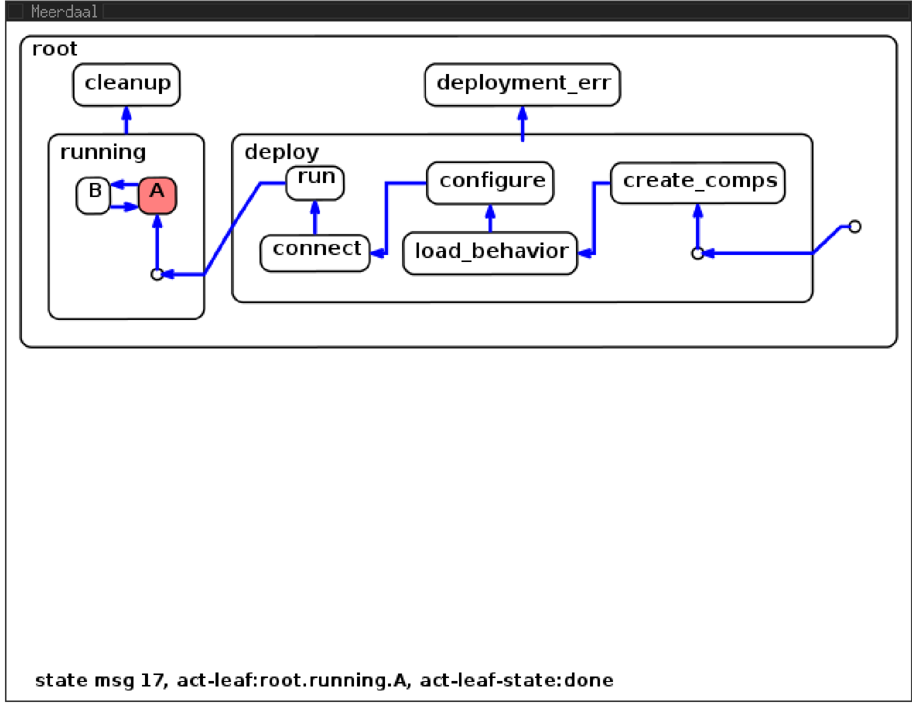

Figure A.1: rFSM online visualization tool.

A.2 rFSM commander

rFSM commander is a ncurses[1] based application that permits interacting with a running rFSM instance. Like the previous visualization tool, rfsm-commander connects to a running rFSM instance via the rFSM protocol and retrieves the current rFSM model. It then display the current state while maintaining a history of previously active states and how long each of these was active. In addition, all known events are displayed; these events can conveniently be injected into the running FSM by typing the corresponding number.

This tool has been publicly released and is available under the following link (https://github.com/kmarkus/rfsm-commander).

Both tools have been implemented using CHICKEN Scheme[2].

[1] http://www.gnu.org/software/ncurses
[2] http://www.call-cc.org/

```
mk@PMA-10-048: ~

active fqn: root.running.A
fqn state:  done
t in-state: 50.964

e_cleanup[1] e_run_OK[2] e_conf_err[3] e_connect_OK[4]
e_configure_OK[5] e_load_behavior_OK[6]
e_create_comps_OK[7] e_A[8] e_B[9]
[    10.03]   [      0.91]     root.deploy.run
[     9.12]   [      0.72]     root.deploy.connect
[     8.41]   [      0.92]     root.deploy.configure
[     7.49]   [      0.10]     root.deploy.load_behavior
[     7.39]   [      6.45]     no-fqn

raise event:
model message received.
```

Figure A.2: rFSM online interaction tool.

References

Abdellatif, T., S. Bensalem, J. Combaz, L. de Silva, and F. Ingrand (2012). Rigorous design of robot software: A formal component-based approach. *Robotics and Autonomous Systems 60*(12), 1563 – 1578.

Agerholm, S. and P. G. Larsen (1999). A lightweight approach to formal methods. In *Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods*, FM-Trends 98, London, UK, UK, pp. 168–183. Springer-Verlag.

Albu-Schäffer, A., S. Haddadin, C. Ott, A. Stemmer, T. Wimböck, and G. Hirzinger (2007). The DLR lightweight robot: design and control concepts for robots in human environments. *Industrial Robot: An International Journal 34*(5), 376–385.

Alur, R. and D. L. Dill (1994). A theory of timed automata. *Theoretical Computer Science 126*(2), 183–235.

Ando, N., T. Suehiro, and T. Kotoku (2008). A software platform for component based RT-system development: OpenRTM-Aist. In *Conf. Simulation, Modeling, and Programming of Autonomous Robots*, Venice, Italia, pp. 87–98.

Andrade, L., J. L. Fiadeiro, J. Gouveia, and G. Koutsoukos (2002). Separating computation, coordination and configuration. *Journal of software maintenance and evolution: research and practice 14*, 353–369.

Arbab, F. (1998). What do you mean, coordination? In *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, pp. 11–22.

Arbab, F. (2004, June). Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci. 14*(3), 329–366.

Arbab, F., I. Herman, and P. Spilling (1993, February). An overview of manifold and its implementation. *Concurrency: Pract. Exper. 5*(1), 23–70.

Atkinson, C. and T. Kühne (2003). Model-driven development: a metamodeling foundation. *IEEE software 20*(5), 36–41.

Autosar Consortium (2003). Autosar—AUTomotive Open System ARchitecture. http://www.automationml.org.

Baillie, J.-C. (2004). Urbi: A universal language for robotic control. *International journal of Humanoid Robotics*.

Bass, L., P. Clements, and R. Kazman (2003). *Software Architecture in Practice* (2 ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Basu, A., M. Bozga, and J. Sifakis (2006). Modeling heterogeneous real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '06, Washington, DC, USA, pp. 3–12. IEEE Computer Society.

Bauer, N. and S. Engell (2002). A comparison of sequential function charts and statecharts and an approach towards integration. In *Proceedings of the Workshop INT'02, Grenoble 2002*, pp. 58–69.

Bensalem, S., L. de Silva, F. Ingrand, and R. Yan (2011). A verifiable and correct-by-construction controller for robot functional levels. *Journal of Software Engineering in Robotics 2*(1), 1–19.

Bentley, J. (1986, August). Programming pearls: little languages. *Commun. ACM 29*(8), 711–721.

Bézivin, J. (2005). On the unification power of models. *Software and Systems Modeling 4*(2), 171–188.

Biggs, G. and B. MacDonald (2003). A survey of robot programming systems. http://citeseer.ist.psu.edu/biggs03survey.html.

Billington, D., V. Estivill-Castro, R. Hexel, and A. Rock (2010). Modelling behaviour requirements for automatic interpretation, simulation and deployment. In N. Ando, S. Balakirsky, T. Hemker, M. Reggiani, and O. von Stryk (Eds.), *Simulation, Modeling, and Programming for Autonomous Robots*, Volume 6472 of *Lecture Notes in Computer Science*, pp. 204–216. Springer Berlin / Heidelberg.

Bjarnason, E., G. Hedin, and K. Nilsson (1999, March). Interactive language development for embedded systems. *Nordic Journal of Computing 6*(1), 36–54.

Bohren, J. and S. Cousins (2010, dec.). The smach high-level executive [ros news]. *Robotics Automation Magazine, IEEE 17*(4), 18 –20.

Börger, E. and R. F. Stärk (2003). *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer.

Borland, S. (2003). Transforming statechart models to DEVS. Master's thesis, Montreal, Canada.

Botelho, S. and R. Alami (1999). M+: a scheme for multi-robot cooperation through negotiated task allocation and achievement. In *Proceedings. 1999 IEEE International Conference on Robotics and Automation, 1999.*, Volume 2, pp. 1234 –1239 vol.2.

Breen, M. (2004). Statecharts: Some critical observations.

Bruyninckx, H. and J. De Schutter (1996). Specification of force-controlled actions in the "Task Frame Formalism": A survey. *IEEE Transactions on Robotics and Automation 12*(5), 581–589.

Bruyninckx, H., N. Hochgeschwender, L. Gherardi, M. Klotzbuecher, G. Kraetzschmar, D. Brugali, A. Shakhimardanov, J. Paulus, M. Reckhaus, H. Garcia, D. Faconti, and P. Soetens. The Brics Component Model: A model-based development paradigm for complex robotics software systems.

Burmester, S., H. Giese, and M. Tichy (2004). Model-driven development of reconfigurable mechatronic systems with mechatronic uml. In U. Aßmann, M. Aksit, and A. Rensink (Eds.), *MDAFA*, Volume 3599 of *Lecture Notes in Computer Science*, pp. 47–61. Springer.

Chanel, C. P. C., F. Teichteil-Königsbuch, and C. Lesire (2012). Pomdp-based online target detection and recognition for autonomous uavs. In *ECAI 2012 - 20th European Conference on Artificial Intelligence.*, Volume 242 of *Frontiers in Artificial Intelligence and Applications*, pp. 955–960.

Conway, M. E. (1963, July). Design of a separable transition-diagram compiler. *Commun. ACM 6*, 396–408.

Coste-Maniere, E. and N. Turro (1997). The MAESTRO language and its environment: specification, validation and control of robotic missions. See IROS97 (1997), pp. 836–841.

Crockford, D. (2006). The application/json Media Type for JavaScript Object Notation (JSON). `http://tools.ietf.org/html/rfc4627`.

David, R. and H. Alla (1992). *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

De Laet, T., S. Bellens, and H. Bruyninckx (2012). Semantics underlying geometric relations between rigid bodies in robotics. `https://retf.info/rrfcs/0005`. Last visited September 2012.

De Laet, T., S. Bellens, H. Bruyninckx, and J. De Schutter (2012). Geometric relations between rigid bodies: from semantics to software. *IEEE Robotics and Automation Magazine*.

De Laet, T., S. Bellens, R. Smits, E. Aertbeliën, H. Bruyninckx, and J. De Schutter (2012). Geometric relations between rigid bodies: Semantics for standardization. *IEEE Robotics and Automation Magazine*.

De Schutter, J., T. De Laet, J. Rutgeerts, W. Decré, R. Smits, E. Aertbeliën, K. Claes, and H. Bruyninckx (2007). Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty. *The International Journal of Robotics Research 26*(5), 433–455.

De Schutter, J. and H. Van Brussel (1988). Compliant robot motion I. A formalism for specifying compliant motion tasks. *The International Journal of Robotics Research 7*(4), 3–17.

Delanote, D., S. Van Baelen, W. Joosen, and Y. Berbers (2008, april). Using AADL to model a protocol stack. In *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*, pp. 277 –281.

DFKI (2011). Rock: the robot construction kit. http://www.rock-robotics.org. Last visited November 2012.

Dijkstra, E. W. (1982). On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pp. 60–66. Springer-Verlag.

Dragert, C., J. Kienzle, and C. Verbrugge (2011). Toward high-level reuse of statechart-based ai in computer games. In *Proceedings of the 1st International Workshop on Games and Software Engineering*, GAS '11, New York, NY, USA, pp. 25–28. ACM.

Dromey, R. (2003, sept.). From requirements to design: formalizing the key steps. In *First International Conference on Software Engineering and Formal Methods.*, pp. 2 –11.

Eclipse Foundation. The Eclipse Integrated Development Environment. http://www.eclipse.org.

Eclipse Foundation. Eclipse Modelling Framework Project. http://www.eclipse.org/modeling/emf/.

Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong (2003). Taming heterogeneity—The Ptolemy approach. *Proceedings of the IEEE 91*(1), 127–144.

Elmqvist, H., F. Gaucher, S. Mattsson, and F. Dupont (2012). State Machines in Modelica. In M. Otter and D. Zimmer (Eds.), *Proceedings of 9th International Modelica Conference*, Munich, Germany.

Elrad, T., O. Aldawud, and A. Bader (2002). Aspect-oriented modeling: Bridging the gap between implementation and design. In D. Batory, C. Consel, and W. Taha (Eds.), *Generative Programming and Component*

Engineering, Volume 2487 of *Lecture Notes in Computer Science*, pp. 189–201. Springer Berlin / Heidelberg.

Eshuis, R. (2009). Reconciling statechart semantics. *Sci. Comput. Program. 74*(3), 65–99.

Finkemeyer, B., T. Kröger, and F. M. Wahl (2005). Executing assembly tasks specified by manipulation primitive nets. *Advanced Robotics 19*(5), 591–611.

Flatscher, R. G. (2002). Metamodeling in EIA/CDIF–Meta-Metamodel and Metamodels. *ACM Trans. on Modeling and Computer Simulation 12*(4), 322–342.

Fleury, S., M. Herrb, and R. Chatila (1997). GenoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture. See IROS97 (1997), pp. 842–848.

Fowler, M. (2005, Juni). Language workbenches: The killer-app for domain specific languages?

Gamha, Y., N. Bennacer, L. Ben Romdhane, G. Vidal-Naquet, and B. Ayeb (2007, july). A statechart-based model for the semantic composition of web services. In *2007 IEEE Congress on Services*, pp. 49–56.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design patterns: elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.

Gancet, J., G. Hattenberger, R. Alami, and S. Lacroix (2005, aug.). Task planning and control for a multi-uav system: architecture and algorithms. In *IROS2005*, pp. 1017–1022.

Gansner, E. R. and S. C. North (2000). An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE 30*(11), 1203–1233.

Gat, E. (1991, apr). Alfa: a language for programming reactive robotic control systems. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, Volume 2, Sacramento, CA, pp. 1116–1121. ICRA91.

Gateau, T., C. Lesire, and M. Barbier (2012). Robust strategies for multirobot team collaboration under uncertain communications. In *International Symposium on Distributed Autonomous Robotic Systems (DARS), Poster session, Baltimore, MD, USA.*

Gelernter, D. (1985, January). Generative communication in linda. *ACM Trans. Program. Lang. Syst. 7*(1), 80–112.

Gelernter, D. and N. Carriero (1992). Coordination languages and their significance. *Communications of the ACM 35*(2), 97–107.

Gerum, P. (2004). Xenomai – implementing a rtos emulation framework on gnu/linux.

Ghosal, A. (2008, Jan). *A Hierarchical Coordination Language for Reliable Real-Time Tasks*. Ph. D. thesis, EECS Department, University of California, Berkeley.

Gleixner, T. (2011). cyclictest. https://rt.wiki.kernel.org/index.php/Cyclictest. Last visited January 2013.

Harel, D. (1987). State charts: A visual formalism for complex systems. *Science of Computer Programming 8*, 231–274.

Harel, D., H. Lanchover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot (1990). STATEMATE: A working environment for the development of complex reactive system. *IEEE Transactions on Software Engineering 16*, 403–414.

Harel, D. and A. Naamad (1996). The STATEMATE semantics of statecharts. *ACM Trans. on Software Engineering Methodolody 5*(4), 293–333.

Henzinger, T. (1996, jul). The theory of hybrid automata. In *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*, pp. 278–292.

Hewitt, C., P. Bishop, and R. Steiger (1973). A universal modular ACTOR formalism for artificial intelligence. In *IJCAI1973*, pp. 235–245.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM 21*(8), 666–677.

Hoare, C. A. R. (1985). *Communicating sequential processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Hrúz, B. and M. Zhou (2007). *Modeling and Control of Discrete-Event Dynamical Systems: With Petri Nets and Other Tools*. Advanced Textbooks in Control and Signal Processing. Springer.

Hudak, P., C. Antony, H. Nilsson, and J. Peterson (2003). Arrows, robots, and functional reactive programming. In *Lecture Notes in Computer Science*, Volume 2352. Springer-Verlag.

Ierusalimschy, R., W. Celes, and L. H. de Figueiredo (2012). Lua Programming Language. http://www.lua.org. Last visited 2012.

Ierusalimschy, R., L. H. de Figueiredo, and W. Celes (2007). The evolution of lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, New York, NY, USA, pp. 2–1–2–26. ACM.

Ierusalimschy, R., L. H. de Figueiredo, and W. Celes (2012). Lua 5.1 Reference Manual. http://www.lua.org/manual/5.1/manual.html. Last visited 2012.

Ierusalimschy, R., L. H. de Figueiredo, and W. C. Filho (1996). Lua—an extensible extension language. *Softw. Pract. Exper. 26*(6), 635–652.

International Electrotechnical Commission (2010). *IEC 61508 Ed. 2: Functional Safety*. IEC.

International Electrotechnical Commission, T. C. . (2003). *IEC 61131-3 Ed. 2: Programmable controllers — Part 3: Programming Languages*. IEC.

IROS97 (1997). *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Grenoble, France. IROS97.

ISO/IEC/IEEE (2010, dec). Systems and software engineering – vocabulary. Technical report.

Jackson, D. (2002, April). Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol. 11*(2), 256–290.

Jackson, D. and J. Wing (1996, April). Lightweight formal methods. *IEEE Computer*, 21–22.

Jensen, K. (1987). Coloured petri nets. In W. Brauer, W. Reisig, and G. Rozenberg (Eds.), *Petri Nets: Central Models and Their Properties*, Volume 254 of *Lecture Notes in Computer Science*, pp. 248–299. Springer Berlin / Heidelberg. 10.1007/BFb0046842.

Joyeux, S., R. Philippsen, R. Alami, and S. Lacroix (2009). A plan manager for multi-robot systems. *The International Journal of Robotics Research*.

Kaelbling, L. P. (1987). Rex: A symbolic language for the design and parallel implementation of embedded systems. In *Proceedings of the AIAA Conference on Computers in Aerospace*, Wakefield, Massachusetts.

Kaelbling, L. P. (1988). Goals as parallel program specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Minneapolis-St. Paul, Minnesota.

Kalibera, T., F. Pizlo, A. L. Hosking, and J. Vitek (2011). Scheduling real-time garbage collection on uniprocessors. *ACM Trans. Comput. Syst. 29*(3), 8.

Kiczales, G. (1996). Aspect-oriented programming. *ACM Computing Surveys*, 154.

Klotzbuecher, M. (2011). rFSM statecharts. http://www.orocos.org/rFSM. Last visited January 2012.

Klotzbuecher, M. (2012). uMF micro-modelling framework. https://github.com/kmarkus/umf. Last visited September 2012.

Klotzbuecher, M., G. Biggs, and H. Bruyninckx (2012, November). Pure coordination using the coordinator–configurator pattern. In *Proceedings of the 3rd International Workshop on Domain-Specific Languages and models for ROBotic systems*.

Klotzbuecher, M. and H. Bruyninckx (2011, October). Hard real-time control and coordination of robot tasks using Lua. In *Proceedings of the Thirteenth Real-Time Linux Workshop*.

Klotzbuecher, M. and H. Bruyninckx (2012, September). A lightweight, composable metamodelling language for specification and validation of internal domain specific languages. In *Proceedings of the 8th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*.

Klotzbuecher, M., R. Smits, H. Bruyninckx, and J. De Schutter (2011). Reusable hybrid force-velocity controlled motion specifications with executable domain specific languages. In *Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Francisco, California, pp. 4684–4689. IROS2011.

Klotzbuecher, M., P. Soetens, and H. Bruyninckx (2010). OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages. In *International Workshop on Dynamic languages for RObotic and Sensors*, pp. 284–289.

Kopetz, H. (1993). Should responsive systems be event-triggered or time-triggered ? *Institute of Electronics, Information, and Communications Engineers Transactions on Information and Systems E76-D*(11), 1325–1332.

Kopetz, H. and G. Bauer (2003, jan). The time-triggered architecture. *Proceedings of the IEEE 91*(1), 112–126.

Korean Institute for Advanced Intelligent Systems. OPRoS. http://opros.or.kr/. Last visited November 2010.

Kortenkamp, D. and R. G. Simmons (2008). Robotic systems architectures and programming. In B. Siciliano and O. Khatib (Eds.), *Springer Handbook of Robotics*, pp. 187–206. Springer.

Lesser, V., K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. N. Prasad, A. Raja, R. Vincent, P. Xuan, and X. Q. Zhang (2004, July). Evolution of the gpgp/taems domain-independent coordination framework. *Autonomous Agents and Multi-Agent Systems 9*(1-2), 87–143.

Mallet, F. and R. Simone (2009). MARTE vs. AADL for Discrete-Event and Discrete-Time Domains. In M. Radetzki (Ed.), *Languages for Embedded Systems and their Applications*, Volume 36 of *Lecture Notes in Electrical Engineering*, pp. 27–41. Springer Netherlands.

Marty, J. C., A. E. K. Sahraoui, and M. Sartor (1998, November). Statecharts to specify the control of automated manufacturing systems. *International Journal of Production Research 36*(11), 3183–3215.

Masmano, M., I. Ripoll, P. Balbastre, and A. Crespo (2008). A constant-time dynamic storage allocator for real-time systems. *Real-Time Syst. 40*(2), 149–179.

Mason, M. T. (1979). Compliance and force control for computer controlled manipulators. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Mason, M. T. (1981). Compliance and force control for computer controlled manipulators. *IEEE Transactions on Systems, Man, and Cybernetics SMC-11*(6), 418–432.

MathWorks. Design and simulate state charts by The Mathworks. www.mathworks.de/products/stateflow/.

McCarragher, B. J. (1994). Petri net modeling for robotic assembly and trajectory planning. *IEEE Transactions on Industrial Electronics 41*(6), 631–640.

Medeiros, A. A. D. (1998, 04). A survey of control architectures for autonomous mobile robots. *Journal of the Brazilian Computer Society 4*.

Mellor, S. J. and M. Balcer (2002). *Executable UML: A Foundation for Model-Driven Architectures*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. Foreword By-Jacoboson, Ivar.

Mens, T. and P. Van Gorp (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science 152*(0), 125 – 142.

Merkle, B. (2010). Textual modeling tools: overview and comparison of language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, New York, NY, USA, pp. 139–148. ACM.

Mernik, M., J. Heering, and A. M. Sloane (2005, December). When and how to develop domain-specific languages. *ACM Comput. Surv. 37*(4), 316–344.

Merz, T., P. Rudol, and M. Wzorek (2006, july). Control system framework for autonomous robots based on extended state machines. In *International Conference on Autonomic and Autonomous Systems, 2006. ICAS '06. 2006*, pp. 14.

Mikk, E., Y. Lakhnech, M. Siegel, and G. J. Holzmann (1998). Implementing statecharts in promela/spin. In *Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, WIFT '98, Washington, DC, USA, pp. 90–. IEEE Computer Society.

Miller, J. and J. Mukerji (2003). MDA Guide version 1.0.1. Technical report, Object Management Group (OMG).

Modelica Association. Modelica: Language design for multi-domain modeling. http://www.modelica.org/.

Object Management Group. Object Constraint Language. http://www.omg.org/spec/OCL/.

Object Management Group. OMG. http://www.omg.org.

Object Management Group. Query View Transformation. http://www.omg.org/spec/QVT/.

Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), version 1.0. http://www.omg.org/spec/FUML/1.0/.

Object Management Group (2006). Meta Object Facility (MOF) core specification. http://www.omg.org/technology/documents/formal/data_distribution.htm.

Object Management Group (2009). UML specification. http://www.omg.org/spec/UML.

Object Management Group (2011). Unified Modeling Language (UML) superstructure specification, version 2.4.1. http://www.uml.org/.

Object Management Group (2012). Systems Modelling Language (SysML), version 1.3. http://www.sysml.org/.

Ogasawara, T. (1995). An algorithm with constant execution time for dynamic storage allocation. In *RTCSA '95: Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, Washington, DC, USA, pp. 21. IEEE Computer Society.

Olofson, D. (2005). The Extensible Embeddable Language. http://eel.olofson.net/. Last visited January 2013.

Otter, M., M. Malmheden, H. Elmqvist, S. E. Mattson, and C. Johnsson (2009). A new formalism for modeling of reactive and hybrid systems. In *Modelica Conference*, pp. 364–377.

Pall, M. (2011). Luajit: The lua just-in-timer compiler. http://luajit.org. Last visited January 2013.

Papadopoulos, G. A. and F. Arbab (1998). Coordination models and languages. Technical report, Amsterdam, The Netherlands, The Netherlands.

Perlis, A. J. (1982, September). Special feature: Epigrams on programming. *SIGPLAN Not. 17*(9), 7–13.

Petersson, J., P. Hudak, and C. Elliott (1999). Lambda in motion: Controlling robots with haskell. In *Lecture Notes in Computer Science*, Volume 1551, pp. 91–105. Springer-Verlag.

Petri, C. A. (1962). *Kommunikation mit Automaten.* Ph. D. thesis, Institut für instrumentelle Mathematik, Bonn.

Pizlo, F. and J. Vitek (2008). Memory management for real-time java: State of the art. *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC) 0*, 248–254.

Pohlmann, U., S. Dziwok, J. Suck, B. Wolf, C. C. Loh, and M. Tichy (2012). A modelica library for real-time coordination modeling. In M. Otter and D. Zimmer (Eds.), *Proceedings of 9th International Modelica Conference*, Munich, Germany.

Prassler, E., H. Bruyninckx, K. Nilsson, and A. Shakhimardanov (2009). The use of reuse for designing and manufacturing robots. Technical report, Robot Standards project. http://www.robot-standards.eu//Documents_RoSta_wiki/whitepaper_reuse.pdf.

Quigley, M., K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng (2009). ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software.*

Radestock, M. and S. Eisenbach (1996). Coordination in evolving systems. In *Trends in Distributed Systems. CORBA and Beyond*, pp. 162–176. Springer-Verlag.

Ramadge, P. J. and W. M. Wonham (1987, January). Supervisory control of a class of discrete event processes. *SIAM J. Control Optim. 25*(1), 206–230.

Real-Time for Java Expert Group (RTJEG) (2005, June). Real-time specification for java (rtsj). http://www.rtsj.org/specjavadoc/book_index.html.

Renggli, L., S. Ducasse, T. Gîrba, and O. Nierstrasz (2010). Domain-specific program checking. In J. Vitek (Ed.), *Objects, Models, Components, Patterns*, Volume 6141 of *Lecture Notes in Computer Science*, pp. 213–232. Springer Berlin / Heidelberg.

Renggli, L., S. Ducasse, and A. Kuhn (2007). Magritte - a meta-driven approach to empower developers and end users. In G. Engels, B. Opdyke, D. Schmidt, and F. Weil (Eds.), *Model Driven Engineering Languages and Systems*, Volume 4735 of *Lecture Notes in Computer Science*, pp. 106–120. Springer Berlin / Heidelberg.

RoboHow. The robohow project. http://robohow.eu/.

Rosell, J. (2004). Assembly and task planning using Petri nets: A survey. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture 218*, 987–994.

ROSETTA. Robot control for skilled execution of tasks in natural interaction with humans; based on autonomy, cumulative knowledge and learning. http://www.fp7rosetta.eu/.

SAE International. AADL: The SAE Architecture Analysis and Design Language. http://www.aadl.info.

Schreiber, G., A. Stemmer, and R. Bischoff (2010, May). The Fast Research Interface for the KUKA Lightweight Robot. In IEEE Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications – How to Modify and Enhance Commercial Controllers (ICRA 2010).

Shakhimardanov, A., N. Hochgeschwender, M. Reckhaus, and G. K. Kraetzschmar (2011). Analysis of software connectors in robotics. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Shangai, China, pp. 1030–1035.

Sheridan, T. B. (1992). *Telerobotics, Automation, and Human Supervisory Control*. Cambridge, MA, USA: MIT Press.

Shlaer, S. and S. J. Mellor (1988). *Object-oriented systems analysis: modeling the world in data*. Upper Saddle River, NJ, USA: Yourdon Press.

Simmons, R. and D. Apfelbaum (1998, oct). A task description language for robot control. In *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, Volume 3, pp. 1931 –1937 vol.3.

Simons, A. J. H. (2000). On the compositional properties of UML statechart diagrams. In *Rigorous Object-Oriented Methods*, Workshops in Computing. BCS.

Smits, R. (2010, May). *Robot skills: design of a constraint-based methodology and software support*. Ph. D. thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium.

Smits, R., H. Bruyninckx, and E. Aertbeliën (2001). KDL: Kinematics and Dynamics Library. http://www.orocos.org/kdl. Last visited August 2012.

Smits, R., T. De Laet, K. Claes, H. Bruyninckx, and J. De Schutter (2008). iTaSC: a tool for multi-sensor integration in robot manipulation. In *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems*, Seoul, South-Korea, pp. 426–433. MFI2008.

Soetens, P. (2006, May). *A Software Framework for Real-Time and Distributed Robot and Machine Control*. Ph. D. thesis, Department of Mechanical

Engineering, Katholieke Universiteit Leuven, Belgium. http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf.

Soetens, P. and H. Bruyninckx (2005). Realtime hybrid task-based control for robots and machine tools. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, Barcelona, Spain, pp. 260–265. ICRA2005.

Spinellis, D. (2001, feb). Notable design patterns for domain specific languages. *Journal of Systems and Software 56*(1), 91–99.

Sullivan, J. (2005, june). Impediments to and incentives for automation in the air force. In *Proceedings. 2005 International Symposium on Technology and Society, 2005. Weapons and Wires: Prevention and Safety in a Time of Fear. ISTAS 2005.*, pp. 102–110.

Sun Microsystems (2006). Memory management in the java hotspottm virtual machine. http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf.

Tarr, P., H. Ossher, W. Harrison, and S. M. Sutton, Jr. (1999). N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, New York, NY, USA, pp. 107–119. ACM.

Van de Poel, P., W. Witvrouw, H. Bruyninckx, and J. De Schutter (1993). An environment for developing and optimizing compliant robot motion tasks. In *Proceedings of the 1993 International Conference on Advanced Robotics*, Tokyo, Japan, pp. 713–718.

van Deursen, A., P. Klint, and J. Visser (2000, June). Domain-specific languages: an annotated bibliography. *SIGPLAN Not. 35*(6), 26–36.

Vanthienen, D., T. De Laet, R. Smits, and H. Bruyninckx (2011). itasc software. http://www.orocos.org/itasc. Last visited March 2013.

von der Beeck, M. (1994). A comparison of statecharts variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil (Eds.), *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Volume 863 of *Lecture Notes in Computer Science*, pp. 128–148. Springer Berlin / Heidelberg.

W3C (2001). XSD: XML Schema Definition. http://www.w3.org/XML/Schema.

W3C (2010). State Chart XML (SCXML): State machine notation for control abstraction. W3C Working Draft. http://www.w3.org/TR/scxml/.

Wainer, G. A. and P. Mosterman (2011). *Discrete-Event Modeling and Simulation: Theory and Applications*. Taylor and Francis.

Willow Garage (2008). Robot Operating System (ROS). http://www.ros.org. Last visited 2012.

Wittocx, J., M. Mariën, and M. Denecker (2008, Nov). The IDP system: A model expansion system for an extension of classical logic. In M. Denecker (Ed.), *Proceedings of the 2nd Workshop on Logic and Search*, pp. 153–165. ACCO.

Witvrouw, W., P. Van de Poel, and J. De Schutter (1995). Comrade: Compliant motion research and development environment. In *3rd IFAC/IFIP workshop on Algorithms and Architectures for Real-Time Control*, Ostend, Belgium, pp. 81–87.

Zeigler, B. P. (1976). *Theory of Modeling and Simulation*. John Wiley.

Zhao, Q. and B. Krogh (2006, sept.). Formal verification of statecharts using finite-state model checkers. *Control Systems Technology, IEEE Transactions on 14*(5), 943 –950.

Curriculum

Personal data

Markus Klotzbücher
born September 7, 1978 in Stanford, USA.
markus.klotzbuecher@mech.kuleuven.be or
mk@marumbi.de

Education

- **2009-2013:** PhD student at the Department of Mechanical Engineering, University of Leuven, Belgium.
  - Supervised by Prof. Herman Bruyninckx.
- **2008-2009** Predoctoral phase, attending courses and project work at University of Leuven.
- **1999-2004:** Computer Engineering at the University of Applied Sciences Konstanz, Germany.
  - Semester abroad in 2002: Pontifical Catholic University of Paraná (Pontifícia Universidade Católica do Paraná), Brazil.
  - Thesis: *Development of a Linux Overlay Filesystem for Software Updates in Embedded Systems.*

Experience

- **2004-2009:** Freelance consultant
  - Software development for embedded and real-time systems in the automation and telecommunication industry.
  - Teaching industrial seminars on Embedded Linux System Design and Linux Device Driver Development (both in cooperation with DENX Software Engineering).

- **2002:** Internship at Siemens–CITS (Centro Internacional de Tecnologia de Software) in Curitiba, Brazil.
  - Participation in software development of a high end telecommunication product.

- **2000-2001:** Internship at UPAQ Ltd., Zurich, Switzerland.
  - Work in network security and system administration for one semester.

- **1998-1999:** Civilian service as paramedic at the German Red Cross.
  - Training as *Rettungssanitäter* (second highest paramedical qualification).
  - Afterwards further voluntary work and training.

Publications

Articles in Internationally Reviewed Journals

1. Klotzbücher, M., Bruyninckx, H. (2012). Coordinating Robotic Tasks and Systems with rFSM Statecharts. JOSER: Journal of Software Engineering for Robotics, 3 (1), 28-56.

Papers at international scientific conferences and symposia, published in full in proceedings

1. Bruyninckx, H., Hochgeschwender, N., Gherardi, L., Klotzbücher, M., Kraetzschmar, G., Brugali, D., Shakhimardanov, A., Paulus, J., Reckhaus, M., Garcia, H., Faconti, D. and Soetens, P. (2013). The BRICS Component Model: A Model-Based Development Paradigm For Complex Robotics Software Systems.

2. Klotzbücher, M., Bruyninckx, H. (2012). A Lightweight, Composable Metamodelling Language for Specification and Validation of Internal Domain Specific Languages. Proceedings of the 8th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES '12), ASM, Essen, 3-7 September 2012, LNCS.

3. Brugali, D., Gherardi, L., Klotzbücher, M., Bruyninckx, H. (2012). Service Component Architectures in Robotics: The SCA-Orocos Integration. In Hähnle, R. (Ed.), Knoop, J. (Ed.), Margaria, T. (Ed.), Schreinerh, D. (Ed.), Steffen, B. (Ed.), International Workshops, SARS 2011 and MLSC 2011, Held Under the Auspices of ISoLA 2011 in Vienna, Austria, October 17-18, 2011: Vol. 336. ISoLA. Vienna, October 17-18, 2011 (pp. 46-60). Berlin Heidelberg: Springer.

4. Klotzbücher, M., Smits, R., Bruyninckx, H., De Schutter, J. (2011). Reusable Hybrid Force-Velocity controlled Motion Specifications with executable Domain Specific Languages. IEEE/RSJ International Conference on Intelligent Robots and Systems. San Francisco, USA., 25-30 September, 2011 IEEE.

5. Klotzbücher, M., Bruyninckx, H. (2011). Hard Real-Time Control and Coordination of Robot Tasks using Lua. Proceedings of the Thirteenth Real-Time Linux Workshop. Real Time Linux Workshop. Czech Technical University, Prague, 20-22 October 2011 (pp. 37-43) Open Source Automation Development Lab (OSADL) eG.

6. Klotzbücher, M., Soetens, P., Bruyninckx, H. (2010). OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages. Proceedings of SIMPAR 2010 Workshops. SIMPAR Workshops, Darmstadt, Germany, 15-16 November 2010 (pp. 284-289).

Meeting abstracts, presented at international scientific conferences and symposia, published or not published in proceedings or journals

1. Buys, K., Bellens, S., Vanthienen, N., De Laet, T., Smits, R., Klotzbücher, M., Decré, W., Bruyninckx, H., De Schutter, J. (2011). Haptic coupling with augmented feedback between the KUKA youBot and the PR2 robot arms. International Conference on Intelligent Robots and Systems. San Francisco, Califormia, 25-30 September, 2011.

2. Vanthienen, N., De Laet, T., Smits, R., Buys, K., Bellens, S., Klotzbücher, M., Bruyninckx, H., De Schutter, J. (2011). Demonstration of iTaSC as a unified framework for task specification, control, and coordination for mobile manipulation. IEEE/RSJ International Conference on Intelligent Robots and Systems. San Francisco, 25-30 September 2011.

3. Buys, K., Bellens, S., Vanthienen, N., Decré, W., Klotzbücher, M., De Laet, T., Smits, R., Bruyninckx, H., De Schutter, J. (2011). Haptic coupling with the PR2 as a demo of the OROCOS - ROS - Blender integration. IROS. San Francisco, California, 25-30 September 2011.

4. Vanthienen, N., De Laet, T., Decré, W., Smits, R., Klotzbücher, M., Buys, K., Bellens, S., Gherardi, L., Bruyninckx, H., De Schutter, J. (2011). iTaSC as a unified framework for task specification, control, and coordination, demonstrated on the PR2. IEEE/RSJ International

Conference on Intelligent Robots and Systems. San Francisco, 25-30 September 2011.

5. Soetens, P., Klotzbücher, M. (2009) Real Time Toolkit for Open Robot Control Software. Xenomai User Meeting (XUM), September 28, 2009. Eleventh Real Time Linux Workshop, TU Dresden, Germany.

FACULTY OF ENGINEERING
DEPARTMENT OF MECHANICAL ENGINEERING
PRODUCTION ENGINEERING, MACHINE DESIGN AND AUTOMATION DIVISION
Celestijnenlaan 300B box 2420
B-3001 Heverlee
info@mech.kuleuven.be
http://www.mech.kuleuven.be