Managing Software Complexity of Adaptive Systems

Arjan de Roo

Managing Software Complexity of Adaptive Systems

Arjan de Roo

Ph.D. dissertation committee

Chairman and secretary:
Prof. dr. ir. A.J. Mouthaan, University of Twente, The Netherlands

Promotor:
Prof. dr. ir. M. Akşit, University of Twente, The Netherlands

Assistant-promotors:
Dr. ir. L.M.J. Bergmans, University of Twente, The Netherlands
Dr. H. Sözer, Özyeğin University, Turkey

Members:
Prof. dr. U. Aßmann, Technische Universität Dresden, Germany
Prof. dr. ir. T. Basten, Eindhoven University of Technology, The Netherlands
Prof. dr. I. Crnković, Mälardalen University, Sweden
Prof. dr. ing. P.J.M. Havinga, University of Twente, The Netherlands
Prof. dr. ir. G.J.M. Smit, University of Twente, The Netherlands
Prof. dr.-ing. M. Südholt, École des Mines de Nantes, France

CTIT Ph.D. thesis series no. 12-217
Center for Telematics and Information Technology
P.O. Box 217 - 7500 AE Enschede
The Netherlands

This work has been carried out as part of the OCTOPUS project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute program.

ISBN 978-90-365-3319-5
ISSN 1381-3617 (CTIT Ph.D. thesis series no. 12-217)
DOI: 10.3990/1.9789036533195

Cover design: *Seoul by Night* by Arjan de Roo and Kardelen Hatun
Printed by Ipskamp Drukkers B.V., Enschede, The Netherlands
Copyright © 2012, Arjan de Roo, Enschede, The Netherlands
All rights reserved

MANAGING SOFTWARE COMPLEXITY OF ADAPTIVE SYSTEMS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 2 februari 2012 om 14.45 uur

door

Auke Jan de Roo

geboren op 14 september 1982
te Stadskanaal

Dit proefschrift is goedgekeurd door:

Prof. dr. ir. Mehmet Akşit (promotor)
Dr. ir. L.M.J. Bergmans (assistent-promotor)
Dr. H. Sözer (assistent-promotor)

"Man is not born to solve the problem of the universe,
but to find out what he has to do; and to restrain himself
within the limits of his comprehension."

Johann Wolfgang von Goethe

Acknowledgments

Life offers many challenges of varying difficulty and complexity. One can decide to ignore or avoid these challenges, and live life the easy way. One can also accept these challenges with an eagerness to learn and grow, and experience life to the fullest.

In late 2007 I accepted the challenge to become a PhD student. Finishing a PhD is something one cannot achieve alone. During the past four years there were many people that believed in me and without their help and support I would not have been able to write this thesis. I am very grateful that these people are part of my life and would like to thank them for their share in my PhD.

First of all, I would like to thank my promotor Mehmet Akşit. You offered me the opportunity to work in the Software Engineering group as a PhD student on the Octopus project. I deeply respect you, and during our inspiring meetings you often amazed me with your wealth of knowledge on widely varying subjects, and your ability to quickly grasp important concepts and always think multiple steps ahead. The discussions we had during these meetings provided the basis for the work in this thesis.

I would like to thank my daily supervisors, Lodewijk Bergmans and Hasan Sözer. Both were of great support to me during my PhD. Lodewijk, during my Master's project you encouraged me to continue afterwards with a PhD, something that I hadn't given much thought before, and something that I never regretted afterwards. During the first year of my PhD, you introduced me into the world of research and inspired me with your endless enthusiasm for the subject. Hasan, you took over my daily supervision in the beginning of my second year. You just finished a PhD yourself, so this must have been an exciting new challenge for you. For the next two years we closely cooperated and your structured and focused way of working were of great guidance to me. It is no coincidence that during this period most of the results of my thesis materialized. Furthermore, you broadened my perspective and my research to the subjects of software architectures and software reliability. I always enjoyed our meetings and I am very thankful that we could continue our cooperation after you accepted a position at Özyeğin University, early 2011. We definitely keep in contact, and I will visit you somewhere in Spring in İstanbul! Lodewijk, during the final year of my PhD your critical, unbiased view enabled me to improve my work and your guidance and encouragements inspired me to push myself further towards my limits. In the end, this resulted in a significantly better thesis. As both our interests include entrepreneurship, I hope we can continue our cooperation after finishing my PhD!

I also would like to thank the members of my PhD committee: prof. dr. Uwe

Aßmann, prof. dr. ir. Twan Basten, prof. dr. Ivica Crnković, prof. dr. ing. Paul Havinga, prof. dr. ir. Gerard Smit and prof. dr.-ing. Mario Südholt. I am very honored that you accepted our invitation to be part of my committee and thank you for reading my thesis and providing valuable feedback to improve it.

I enjoyed the past four years working in the Software Engineering group, and therefore I would like to thank all members of the group. I want to thank Jeanette Rebel-de Boer for handling many of the administrative procedures around my defense. I would like to explicitly mention my office mates, with whom I had fun over the years: Michiel Hendriks, Wilke Havinga, Tom Staijen, Kardelen Hatun, Arda Goknil and Haihan Yin. I especially want to thank Kardelen Hatun. I really enjoyed the many discussions we had in the car, driving to Venlo every Tuesday. We shared frustrations, gave each other advice or just spoke about varying subjects, ranging from gardening to Turkish politics. You are a great friend to me!

I would like to thank all members involved in the Octopus project. First of all, the people at Océ, who were always open for new solutions and provided constructive feedback. You offered me the opportunity to perform my research in an industrial context, enabling results that are not merely academic exercises. In particular, I want to thank René Waarsing, who always provided useful feedback during meetings and who is really good in understanding academic solutions and seeing their value for Océ, and Ronald Fabel, who provided coaching to me during the first couple of years in the project and who introduced me to real software architecting in industry. Second, I thank all the people at ESI, in particular Jacques Verriet, who reviewed many of my papers and provided useful feedback to them. And finally, I thank all the colleague-researchers I had the opportunity to work with during the project. In particular, I want to mention Arjen Hommersom and Magda Chmarra; we three faced the challenge to get the project started, but also had a lot of fun along the way. Arjen, thanks for your efforts every week in the first year of the project to pick me and Magda up from the station in the morning and to drop us of in the afternoon again, a considerable detour for you; this made the burden of travelling by public transport to Venlo every week a little bit more bearable. Magda, thanks for helping me order stuff from a Polish website, shipping it to your parent's address and bringing it with you to the Netherlands (and Sander, thanks for dropping this stuff of for me in Nijmegen)!

During my PhD I shared many nice moments with my friends and made a lot of new friends, some of who became very close and special to me. I want to thank you for making my PhD life more enjoyable! I especially made many friends being a member of DBV DIOK. We had lot of fun together playing badminton and going to tournaments and parties. Unfortunately, with the end of my PhD my days as a member are also ending. But I hope to continue seeing you, either on a badminton court or outside. In particular, I want to thank my friends Afke Stellingwerff and Christian Beltman, for being my paranymphs and supporting me during my defense.

Last, but not least, I want to thank my dear family: my parents Siegert and Aafke, and my sister Tessa. Your unconditional love and believe in me are of great support to me, to face all the challenges I accept in live. I love you!

Arjan de Roo
Enschede, January 2012

Abstract

To survive under competitive pressure, embedded system companies build systems that can deal with changing customer needs and operating conditions, and deterioration of the hardware over the lifetime of the embedded system. Engineers face the challenge to design such adaptive systems, while keeping hardware costs low. To accomplish this, increasingly sophisticated control strategies are being designed and implemented in embedded systems. An example of such a sophisticated control strategy is runtime optimization of multiple system qualities, such as power consumption and productivity, and dynamically making trade-offs between such qualities (or objectives) based on (varying) user needs.

Large part of the control logic of embedded systems is implemented in software. The implementation of sophisticated control strategies introduces additional complexity in embedded control software. Part of this complexity is inevitable: it is a result of essential complexity in the selected control strategy. However, the lack of structured methods to design and incorporate sophisticated control strategies in software and the lack of proper abstraction mechanisms in programming languages to express these strategies introduce accidental complexity in the software. Accidental complexity reduces software quality with respect to several quality criteria such as comprehensibility, reliability, maintainability and reusability. The subject of this thesis is how to manage the complexity introduced by sophisticated control strategies, and how to reduce the impact of this complexity on software quality.

This thesis provides three main contributions. First, this thesis proposes a novel technique to compose domain-specific models of physical characteristics (physical models) with control software modules written in a general-purpose programming language. As such, it combines the benefits of domain-specific abstractions in a domain-specific modeling language with the freedom of a general-purpose programming language. Second, this thesis provides a method for runtime verification of models of physical characteristics that are utilized in embedded control software, as such models may be wrong or inaccurate. Third, this thesis presents a structured method to include multi-objective optimization solutions in the architecture of embedded control software. This method prevents tailored solutions and tight integration of optimization algorithms with control software modules.

One method to make an embedded system more adaptive is to allow the values of certain controlled physical variables (e.g., printing speed and certain temperatures needed for correct printing in digital document printing systems) in the system to vary (between certain ranges). Instead of controlling these physical variables to fixed setpoints, varying their values enables the system to operate more effectively under

changing circumstances. However, correct behavior of an embedded system usually depends on a set of mathematical relations between the physical variables that need to be satisfied. If the values of certain physical variables are allowed to vary, it becomes the responsibility of the control software to ensure the satisfaction of these mathematical relations. As the mathematical relations to be satisfied are based on physical characteristics of the system, models of these physical characteristics (i.e., physical models) become part of control software. In current practices, usually general-purpose languages (GPLs), such as C and C++, are used for the development of embedded control software. Although a GPL is suitable for expressing general-purpose computation, it is less effective for expressing physical models: Domain-specific abstractions are translated to general-purpose implementation abstractions. This reduces the comprehensibility of the implementation (especially for domain experts) and compromises evolvability and reusability of the software. Moreover, domain-specific static and dynamic checks may not be applied effectively. However, there exist domain-specific modeling languages (DSMLs) and tools to specify physical models. Examples of these are 20-Sim and Matlab Simulink. Although they are commonly used for simulation and documentation of physical systems, they are not often used to implement embedded control software. This is due to the fact that these DSMLs are not suitable to express general-purpose computation and the resulting models cannot be easily composed with other software modules that are implemented in GPLs. This thesis proposes a novel technique to apply 20-Sim models, containing physical characteristics, in embedded control software and compose these models with software modules written in a GPL, using the Composition Filters model. This technique combines the benefits of using a DSML to model physical characteristics (e.g., domain-specific abstractions, reuse of models between engineering phases) with the freedom of a GPL to implement general-purpose computation. The application of the Composition Filters model provides aspect-oriented composition of physical models with software modules, abstracting and reducing the dependencies between them. The Composition Filters model also offers a declarative composition mechanism, which enhances the possibilities for static analysis. This thesis proposes a method to analyze the composition filters that compose the 20-Sim models with GPL software modules, which is for example useful for design time verification and code generation.

The implemented physical models may not always accurately reflect physical reality, e.g., because the physical system has evolved, the physical system is used in different circumstances than it was tested for, or the physical system has changed because of wear and tear. As inaccuracies in physical models may lead to incorrect behavior of the system, the accuracy of physical models needs to be verified. This is the second problem that is addressed in this thesis. One cannot test or statically verify the system for all possible operating conditions, thus runtime verification is necessary. However, traditional runtime verification techniques cannot be applied, as their aim is to verify the conformance of a software system to a model of the software system. The aim of our approach is to verify the conformance of a model of physical characteristics used in software with physical reality. This thesis proposes a novel approach for runtime verification of physical models in embedded control software, exploiting redundancy in these models (e.g., because of redundant or overlapping sensor information).

Embedded systems are expected to behave optimally under changing circumstances and they have to satisfy varying customer needs. Therefore, these systems have to optimize multiple objectives regarding different system qualities (e.g., maximize productivity and minimize energy consumption) at runtime and they have to make dynamic trade-offs among these objectives. There exist algorithms to perform multi-objective optimization of system qualities. However, the system-wide impact of such algorithms combined with the lack of structured methods to design such multi-objective optimization in embedded control software leads to solutions that are tailored to the specific system and tightly integrated with control software modules. This thesis presents a structured method, called MO2 method, to include multi-objective optimization solutions (*MOO solutions*) in the architecture of embedded control software. The MO2 method provides the MO2 architectural style and a toolchain. The MO2 architectural style enables the specification of control architectures that include MOO solutions. The toolchain contains tools to edit architectural models according to the MO2 style, to validate the consistency of these models concerning the implemented MOO solution, and to generate an implementation of the optimizer that performs the multi-objective optimization in the embedded control software. The MO2 method gives engineers the possibility to systematically introduce MOO solutions in embedded control software and to reuse MOO solutions and algorithms between different systems. Furthermore, the method supports documentation of MOO solutions in the software architecture.

The techniques proposed in this thesis are validated using a qualitative evaluation of the ability of the proposed techniques to manage software complexity in adaptive embedded systems. Realistic evolution scenarios are used to evaluate the maintainability and evolvability of software applying state-of-the-practice techniques and of software applying the techniques proposed in this thesis. Furthermore, an experiment has been performed to test the performance of a system that applies the MO2 method for optimization. A comparison is made with other systems that use state-of-the-practice engineering solutions for optimization.

Contents

Acknowledgments **vii**

Abstract **ix**

1 Introduction **1**
1.1 Background: Adaptive Systems, Software Quality and Complexity . . 1
1.1.1 Adaptive Systems 2
1.1.2 Software Quality 6
1.1.3 (Software) Complexity 7
1.2 Context: Adaptive Embedded Systems 9
1.2.1 Embedded Systems and the Motivation for their Adaptivity . . 9
1.2.2 Behavior of a Physical System 10
1.2.3 Adaptivity: Increasing Actual Behavior Space 12
1.3 Industrial Case Studies 13
1.3.1 Case Study I: Warm Process 13
1.3.2 Case Study II: Drum Shuttling 17
1.3.3 Software Challenge 19
1.4 Motivation & Objectives 20
1.5 Solutions and Contributions 21

2 Composing Domain-Specific Physical Models with Embedded Control Software **23**
2.1 Introduction 23
2.2 Problem Statement 25
2.2.1 Current State-of-the-practice 26
2.2.2 Development with GPLs 26
2.2.3 Development based on DSMLs 28
2.2.4 Composing DSML and GPL Artifacts 29
2.3 Approach Overview 30
2.4 Specifying and Executing Models of Physical Characteristics 31
2.4.1 Introduction to 20-Sim/SIDOPS+ 32
2.4.2 Definition: Dependency Graph 33

2.4.3 Instantiation of Physical Models 36
2.4.4 Composing Physical Models 38
2.4.5 Executing Physical Models 39
2.4.6 Definition: Derivation Graph 48
2.5 Composition using the Composition Filters Model 50
2.5.1 Background: The Composition Filters Model 50
2.5.2 Composition Overview 53
2.5.3 Base Interface 55
2.5.4 The Event Model 55
2.5.5 Resolving Inconsistencies 59
2.5.6 Default Composition Filter 64
2.6 Design 64
2.6.1 Structure of the DSML Interpreter 64
2.6.2 Connection with the Compose* Runtime 65
2.6.3 Basic Equation Solver 67
2.7 Example Control Software Designs 70
2.8 Evaluation 74
2.8.1 Implementing Physical Models with a DSML 74
2.8.2 Application of Composition Filters 75
2.9 Discussion 76
2.9.1 Applicability on Systems with Tight Timing Constraints 76
2.9.2 Separating Design Rationale from Control Logic 77
2.9.3 Control Logic in a DSML 77
2.9.4 Risks of Physical Model Decomposition 78
2.9.5 Difference between 20-Sim Simulation and Physical Model Execution 79
2.9.6 Dynamic Adaptation of Physical Models 80
2.10 Related Work 80
2.10.1 Domain-specific Models in Embedded Control Software 80
2.10.2 Interaction Based Approaches 81
2.10.3 Connection with System Modeling Approaches 82
2.10.4 Heterogeneous Composition of Computational Models 83
2.11 Conclusion and Future Work 84

3 Verification & Analysis of Physical Models in Embedded Control Software 87
3.1 Introduction 87
3.2 Problem Statement 88
3.2.1 Verifying Physical Relationships 89
3.2.2 Failures Observable in Physical System Behavior 89
3.2.3 Analyzing the Composition 90
3.3 Runtime Verification of Physical Models 91
3.3.1 Using Redundancy to Verify Correctness 91
3.3.2 Diagnosing Faults 94
3.3.3 Example Applications 97
3.4 Composition Analysis 99

3.4.1 Background: Composition Filter Reasoning 99
3.4.2 Extending Filter Reasoning to the Event Model 100
3.4.3 Applications of Filter Reasoning 105
3.5 Applying Inconsistency Monitoring for Calibration and Broken Sensor or Component Detection 107
3.5.1 Calibration 107
3.5.2 Broken Sensor or Stepper Motor Detection 109
3.5.3 Benefits of the Composition Filters Model 111
3.6 Discussion 112
3.6.1 Efficiency of the Monitoring Approach 112
3.6.2 Moment of Checking 112
3.6.3 Recovery Actions 112
3.7 Related Work 113
3.8 Conclusion 114

4 The MO2 Method for Runtime Optimization of Multiple System Qualities in Embedded Control Software 117
4.1 Introduction 117
4.2 Background: Multi-Objective Optimization 118
4.2.1 Problem Description 119
4.2.2 Optimization 122
4.3 Context & Problem 124
4.3.1 Analysis of Optimization in the Warm Process Case Study . . 125
4.3.2 Lack of Systematic Methods 127
4.3.3 Requirements for a Systematic Specification Method 128
4.4 MO2 Method Overview 128
4.5 MO2 Architectural Style 130
4.5.1 Style Description 130
4.5.2 Style Notation 131
4.5.3 Basics of Components, Ports and Connectors 132
4.5.4 Specifying the MO2 Solution 133
4.6 MO2 Toolchain 137
4.6.1 MO2 Model Editor 138
4.6.2 MO2 Model Processor / Mathematical Representation 139
4.6.3 MO2 Consistency Validator 147
4.6.4 MO2 Code Generator 153
4.6.5 MO2 Code Weaver 160
4.7 Advanced Application 161
4.7.1 Optimization over Time 161
4.7.2 Hierarchical Optimization 163
4.8 Discussion 164
4.8.1 Values Provided to Ports with the isDecisionVariable Property Set 164
4.8.2 Generating Code for Analyzable Components 165
4.8.3 Computational Performance of Multi-Objective Optimization . 165
4.9 Related Work 165

4.9.1 Architectural Styles 165
4.9.2 Theory of Multi-Objective Optimization 166
4.10 Conclusion 167

5 Experimentation & Validation **169**
5.1 Experiment: Optimization Performance 169
5.1.1 Test System Requirements & Assumptions 170
5.1.2 Experiment Setup 170
5.2 Experiment: Control Software Implementations 171
5.2.1 Implementation 1: Multi-Objective Optimization 172
5.2.2 Implementation 2: Intelligent Speed 177
5.2.3 Implementation 3: Eco Mode 179
5.3 Experiment: Results 181
5.3.1 Main Results 181
5.3.2 Productivity in Detail 182
5.4 Experiment: Evaluation & Discussion 187
5.4.1 Best Performance for the MO2 Implementation 187
5.4.2 Instable Eco Mode Implementation 187
5.4.3 Power Margins 187
5.4.4 Delay in Speed Changes 189
5.4.5 Reverse Correlation between Productivity and Energy Consumed 190
5.5 Qualitative Evaluation using Evolution Scenarios 190
5.5.1 Evolution Scenario Type 1: Change in Physical Characteristics 191
5.5.2 Evolution Scenario Type 2: Adding or Changing Runtime Verification 195
5.5.3 Evolution Scenario Type 3: Adding or Changing Multi-Objective Optimization Solutions 200
5.5.4 Summary of Development Impact 205
5.6 Conclusion 206

6 Conclusion & Future Directions **207**
6.1 Problems Addressed 207
6.1.1 Physical Models in Embedded Control Software 208
6.1.2 Designing Multi-Objective Optimization Functionality in Embedded Control Software 209
6.2 Integrated Overview of the Approaches 211
6.2.1 Physical System Design 212
6.2.2 The MO2 Method 212
6.2.3 Composition of Physical Models with Software Modules 213
6.2.4 Verification of Physical Models 213
6.3 Contributions 214
6.3.1 Managing Complexity 214
6.3.2 Analysis & Verification 218
6.4 Future Directions 219
6.4.1 Managing Complexity of other Domain-Specific Functionality . 220
6.4.2 Composition of Domain-Specific Concerns 220

6.4.3 Vertical Integration of Engineering Disciplines 221

A Terminology **223**
A.1 Dependency Graph 223
A.2 Derivation Graph 224

B Derivation of I_{gs}^{act} **225**

C xADL Schema Extension **227**

D Object Models **231**

E Generated Multi-Objective Optimization Code **233**

Bibliography **235**

Samenvatting **245**

Notes **249**

List of Figures

1.1 Combining Zadeh and Martín 4
1.2 Schematic overview of the three behavior spaces of a physical system . 10
1.3 Schematic view of the *Warm Process* 14
1.4 Software dependencies in control system 1 15
1.5 Software dependencies in control system 2 16
1.6 Schematic view of the drum and components for rotation and shuttling 17
1.7 Software dependencies in control system 1 18
1.8 Software dependencies in control system 2 19

2.1 Overview of our approach 25
2.2 Detailed overview of our approach 30
2.3 Example dependency graph 36
2.4 Instantiation of a physical model 37
2.5 Interaction points between models 39
2.6 Example dependency graph 42
2.7 Forward solving applied 43
2.8 Backward solving applied 45
2.9 Before no-update solving 46
2.10 No-update solving applied 47
2.11 Example derivation graph 49
2.12 Overview of the Composition Filters model 50
2.13 Some classes in a Pacman game 51
2.14 Composition filters applied to a physical model instance 54
2.15 Base interface of physical model instances 55
2.16 Dependency graph showing multiple ways to update certain variables . 60
2.17 Class diagram of the DSML interpreter 65
2.18 Class diagram showing the connection with Compose* 66
2.19 The abstract syntax tree of Equation 1.2 68
2.20 Operator definitions 68
2.21 Example of an equation solving problem 69
2.22 Solution after applying the basic equation solving algorithm 69
2.23 Warm Process software structure and data flow 70
2.24 Different artifacts shown in the overview figure of the approach 71

2.25 Drum Shuttling software structure and data flow 72
2.26 Different artifacts shown in the overview figure of the approach 73

3.1 Dependency graph of the `BeltTemperature` model 93
3.2 Drum Shuttling software structure and data flow 108

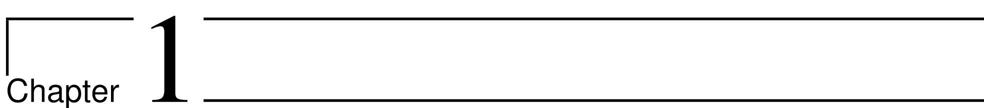
4.1 The feasible decision space 121
4.2 The objective space 121
4.3 The objective space and Pareto frontier 123
4.4 The Pareto frontier plotted in the feasible decision space 124
4.5 Warm process software architecture 126
4.6 Overview of the Method 129
4.7 MO2 model of the case study 133
4.8 SubModel component of the Warm Process MO2 model 137
4.9 Overview of the toolchain 138
4.10 Screenshot of the MO2 Model Editor 139
4.11 Example component, SIDOPS+ specification and component constraint 140
4.12 Corresponding dependency graph 140
4.13 Matching of ports to variable nodes 141
4.14 Dependency graph extended with the component's constraint 142
4.15 The component's derivation graph 143
4.16 Derivation graph for the example architecture 145
4.17 Structure of the derivation graph of a MO2 submodel 146
4.18 Dependent nodes in the example derivation graph 150
4.19 Example expansion of a variable node 157
4.20 MO2 view with optimization over time 162
4.21 MO2 view showing hierarchical application of the style 164

5.1 Simulation setup 171
5.2 MO2 Model of the Warm Process case study 173
5.3 Software structure of the Multi-Objective Optimization Control implementation 175
5.4 Software structure of the Intelligent Speed Control implementation . . 179
5.5 Software structure of the Eco Mode Control implementation 180
5.6 Average printing speed 183
5.7 Average energy consumption 183
5.8 Average power margin 184
5.9 Average deviation from print quality 184
5.10 Performance of the four implementations concerning productivity with a constant power supply 185
5.11 Performance of the four implementations concerning productivity for 20 scenarios with a fluctuating power supply 185
5.12 Speed in one given scenario with a fluctuating power supply 186
5.13 Speed in the given scenario for the Eco Mode implementation 186
5.14 Power margin during one scenario for the Intelligent Speed implementation 188

5.15 Power margin during the same scenario for the MO2 implementation with 100 W margin 189

6.1 Integration of our approach 211

D.1 Dependency/Derivation graph object model 231
D.2 MO2 object model 232



Chapter 1

Introduction

There is a constant pressure for embedded system manufacturers to build better embedded systems for lower costs. These systems need to cope with an increased variation in customer needs, operating conditions, etc. To enable this, more advanced algorithms to control the system are being applied in embedded control software. However, the implementation of these advanced algorithms increases the complexity of embedded control software. Higher complexity has a negative impact on software quality, such as comprehensibility, maintainability and evolvability. Because of these drawbacks, engineers may decide not to implement more advanced algorithms to control the system. As such, opportunities to gain a competititive advantages may be missed.

This thesis provides structured methods and techniques to better manage the complexity caused by the more advanced algorithms for adaptive control behavior. The application of these structured methods and techniques will decrease the impact of more advanced algorithms on software quality, enabling engineers to develop more adaptive systems.

This chapter is organized as follows. It starts with the background on general concepts used in this thesis, to obtain a common understanding of these concepts. This is followed by a section that introduces the context of this thesis, a research project that studies how adaptive embedded systems can be designed. The section also describes how the research in this thesis fits within this context. Next, two industrial case studies are introduced. These case studies will be used throughout this thesis to demonstrate the problems and provided solutions. Finally, the objectives for this thesis are defined and motivated.

1.1 Background: Adaptive Systems, Software Quality and Complexity

This section provides background information on some important concepts used in this thesis. The research in this thesis is performed in the context of adaptive embedded systems. Therefore, this section first provides a definition of *adaptive systems*.

The research in this thesis focuses on managing complexity of software for adaptive embedded systems. Therefore, definitions of *complexity*, *software complexity* and *software quality* are given.

1.1.1 Adaptive Systems

In the literature there are diverse definitions of *adaptive systems*. This section discusses two relevant definitions, each taking a different perspective on adaptive systems. The differences between the two definitions are discussed and a combined application of the two definitions is made, giving additional insights into adaptive systems.

Martín's Definition

In [85] Martín et al. define an adaptive system as follows:

> "*An adaptive system is a set of elements which interact with each other and has at least one process which controls the system's adaptation, that is, the correlation between structure, function or behavior and its environment, to increase its efficiency to achieve its goals.*"

In the definition of Martín et al., the adaptive system is divided into two parts, a *goal system* and an *adaptation process*. The goal system contains the elements that operate to fulfill the goals for which the system was designed. An example of this could be a printer system, including control software, which has the goal to print documents. The adaptation process contains the process or processes that adapt the elements in the goal system to let the goal system achieve its goals more effectively. An example of such a process in a printer system could be higher-level control that adapts the printer system to cope with changing environment temperatures or to cope with wear and tear.

Zadeh's Definition

In [113] Zadeh aims to formally define the notion of adaptive systems. He makes a distinction between the external manifestation of adaptive behavior and the internal mechanism by which it is achieved. In his definition of an adaptive system, the internal mechanism to achieve the adaptive behavior is ignored, as it is not relevant how the adaptive behavior is achieved.

To define an adaptive system, Zadeh introduces the following concepts:

- A system s
- A time-based function i. This function represents the input to the system over time (i.e., it maps time to a vector of input values). It can be seen as an operation scenario for the system. The input includes input from external controllers, changes in environmental conditions and external disturbances.
- $b = s(i)$ represents the behavior of system s given function i as its input.

- A set of time-based functions I. Each $i \in I$ can be the input to the system, i.e., each $i \in I$ is a possible operation scenario for the system. The set I is typically infinite.
- $B = S(I)$ represents the set of behaviors of the system s for all possible input functions in the set I. $S(I)$ is defined as follows: $S(I) = \{s(i)|i \in I\}$.
- A performance function p, which gives a performance measurement for the behavior b.
- A performance set-function P, which gives the set of performance measurements for all behaviors $B = S(I)$ for a given system s and set of input functions I. It is defined as: $P(B) = \{p(b)|b \in B\}$.
- A performance set W, containing all performance measurements that are acceptable.

Zadeh defines a system s to be adaptive for input set I under the performance set W, if the performance of the system is acceptable for I, i.e. $P(S(I)) \subseteq W$.

Note that under Zadeh's definition it is not the question whether a system is adaptive or not; every system is to a certain extent adaptive. The question is, however, for which I a system is adaptive under a given W. The larger this I, the more adaptive the system under the given W. Put into other words, if for two systems s_1 and s_2, s_2 can handle all functions i that s_1 can handle, and there is a function i' that can be handled by s_2 but not by s_1, then s_2 is more adaptive than s_1[1].

Zadeh vs Martín

The difference between Zadeh's definition and the definition of Martín et al. is that Zadeh defines an adaptive system from a perspective external to the system; how adaptive the system is, is defined as a performance measurement of the system for a given input set and the acceptability of the resulting performance value. Zadeh does not specify what the internal properties of an adaptive system are, as opposed to Martín's definition, in which an adaptive system is divided into two parts, one part that behaves to reach goals and one part that adapts the other part to make it more effective. As such, Martín's definition makes a clear distinction between systems that are adaptive (i.e., systems that have the described separation into two parts) and systems that are not adaptive (i.e., systems that don't have the described separation into two parts)

Zadeh and Martín Combined

Because Zadeh and Martín define adaptive systems from different perspectives, they complement each other. The combined application of the two definitions on an example system can therefore be insightful.

[1] Note that there only is a partial ordering between systems concerning adaptivity; within Zadeh's definition there can be two systems s_1 and s_2 and two functions i_1 and i_2, where s_1 accepts i_1 as input while s_2 does not, and s_2 accepts i_2 as input while s_1 does not. In this case it cannot be concluded that one of the systems is more adaptive than the other system, and as such there is no ordering relationship between the two systems.

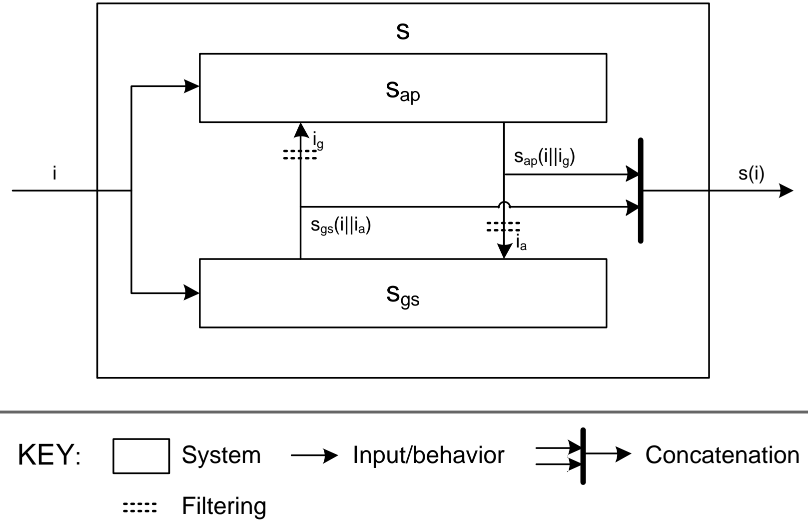


Figure 1.1: Combining Zadeh and Martín

Figure 1.1 shows a system s that is a composition of two subsystems, a goal system s_{gs} and an adaptation process s_{ap}, following Martín's definition. Time-based function i represents the input to the system s over time. As given in Zadeh's definition, i is a time-based function that produces a vector of values for each time instance. This vector of values is the input to the system at the given time instance. The dimension of this vector (i.e., the result vector of function i) is represented by k. This function i is part of the input to both subsystems s_{gs} and s_{ap}.

Input The input to the adaptation process s_{ap} consists of i and i_g. i is the input to the entire system s. i_g is the input to the (additional) monitoring interface of the adaptation process. Using this monitoring interface, the adaptation process receives information about the goal system. This information is used by the adaptation process to determine how to adapt the goal system to make it more effective. The information in i_g might originate from physical sensors in the goal system, software monitors in the software part of the goal system, etc. The dimension of the result vector of i_g is represented by n. The combined input to the adaptation process is represented by $i||i_g$[2].

The input to the goal system s_{gs} consists of i and i_a. i is the input to the entire system s. i_a is the input to the (additional) adaptation interface of the goal system s_{gs}. The dimension of the result vector of i_a is represented by m. The combined input to the goal system is the concatenation of the two input functions i and i_a. The

[2]The concatenation of two time-based functions i_1 and i_2 results in a new time-based function $i = i_1||i_2$. This function i is constructed by applying its input to both subfunctions i_1 and i_2 and concatenating the two result vectors, i.e., $i(t) = (i_1||i_2)(t) = i_1(t)||i_2(t)$. Concatenation of two vectors $v_1 = [v_1[1], v_1[2], \ldots, v_1[m]]$ and $v_2 = [v_2[1], v_2[2], \ldots, v_2[n]]$ is defined as $v_1||v_2 = [v_1[1], v_1[2], \ldots, v_1[m], v_2[1], v_2[2], \ldots, v_2[n]]$.

concatenation of i and i_a is represented by $i||i_a$.

Behavior The behavior b_{gs} of the goal system s_{gs} is a function of its input and given by $b_{gs} = s_{gs}(i||i_a)$. The behavior b_{ap} of the adaptation process s_{ap} is a function of its input and given by $b_{ap} = s_{ap}(i||i_g)$. The behavior b of the system s is a concatenation of the behavior of the adaptation process and the behavior of the goal system. It is given by $b = b_{gs}||b_{ap}$

Part of the behavior of the goal system becomes the i_g part of the input to the adaptation process. This is the part of the behavior that the adaptation process monitors, for example using sensors and software monitors. The part of the behavior b_{gs} that becomes i_g is determined by a filtering function f_g, as in $i_g = f_g(b_{gs})$. Part of the behavior of the adaptation process s_{ap} becomes the i_a part of the input to the goal system. This is determined by a filtering function f_a, as in $i_a = f_a(b_{ap})$.

Performance There is a performance function p that provides a performance measurement for the behavior of system s and a set W that contains all acceptable performance values. According to Martín's definition, the goal system is responsible for the goal behavior of the entire system, while the adaptation process is solely responsible for adapting the goal system to be more effective. What follows from this is that the outcome of the performance function p is related only to the behavior of the goal system. Therefore, we ignore the behavior of the adaptation process as part of the input to p and use only the behavior of the goal system as input to p.

Some Input Sets Suppose $\mathcal{I}_{gs}$ is the set of all possible inputs to the goal system s_{gs}. Not all inputs in this set lead to acceptable behavior of the goal system. The set of inputs for which the goal system gives acceptable behavior is given by[3]:

$$I_{gs}^{acc} = \{i_{gs} \in \mathcal{I}_{gs} | p(s_{gs}(i_{gs})) \in W\}$$

Given a certain goal system s_{gs}, then the acceptable behavior to this goal system is represented by I_{gs}^{acc}. The goal system is part of a system s. The set of all inputs this system s can accept is represented by I^{acc}. I^{acc} is restricted by the goal system and can maximally be I^{accMax} (i.e., $I^{acc} \subseteq I^{accMax}$ for any system s that contains a given goal system s_{gs}), where I^{accMax} is defined as:

$$I^{accMax} = \{i \in \mathcal{I} | \exists i_{gs} \in I_{gs}^{acc} i_{gs}[1..k] = i\}$$

Note that, although the system can give acceptable behavior for each input from this set, it is not guaranteed that the system does give acceptable behavior. It depends on the specific i_a provided to the goal system s_{gs}, thus on the specific adaptation process used. But in general, it would be possible for any given $i \in I^{accMax}$ to create an adaptation process s_{ap} for which the combined system (of this adaptation process and the given goal system) gives acceptable behavior when the input is i.

[3]Suppose i is a time-based function. The notation $i[m \dots n]$ represents a time-based function j that is constructed as follows: $j(t) = i[m \dots n](t) = [i(t)[m], i(t)[m+1], \dots, i(t)[n]]$. In other words, the output of j for a given time instance is the (m, n) subvector of the output of i for the given time-instance.

The set of actual inputs to the goal system is limited by the adaptation process, as this process determines the i_a part of the input. The set of actual inputs to the goal system is given by the following equation:

$$I_{gs}^{act} = \{i_{gs} \in \mathcal{I}_{gs} | i_{gs}[k+1 \dots k+m] = f_a(b_{ap})\}$$

This equation can be rewritten into the following equation (the derivation steps are given in Appendix B):

$$I_{gs}^{act} = \{i_{gs} \in \mathcal{I}_{gs} | i_{gs}[k+1 \dots k+m] = f_a(s_{ap}(i_{gs}[1 \dots k] || f_g(s_{gs}(i_{gs}))))\}$$

The set of actual inputs to the goal system that give acceptable behavior is determined by $I_{gs}^{actacc} = I_{gs}^{act} \cap I_{gs}^{acc}$. A measurement of how much this set differs from the acceptable input set I_{gs}^{acc} of the goal system is given by the function $diff(I_{gs}^{actacc}, I_{gs}^{acc})$. One possibility for this difference function is to calculate the difference in number of elements in the set. Different adaptation processes lead to different I_{gs}^{actacc}. The closer I_{gs}^{actacc} is to I_{gs}^{acc}, the more adaptive the system.

Instantaneous Cycle Note that in this example there is an instantaneous input-cycle between the goal system and the adaptation process; i_a depends on b_{ap} which depends on i_g which depends on b_{gs} which depends on i_a. In real systems, there is a time delay in such loops, i.e. $i_a(t) = s_{ap}(i(t) || f_g(b_{gs}(t - \delta t)))$.

1.1.2 Software Quality

This section gives an introduction into software quality. This thesis will explain later that implementing more advanced algorithms for adaptive systems can have a negative impact on certain aspects of software quality.

The ISO 9001 standard defines quality as "the degree to which a set of inherent characteristics fulfills requirements" [67]. McConnell states that software quality can be divided into internal quality characteristics and external quality characteristics [87]. External quality characteristics refer to those properties that can be observed by the users of the product, like conformance to the functional requirements. The internal quality characteristics cannot be observed by the users; they are internal to the product, such as the evolvability of the product. Higher complexity, caused by better algorithms for adaptivity, is correlated with better external quality, as the functionality of the system improves.

Software quality is a broad concept and can be divided into several quality attributes. A number of software quality models have been developed that try to make a classification of the different quality attributes. McCall's software quality model contains eleven important software quality factors (e.g., reliability, usability, efficiency, maintainability, reusability) divided along three important aspects of a software product (product operation, product revision and product transition) [86]. Each of these quality factors is again decomposed into quality criteria. For example, the *maintainability* quality factor is decomposed into the quality criteria *consistency*, *simplicity*, *conciseness*, *self-descriptiveness* and *modularity*. The ISO 9126 standard defines a

software quality model in which software quality is divided into a comprehensive set of six different characteristics (functionality, reliability, usability, efficiency, maintainability, portability), each having a number of sub-characteristics [68]. For example, maintainability has the sub-characteristics *analyzability*, *changeability*, *stability*, *testability* and *maintainability compliance*.

1.1.3 (Software) Complexity

This section discusses complexity, especially *software complexity*. More advanced algorithms for adaptive systems can increase the complexity of software. An increase in software complexity can have a negative impact on software quality.

Degrees of Complexity

According to the Merriam-Webster's Online Dictionary, complexity is defined as the "quality or state of being hard to separate, analyze, or solve" [88, 89]. In [107] Weaver describes the scientific methods needed to deal with problems of varying complexity. He classifies problems according to three categories of complexity: *simplicity*, *disorganized complexity* and *organized complexity*. Problems falling in the category simplicity have only a few variables with clear relationships. For example, the problems classical physics addresses are of this type (e.g., the relationship between force applied to an object, the mass of the object and the acceleration of the object). Such problems can be easily analyzed and solved by humans.

Problems of *disorganized complexity* are "problems in which the number of variables is very large, and one in which each of the many variables has a behavior which is individually erratic, or perhaps totally unknown" [107]. As such, it is not possible to analyze the behavior of each individual variable. But it may still be possible to analyze properties of such a system as a whole, e.g., using statistical techniques. An example of a problem (or system) of disorganized complexity could be a volume of gas, which contains billions of molecules. It is intractable to analyze and predict the behavior of each individual molecule in the volume, but it is possible to analyze certain global properties of the gas, such as the average speed of the molecules, how often on average a molecule collides with another molecule and the pressure of the gas.

Problems of *organized complexity* fall between problems of simplicity and problems of disorganized complexity. They have more than a few variables, but with the help of advances in computer science and the computing power of modern machines, it becomes possible to analyze each individual variable [107]. As we will see later, the complexity this thesis addresses falls within the category of organized complexity. Although computing machines can handle systems with this type of complexity, humans (e.g., engineers) have difficulty understanding and implementing them.

Types of Software Complexity

According to Fenton and Pfleeger, software complexity can be divided into multiple categories: problem complexity, algorithmic complexity, structural complexity and cognitive complexity [48]. *Problem complexity* and *algorithmic complexity* mainly

deal with execution efficiency. *Structural complexity* refers to the complexity of the programming structures used to implement the algorithm, such as the control flow and the components and their connections. *Cognitive complexity* refers to how hard it is for people to understand the software [48]. In this thesis we study how more advanced algorithms for adaptive systems influence the structural complexity of the software. Problem complexity and algorithmic complexity of these algorithms are not the main focus of this thesis. Furthermore, we do not explicitly address cognitive complexity.

Essential and Accidental Complexity

Brooks was one of the first to describe in [24] a division of the complexity of software into *essential complexity* and *accidental complexity*. *Essential complexity* is complexity that is inherently part of the task to be performed, the problem to be solved or the solution found. Essential software complexity originates from "the fashioning of the complex conceptual structures that compose the abstract software entity" [24]. If a software system solves a certain problem or executes a certain task, the essential complexity inevitably becomes part of the software; it cannot be prevented.

Accidental complexity is complexity due to a difference between the conceptual abstractions of a problem to be solved or task to be performed (i.e. the essential complexity) and the representation of these conceptual abstractions in the chosen programming language [24]. This can be caused by the inability of the chosen programming language to make the appropriate abstractions. Thus a mapping of the conceptual abstractions to the available implementation abstractions needs to be made, leading to additional complexity. An additional cause for accidental complexity is the inappropriate selection of implementation abstractions from the available set of implementation abstractions, adding complexity that could have been prevented by selecting a better set of implementation abstractions. Accidental complexity can be prevented by using/developing implementation abstractions that correspond to the conceptual abstractions of the problem to be solved or task to be performed.

In this thesis we consider both essential and accidental complexity in adaptive systems.

Managing the Complexity of Embedded Control Software

A number of modeling approaches have been introduced to manage the complexity of embedded control software. One of the first popular modeling approaches is the Ward/Mellor method [106], which is an extension for embedded systems of the Yourdon structured method [112]. The Ward/Mellor method provides a number of different model types to model a system, which include entity-relationship diagrams, data-flow diagrams and state-transition diagrams.

With the transition to object-oriented software development, new methods for modeling embedded control software were introduced. The current state-of-the-practice is UML-RT [38, 96], which is an extension of UML for embedded systems. UML-RT supports, among others, capsules to model structure and state-charts to specify the dynamic behavior of capsules. UML-RT is supported by tools such as Rational Rose Technical Developer [5].

1.2 Context: Adaptive Embedded Systems

The previous section introduced general concepts used in this thesis. This thesis is based on work done in the Octopus project [45], where Océ-Technologies B.V. (one of the world's leading manufacturers of professional printer and copier systems) is the carrying industrial partner. The main goal of this project is to study and develop structured methods and techniques that enable engineers to design more adaptive embedded systems. This section explains the goals of this project, to provide the context for the work in this thesis. First, we provide a motivation for adaptivity in embedded systems. Then some concepts concerning the behavior of a physical system are explained. Finally, some strategies to make embedded systems more adaptive are discussed, summarizing work in the Octopus project.

1.2.1 Embedded Systems and the Motivation for their Adaptivity

Embedded systems, such as digital document printing systems, can conceptually be separated into a physical system capable of executing certain physical behavior and a control system that determines the actual behavior of the physical system. The interface between the physical system and the control system consists of sensors and actuators. Under competitive market conditions, embedded systems are designed to cope with a variety of conditions. This includes the following types of variations:

Tasks

The tasks that the physical system has to perform can be subject to variation or change. For example, in printing systems, the paper type and weight may vary, the job data may vary (e.g., color print, black-and-white print), etc.

Environmental Conditions

The system needs to adapt to changes in environmental conditions and to constraints from the environment. Examples of these conditions and constraints in printing systems could be:

- Weak mains: The power provided by the mains might not be sufficient for the system to provide maximal performance. The system should adapt to weak mains by reducing its energy consumption. There might be several possibilities to realize lower energy consumption, for example lowering the printing speed.
- Environment temperature: If the environment temperature is different, the amount of power provided to the heaters of the printer system also needs to be different, to provide the necessary and sufficient amount of heating.

User preferences

The system needs to adapt to changes/differences in user preferences. These preferences may involve trade-offs between several system aspects, such as productivity, energy consumption and quality.

Legislation

In certain countries there is legislation that requires the system to adapt to meet this legislation. Examples include legislation on noise levels, on energy

consumption, etc.

Deterioration

During its life span, the physical system is subject to wear and tear. For example, a heater in a printer system gets less efficient over time, because of accumulating dirt. This changes the behavior of the printer system. The system needs to adapt to cope with these changes, to maintain acceptable print quality.

Failure

Sudden failures can happen in the physical system. For example, in printing systems there can be a paper jam, a component that stops functioning, lack of toner or paper, etc. Failures prevent the system from executing its task. In some cases the system can continue executing its task, by making use of redundancy in the system. For example, a printer system has multiple heaters. If one of these heaters is broken, the printer system may be able to continue printing using the remaining heaters.

1.2.2 Behavior of a Physical System

This section introduces a conceptual view on an embedded system and its behavior. This conceptual view is used in the next section to explain how an embedded system can be designed to be more adaptive, to cope with variety in the conditions.

As mentioned before, an embedded system s can be divided into a physical system s_{ps} and a control system s_{cs}. In the following we define three behavior spaces of the physical system, the *possible behavior space*, the *desirable behavior space* and the *actual behavior space*. Figure 1.2 schematically shows these three behavior spaces and their relationship.

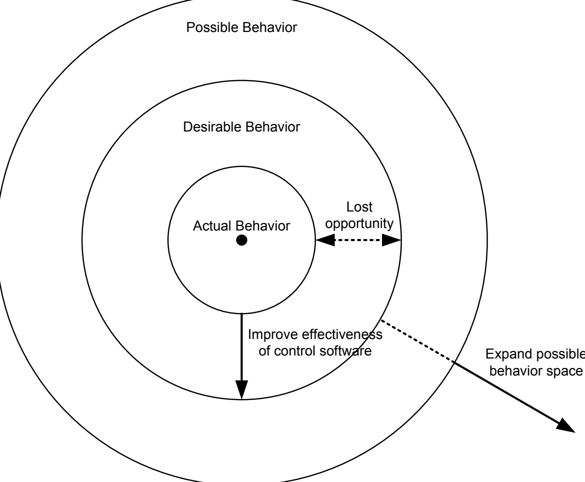


Figure 1.2: Schematic overview of the three behavior spaces of a physical system

Possible Behavior Space

The behavior that is possible with a physical system is bounded by its physical limitations and the limitations of the actuators. The *possible behavior space* contains all behavior possible with the physical system. Using Zadeh's definition of an adaptive system (see Section 1.1.1), the possible behavior space of the physical system s_{ps} is $B_{ps}^{pos} = s_{ps}(\mathcal{I}_{ps})$, where $\mathcal{I}_{ps}$ is the set of all input functions that can be accepted by s_{ps}. In Figure 1.2, the outer ring represents the possible behavior space.

Desirable Behavior Space

Not all behavior in this possible behavior space is desirable behavior. Desirable behavior of the physical system is behavior that the customer expects. For example, most printing systems are physically capable of behaving in such a way that paper will get jammed in the printing system, e.g. by allowing too little space between the sheets of paper. However, this is not desirable behavior of a printing system. Therefore, such behavior should be prevented. The *desirable behavior space* contains all desirable behavior of the physical system. Using Zadeh's definition, this desirable behavior space is defined by a given performance function p and a given acceptable set of performance values W. The desirable behavior space is defined by:

$$B_{ps}^{des} = \{s_{ps}(i) | i \in \mathcal{I}_{ps}, p(s_{ps}(i)) \in W\}$$

In Figure 1.2, the middle ring represents the desirable behavior space.

Actual Behavior Space

The control system determines the actual behavior of the printing system. As such, the goal of the control system is to stay within the boundaries of the desirable behavior space. The actual behavior space contains all behavior imposed on the physical system by the control system. This actual behavior space is defined by $B_{ps}^{act} = f_{ps}(s(\mathcal{I}))$, where $\mathcal{I}$ is the set of all possible input to the system s and f_{ps} is a filtering function that filters the part of the behavior that belongs to the physical system from the behavior of the entire system, as produced by $s(\mathcal{I})$[4].

In Figure 1.2, the inner-most ring represents the actual behavior space. Note that the figure shows the actual behavior space as a subset of the desirable behavior space. But, according to the definition this is not necessarily the case. In practice, however, control systems are designed to produce desirable behavior for the vast majority of the input (i.e., the actual behavior space is by approximation a subset of the desirable behavior space).

Limited Actual Behavior Space and Lost Opportunity

Current conservative engineering techniques, can result in an actual behavior space that is a limited subset of the desirable behavior space. Reasons for this include:

[4] Note the analogy between the application of Zadeh's definition to the system s that is separated into a physical system and a control system, and the application of Zadeh's definition to a system separated, according to Martín's definition, into a goal system and an adaptation process, as was discussed in Section 1.1.1.

- The precise state of the physical system and its environment is not known to the control system. It has only sensor information and this gives a limited view on the state. Also, the perception of the sensor information might be limited, not making complete use of all information available. The control software can only make decisions based on this limited information. Therefore, safety margins are build in the control software to guarantee correct operation of the system under reasonable ranges of unknown state variables.
- The used control algorithms are limited. They might be imprecise in determining the best behavior for certain circumstances. It might be that control algorithms with more desirable properties do not exist. It might also be that they do exist, but are not implemented due to their complexity. In this case a design trade-off has been made between functionality of the control system and engineering complexity. The complexity may be inherent in the control algorithm or may be incidental, e.g., because the control concepts cannot be directly mapped to the implementation mechanisms of the chosen programming language.
- There is limited knowledge about the desirable behavior space; parts of this space and its boundaries may be unknown. Also, the desirable behavior space may change under changing circumstances, e.g., changing input, user needs, environmental conditions. Conservative control algorithms are designed to stay within the known area of the desirable behavior space.
- The actual behavior of a control algorithm might be difficult to determine for all possible circumstances. As such, it is hard to estimate whether a certain control algorithm stays within the desirable behavior space. Therefore, a conservative control algorithm may have been selected, to reduce the possibility that the actual behavior is outside the desirable behavior space.

The gap between the possible behavior space and the actual behavior space results in lost opportunity: the system could perform more effectively if a larger area of the desirable behavior space is exploited. For example, printing on lighter paper consumes less energy. The spare energy could be utilized to increase the printing speed. If the control system has been designed to print at a fixed speed, this opportunity to increase the speed when printing on lighter paper is lost.

1.2.3 Adaptivity: Increasing Actual Behavior Space

The goal of the Octopus project is to develop structured methods and techniques *i*) to increase the possible behavior space, resulting in an increase of the desirable behavior space, allowing less conservative control algorithms and thus increasing the actual behavior space, and *ii*) to increase the actual behavior space, thus making more optimal use of the available desirable behavior space. Examples of improvements could be:

- Techniques to improve knowledge about the system state, e.g.:
  - Additional sensors, better sensors and smarter placement of sensors

  - Derivation of more information from existing sensors, e.g., using probabilistic modeling, qualitative physical modeling, quantitative physical modeling, as in [50, 64].

- More powerful control algorithms, that make better use of the desirable behavior space, e.g., fuzzy control algorithms, model predictive control algorithms, as in [26, 46].

- Better tools to analyze the behavior spaces.

  - To determine whether the actual behavior space is a subset of the desirable behavior space, as in [17]. This reduces the large safety margins taken into account because of a lack of knowledge about the behavior of the control system.

  - To analyze the possible behavior space and determine possibilities to increase the possible behavior space, as in [17, 75].

- Better software engineering techniques to reduce the accidental complexity and better manage the essential complexity of the control algorithms. This enables more advanced/powerful algorithms for adaptivity.

This thesis focuses on better software engineering techniques to reduce the accidental complexity and better manage the essential complexity of control algorithms.

1.3 Industrial Case Studies

In this section we introduce two industrial case studies, taken from the digital document printing systems domain. These case studies have been developed and evaluated within the context of the Octopus project [45]. These case studies will be used throughout this thesis to illustrate the challenges and solutions. Each case describes *i)* an overview of the controlled hardware, *ii)* two control strategies, the second one resulting in a more adaptive system than the first one.

1.3.1 Case Study I: Warm Process

The first case involves a part of the printing process in digital document printing systems called the *Warm Process*. This process is responsible for transferring a *toner image* to paper.

System Description

Figure 1.3 gives a schematic overview of the parts in the printing system responsible for the warm process behavior. The warm process has two main parts; a *paper path* to transport sheets of paper and a *toner belt* to transport toner images. The *contact point* is the location where the paper path meets the toner belt. At this location the toner image is transferred from the toner belt to the sheet of paper. For correct printing, both the sheets of paper and the toner belt should have a certain temperature at the

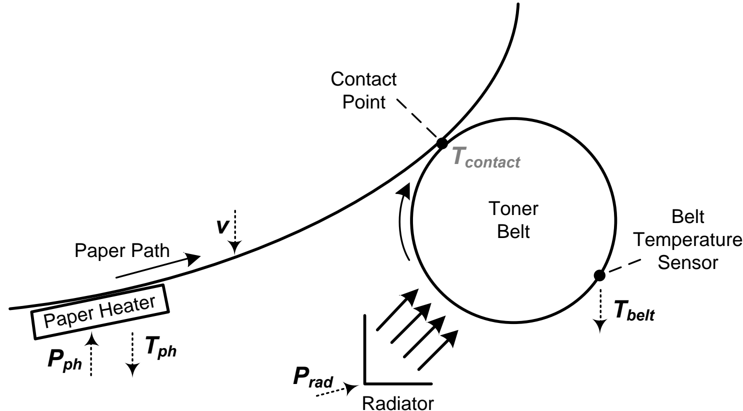


Figure 1.3: Schematic view of the *Warm Process*

contact point. Therefore, the warm process contains two heating systems; a *paper heater* to heat the sheets of paper and a *radiator* to heat the toner belt. In addition, the physical system provides the following sensors and actuators:

- T_{ph}: Sensor that measures the paper heater temperature.
- T_{belt}: Sensor that measures the temperature at the sensor location on the toner belt.
- v: Actuator to set the printing speed[5].
- P_{ph}: Actuator to set the amount of power supplied to the paper heater.
- P_{rad}: Actuator to set the amount of power supplied to the radiator.

Note the variable $T_{contact}$ shown in the figure. This variable represents the temperature of the toner belt in the contact point. There is no sensor in the system to directly measure this value.

In the following, two control systems for this example case are described. Together, they form an evolution scenario; first, control system 1 was applied to the system. To increase the actual behavior space, control system 1 was changed to control system 2.

Control System 1

In the first example control system for the Warm Process case study a lot of properties were fixed at design time. While the physical system allows variable speed, the speed

[5]Note that this only gives a simplified and abstract view of a printing system. In reality, there is no single actuator in the physical system to set the printing speed. Instead, the paper path consists of multiple motors and pinches, which each can be controlled independently. If this control is done in a coordinated way, this leads to correct paper transportation at a certain speed. Note that a virtual speed actuator can be implemented in the control software. The control logic in the implementation of the virtual speed actuator takes care of controlling the different motors in the paper path to obtain the requested speed.

was fixed at a specific value by the control system. This enabled the engineers to find proper setpoints for the paper heater temperature (T_{ph}) and the belt temperature (T_{belt}), to let the system operate acceptably under a wide variety of circumstances (i.e., safe margins are taken into account for the unknown/independent variables). The control system contains classic controllers to maintain these temperature set-points, by controlling the power given to paper heater and the radiator respectively. Figure 1.4 shows the different tasks implemented by the control software and the dependencies between them. `PaperHeaterController` and `RadiatorController` are the two control tasks. `PhysicalSystemI/O` takes care of interfacing with the sensors and actuators. `SetpointConfig` provides the setpoints T_{ph}^{sp}, T_{belt}^{sp} and v.

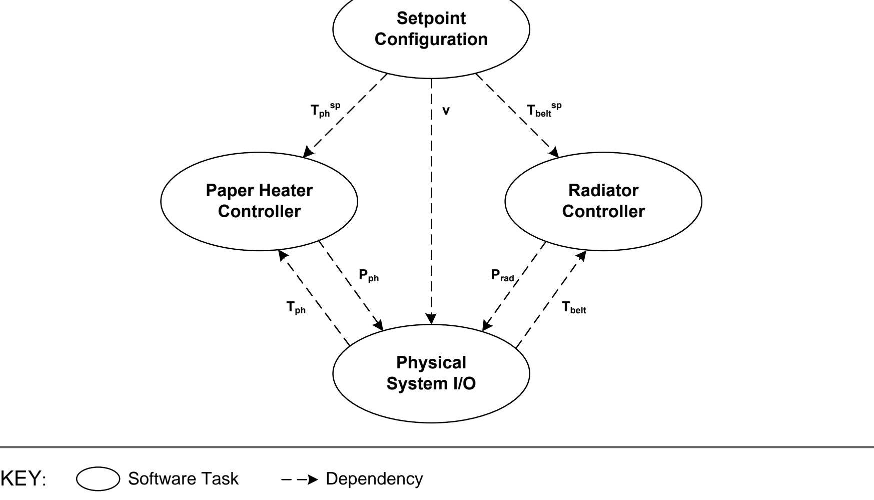


Figure 1.4: Software dependencies in control system 1

Fixing variables in the physical system simplifies the design of the control system. Engineers can easily ensure, for example through experimentation, that the system operates sufficiently under a wide variety of circumstances if the variables are at the given setpoints. The dependencies in the control software are easily understood. This simplifies the implementability and maintainability of the control software.

Control System 2

In the second iteration of the control system, the speed is made flexible, i.e. it is possible to change the speed. For example, it can be lowered to reduce energy consumption or raised when printing on lighter paper (to increase productivity). In this way, the actual behavior space of the system has been increased.

If the speed of the system changes, the temperatures of the paper heater and belt also need to change, to maintain correct print quality. Engineers identified a relationship between the three variables speed (v), paper heater temperature (T_{ph}) and the temperature of the belt at the contact point ($T_{contact}$), that ensures correct

print quality if the relationship holds. The relationship is given by the following equation (c_1, c_2 and c_3 are constants):

$$T_{contact} \overset{desired}{=} c_1 \cdot v - c_2 \cdot T_{ph} + c_3 \quad (1.1)$$

The paper heater reacts slowly to changing temperature setpoints, while the radiator can quickly influence $T_{contact}$. Therefore, engineers decided to mainly use the radiator to adjust the temperatures when the speed changes. This means that Equation 1.1 is used to determine the required $T_{contact}$ (i.e., the setpoint).

As there is no sensor to directly measure $T_{contact}$, engineers had to identify the relationship between $T_{contact}$ and T_{belt}. They found the following equation (in which c_4 is a constant):

$$T_{contact} = c_4 \cdot \frac{P_{rad}}{\sqrt{v}} + T_{belt} \quad (1.2)$$

Figure 1.5 shows the control software tasks and their dependencies. The `PaperHeaterController` represents the same task as in control system 1. There is a dependency with `SetpointConfiguration`, to obtain the setpoint for T_{ph}. The `RadiatorController` now implements a classic control algorithm to control $T_{contact}$ instead of T_{belt}. It gets its setpoint from `PrintQuality`, which contains Equation 1.1. The current $T_{contact}$ is provided to `RadiatorController` by `BeltTemperature`, which contains Equation 1.2. The speed of the system is set by `PaperPathPrintSpeed`. How the required speed is determined is not relevant.

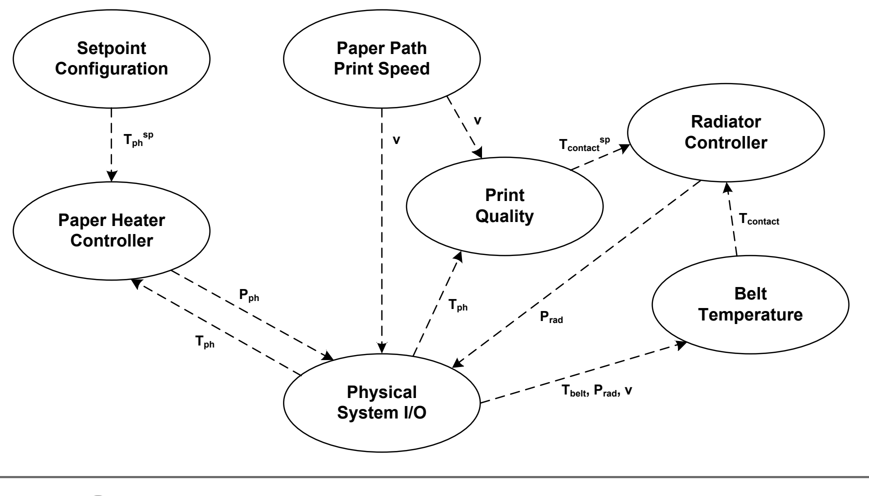


Figure 1.5: Software dependencies in control system 2

1.3.2 Case Study II: Drum Shuttling

The second industrial case is the Drum Shuttling subsystem of a printing system. The drum is a rotating cylindrical component in the printer system, on which the toner image is created. To reduce deterioration, the drum also has to shuttle (i.e., move backward and forward) along its axis.

System Description

Figure 1.6 schematically shows the drum and additional components needed to rotate and shuttle the drum.

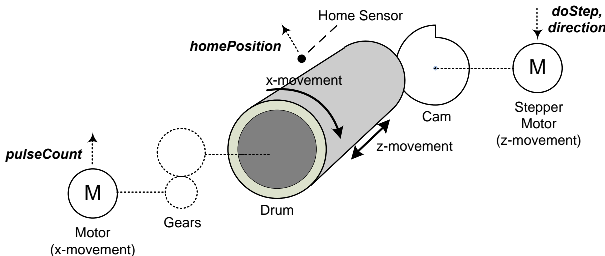


Figure 1.6: Schematic view of the drum and components for rotation and shuttling

There is a motor and gears for the rotational movement of the drum. We call this rotation *x-movement*, and the corresponding distance traveled by the surface of the cylinder the *x-position*.

The linear shuttling movement of the drum is provided by a stepper motor (i.e. a motor that rotates in fixed sized steps) and a cam, which is a component that can translate rotational movement into linear movement. We call this linear movement *z-movement*, and the corresponding position of the drum relative to the *home position* the *z-position*.

The physical system provides the following sensors and actuators:

- $pulseCount$: Sensor that counts and provides the number of hall pulses of the motor for x-movement. On each revolution, the motor gives a fixed number of hall pulses, so $pulseCount$ is proportional to the number of revolutions made by the motor.

- $homeSensor$: Sensor that gives a signal when the drum is at a specific z-position (the home position).

- dir: Actuator to set the direction in which the stepper motor should step.

- $step$: Actuator that executes one step of the stepper motor when a signal is provided.

Control System 1

In the first control system, a controller actuates the stepper motor to do steps at a fixed frequency. The controller takes care that the direction of stepping is reversed if the edge of the interval has been reached. Figure 1.7 shows the dependencies between the different tasks in the control software. It shows the task `PhysicalSystemI/O`, which provides access to the sensors and actuators, and the task `ShuttlingController`, which implements the fixed-frequency stepping control logic.

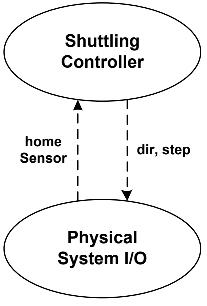


Figure 1.7: Software dependencies in control system 1

Although this example control system is simple and easy to implement, there are some issues with fixed-frequency actuating. For example, if the system is running at a lower speed, there are more steps performed than necessary (because the fixed frequency is chosen to accommodate the highest possible speed), leading to a reduced productivity, as a sheet cannot be printed at the moment a step is performed. The second example control system overcomes these issues.

Control System 2

In the second control system, the z-movement is performed relatively to the x-movement. This means that when the system is running at lower speed, z-movement is also slower, meaning that less steps are performed per time unit. As no sheets can be printed when a step is performed, less steps per time unit is beneficial for the productivity. There is a function $f_{shuttling}$ to determine $zPos$ from a given $xPos$: $zPos = f_{shuttling}(xPos)$. In principle, this is a linear function with the additional behavior that the slope is reversed at the boundaries of $zPos$ (to reverse the direction of movement).

For economical reasons there is no sensor to directly measure $xPos$. Instead, the pulses that the drum motor gives at a fixed rate per revolution are counted ($pulseCount$). The relationship between $pulseCount$ and $xPos$ (rotation relationship) is described by the following equation:

$$xPos = \frac{pulseCount}{C_{pulsesPerRev}} \cdot C_{gearTransm.} \cdot C_{drumCircumf.} \qquad (1.3)$$

Once the $xPos$ is obtained, it is transformed into a desired $zPos$ using the function $f_{shuttling}$. As there is no actuator to directly set the $zPos$, the desired $zPos$ needs to be translated into a desired *stepPos* of the stepper motor. The relationship between $zPos$ and *stepPos* (shuttling relationship) is given by the following equation:

$$stepPos = \frac{zPos}{C_{degreesPerStep} \cdot C_{zMovementPerDegree}} \tag{1.4}$$

Figure 1.8 shows the different tasks in the control software and their dependencies. It shows the task `RotationRelationship` to transform *pulseCount* into $xPos$, the task `ShuttlingController` which performs the function $f_{shuttling}$ and the task `ShuttlingRelationship`, which transforms the desired $zPos$ into a desired *stepPos*. The figure also shows the task `StepperController`, which performs the stepping of the stepper motor to the desired *stepPos*, by using the actuators *dir* and *step*. It uses *homeSensor* to calibrate its internal representation of *stepPos* with the actual step position (this is necessary as the stepper motor might not always perform a step when actuated).

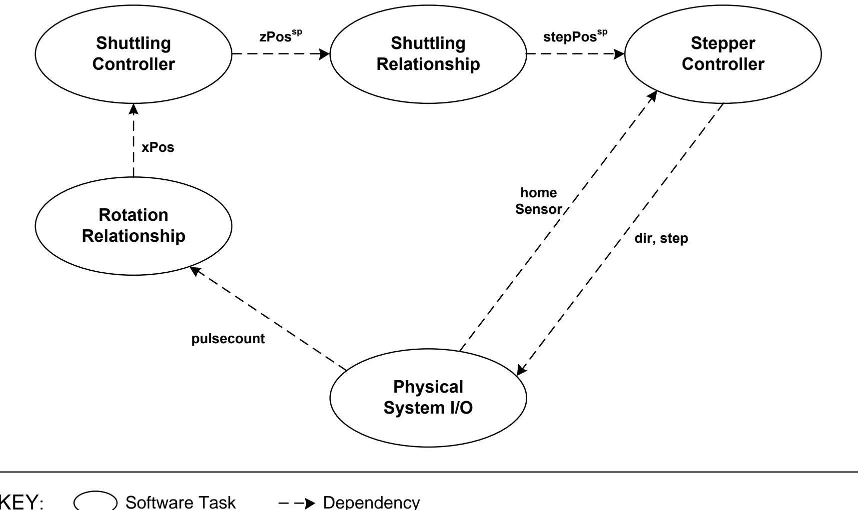


Figure 1.8: Software dependencies in control system 2

1.3.3 Software Challenge

For both example cases two control systems are presented. In both cases, the second control system makes better use of the possibilities of the physical system. However, the complexity of the control software has increased; there are more tasks and more dependencies between the tasks. This example shows the trade-off between increasing the actual behavior space with smarter control algorithms and less complex software.

1.4 Motivation & Objectives

Making systems more adaptive is desirable, as it improves the ability to function in a wider range of circumstances, which can be translated into features or quality improvement. However, to make systems more adaptive, more sophisticated control algorithms are necessary. As shown in the example cases, more sophisticated control algorithms typically increase the complexity of the control software. An increase in complexity reduces software qualities such as understandability, maintainability and evolvability. Control software becomes more costly to implement and maintain. Therefore, engineers make trade-offs between the sophistication of control algorithms and software quality. This thesis provides techniques *i)* to better manage essential complexity and *ii)* to reduce or avoid accidental complexity for several types of adaptivity strategies. These techniques help to reduce the impact of more sophisticated control algorithms on the software quality, which enabled engineers to develop increasingly sophisticated adaptive systems.

One strategy to increase the adaptivity of the system is to apply more knowledge about the physical characteristics of the system in the control logic. An example of this is shown in the Warm Process case study, which was introduced in Section 1.3.1. The case study shows two versions of the control logic. In the second version, the control software explicitly applies knowledge about certain physical characteristics in the system, to more precisely control the system. This results in a more adaptive system, but also in higher complexity of the control software. Therefore, the first objective of this thesis is:

1 How to manage the (essential) complexity introduced by physical models in embedded control software, while reducing or eliminating accidental complexity introduced by the implementation of these physical models.

Implemented models of physical characteristics may not always accurately reflect physical reality, e.g., because the physical system has evolved, the system is used in different circumstances than it was tested for, the physical system has changed because of wear and tear, etc. As inaccuracies in the models of physical characteristics may lead to incorrect behavior of the system, the accuracy of the models needs to be verified. One cannot test or statically verify the system for all possible conditions, thus runtime verification is necessary. As will be explained in Chapter 3, traditional runtime verification techniques cannot be applied to check the correspondence of implemented models of physical characteristics with physical reality. Traditional runtime verification techniques check the correspondence of a software implementation with a software model, instead of checking the correspondence of a model implemented in software (e.g., a model of physical characteristics) with the modelled element (e.g., physical reality). Therefore, the second objective of this thesis is:

2 How to verify at runtime that the implemented models of physical characteristics conform to physical reality.

A second strategy to increase the adaptivity of the system is to apply higher-level control algorithms. In simple control systems, control components focus on a localized

task, e.g., maintaining a specific temperature with a localized heating component in the system. This type of control provides localized optimal behavior, but the system as a whole might not behave optimally. A higher-level control algorithm controls a larger part of the system. In this way, the system as a whole can be controlled to a global optimum. An example of higher-level control is control that performs multi-objective optimization of system qualities, such as energy consumption, productivity and quality. As these qualities are not localized to a specific part of the system, but arise from the system as a whole, the control algorithm has an impact on the entire system. Algorithms to perform multi-objective optimization already exists. What is lacking is a structured method to design a control system that applies such algorithms. Because of the system-wide impact, ad-hoc implementations could result in tangling of the algorithm with lower level control logic, partial implementations and inconsistencies in the application. The third objective of this thesis is:

3 How to design and implement multi-objective optimization functionality in embedded control software in a systematic way.

1.5 Solutions and Contributions

This thesis provides techniques that addresses the three objectives described in the previous section. Chapter 2 analyses in depth how models of physical characteristics become part of control software and how this affects software quality. The chapter then addresses these issues by using domain-specific modeling languages to model the physical characteristics and provide mechanisms to compose at a conceptual level the models of physical characteristics that are specified in a domain-specific modeling language with general-purpose programming language modules that contain the other control logic. This addresses the first objective of this thesis.

Novel runtime verification techniques to perform runtime verification of implemented models of physical characteristics are presented in Chapter 3. This addresses the second objective of this thesis.

Chapter 4 analyses multi-objective optimization and provides a structured method to design and document multi-objective optimization within the architecture of control software, addressing the third objective of this thesis.

In Chapter 5 the techniques introduced in Chapters 2 to 4 are evaluated. An experiment is performed in which four embedded control software implementations are compared concerning the resulting print productivity, using a realistic Simulink model of the Warm Process part of the printer system and different simulation scenarios in which the amount of available power is limited and fluctuating. One of the embedded control implementations utilizes the techniques presented in Chapter 4 to implement multi-objective optimization and the techniques presented in Chapter 2 to compose models of physical characteristics with other software modules. This software implementation is compared to three other software implementations that use other techniques, which we have witnessed in practice, to determine optimal speed of the system in a power constrained environment. This experiment demonstrates the benefits of the techniques presented in this thesis concerning the functionality of the software. Chapter 5 also discusses different realistic evolution scenarios and

provides a qualitative evaluation of the benefits of the techniques presented in this thesis concerning software quality characteristics such as maintainability, evolvability and comprehensibility.

Contributions of this Thesis

Chapter 1

- A novel, precise definition of adaptive embedded systems.

Chapter 2

- How to compose models of physical characteristics, written in a domain-specific modeling language with the modules written in a general-purpose programming language, using aspect-oriented composition techniques.

Chapter 3

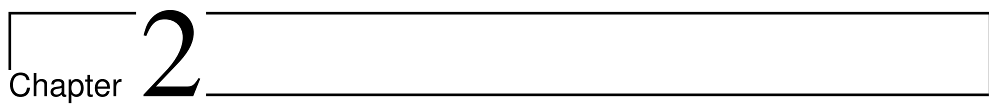
- A novel technique to verify at runtime the consistency of the models of physical characteristics used in control software with the physical reality.
- How to apply aspect-oriented techniques to create monitors for this verification and to handle detected inconsistencies.

Chapter 4

- A novel systematic method to design and document multi-objective optimization within the architecture of complex control software. This includes modeling and analysis techniques, code generation & weaving, and tool support.

Chapter 5

- The evaluation of the techniques presented in Chapters 2, 3 and 4, by designing and implementing a system using the given techniques. Evolution scenarios are used to qualitatively evaluate the benefits of the techniques with respect to the software quality.



Composing Domain-Specific Physical Models with Embedded Control Software[1]

2.1 Introduction

A considerable portion of software systems today are adopted in the embedded domain (e.g., medical equipment, military applications, traffic control systems, consumer electronics). A major portion of embedded software systems aim at controlling physical systems in some way, and as such, part of the characteristics of the physical systems must be represented in embedded control software accordingly. For example, in state-of-the-art printing systems, the speed of the machine is adapted according to the user needs, available power, temperature measurements obtained from several components and the heat capacity of these components. The embedded software has to consider the physical characteristics of the components involved to determine what kind of behavior is required for the physical system (e.g., what should be the speed and temperatures of the printing system to attain sufficient print quality). After this, the embedded software can select and apply a control strategy to obtain this required behavior.

In embedded systems development, the current practice is to use general-purpose programming languages (GPLs) such as C and C++. If the code that deals with the physical characteristics is implemented using a GPL, however, the following issues can arise:

- A GPL has insufficient expression power to express the domain-specific abstractions. Instead, the domain-specific abstractions need to be transformed to semantically equivalent implementation abstractions in the GPL. This increases the complexity of the software system [24], which in turn might reduce the comprehensibility, maintainability and reusability of (parts of) the software system.
- There is a lack of effective static and dynamic domain-specific analysis techniques, because the domain-specific abstractions are lost during implementation

[1] A version of this chapter is under review for publication in a journal.

in the GPL. This can reduce the reliability of the software, as certain domain-specific issues cannot be detected.

The first problem is elaborated in Section 2.2. Chapter 3 deals with the second problem.

Instead of implementing embedded control software in a GPL, one might suggest to use a domain-specific modeling language (DSML), such as Matlab Simulink [20] or 20-Sim [1, 23]. These languages are very suitable to express models of physical characteristics and continuous control logic. However, control software for embedded systems usually also implements other application logic, such as logic to schedule (discrete) tasks, to recover from errors, and to monitor the available resources (e.g., the amount of toner, number of sheets of paper in the paper tray). DSMLs to model physical characteristics are not particularly suitable to express these types of functionality. Therefore, this functionality is usually, and more effectively, implemented in a GPL.

DSMLs, such as Matlab Simulink and 20-Sim, offer to possibility to generate GPL code modules from DSML models. However, these generated code modules often have limited interfaces; other software modules have to be fitted around them, leading to tightly coupled software modules. Furthermore, the generated code is not intended to be human readable and as such becomes a black box in software. Therefore, often it is decided to implement models of physical characteristics and continuous control logic in a GPL instead of using a DSML combined with code generation.

In this chapter, we propose a method to compose physical models specified in a DSML with software modules specified in a GPL at the abstraction level of both languages. We adopt the domain-specific modeling language (DSML) *SIDOPS+* of the *20-Sim* toolset [1] to express the logic that deals with physical characteristics of the system. To compose physical models specified in SIDOPS+ with other software modules specified in a GPL, we apply the Composition Filters model, implemented in the Compose* language and toolset [34]. The Composition Filters model enables loose coupling between software modules and physical models. Furthermore, we extended the Composition Filters model to enable interaction using messages and events for GPL modules and DSML models, respectively. To facilitate this, we defined an event model on the execution semantics of DSML models. As such, our approach combines the benefits (e.g., ease of realization, maintainability, reusability) of a DSML to implement models of physical characteristics, with the freedom of a GPL to implement other application logic, without making compromises to compose the two.

Figure 2.1 shows how the different concepts in our approach are related to each other. There are three types of development artifacts: application logic implemented in GPL modules, physical models specified in a DSML and composition filters specified in the Compose* language. The interaction between software modules and physical models is represented by messages and events. This interaction is enabled by a number of composition filters; these composition filters specify which events in the execution of the physical model are interesting (e.g., the change of the value of a physical variable) and they specify how these interesting events are processed (for example, the event is dispatched to a certain software module or the event is logged). We have defined an event model for physical models. This event model specifies which events in the execution of a physical model can be selected/quantified and what properties

these events have. The composition filters can use these properties to select events of interest. The GPL modules are executed by a GPL runtime environment. An interpreter has been implemented to execute the physical models and the composition filter specifications.

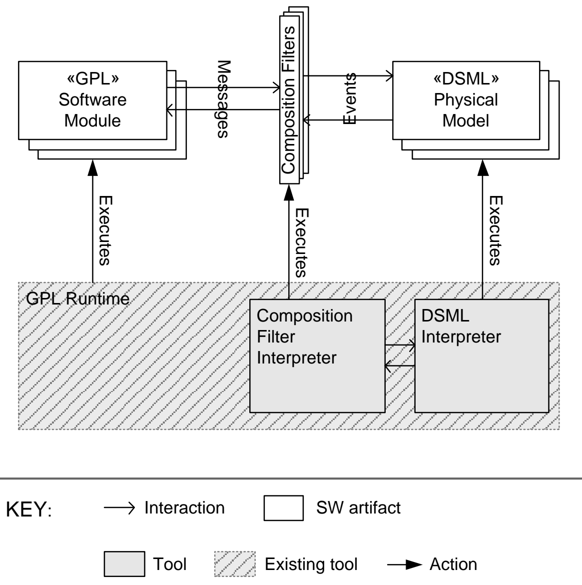


Figure 2.1: Overview of our approach

The main contribution of this chapter is a technique to compose physical models specified in a DSML with software modules specified in a GPL at the abstraction level of both languages, using the Composition Filters model.

The remainder of this chapter is organized as follows. Section 2.2 provides the problem statement. Next, an overview of our approach is given. This overview is followed by two sections that describe how physical models are represented and executed in embedded control software and how physical models can be composed with GPL software modules using the Composition Filters model. Section 2.6 discusses design details of the implemented DSML interpreter. This is followed by the application of our approach to the industrial case studies introduced in Chapter 1. Next, an evaluation of our approach is provided. The chapter ends with a discussion, related work and a conclusion.

2.2 Problem Statement

The case studies introduced in Chapter 1 show that models of physical characteristics (physical models) are part of the control logic implemented in embedded control software. Especially when systems are designed to be more adaptive, more physical models are implemented in embedded control software. However, such implemented

physical models introduce additional complexity in software. If this complexity is not managed properly, it will reduce software quality. This section analyzes the issues related to physical models that are implemented in embedded control software. First, the section briefly explains the current state-of-the-practice approaches. Then, it compares using a general-purpose programming language (GPL) versus using a domain-specific modeling language (DSML) to implement physical models. Benefits and drawbacks of both approaches are analyzed. Finally, this section summarizes characteristics of an approach that addresses the observed issues.

2.2.1 Current State-of-the-practice

Continuous Evolution

During the cooperation with our industrial partner, we have identified that the design of an embedded system evolves continuously due to changing customer needs, technological advances and cost reductions. Examples of evolution scenarios, in the context of the industrial case studies introduced in Section 1.3, are replacing the heaters with different types (having other properties) or changing the gears between the motor and the drum. These changes to the design of the embedded system affect the physical characteristics of the system. When such changes have to be realized, software engineers have to locate the corresponding pieces of code that are affected by these changes and modify them according to the new design.

Design Process

In current practice, two main steps are commonly adopted in the development of embedded systems. First, physical behavior is modeled and simulated using tools like 20-Sim [1, 23] and Matlab Simulink [20]. The purpose of this step is to analyze the behavior of the embedded control system through simulations. Second, an implementation of the embedded control software is realized based on the analysis results. For this realization, two techniques are used: Traditional programming techniques using GPLs like C and C++, and model-driven engineering techniques based on DSMLs for physical models and code generation from DSML models to implementations in a GPL. In the following subsections, we discuss issues and limitations regarding these alternatives and the combination of the two.

2.2.2 Development with GPLs

General-purpose programming languages lack abstractions specific for physical models (i.e., domain-specific abstractions). Therefore, when physical models are implemented in embedded control software, their domain-specific abstractions are translated to general-purpose abstractions in the GPL. The loss of domain-specific abstractions makes it harder to recognize physical models in embedded control software as such. We call this *implicit implementation* of the physical characteristics. This leads to the following issues:

- Translation of domain-specific abstractions to implementation abstractions might introduce accidental complexity [24]. Accidental complexity reduces comprehensibility and maintainability of software systems.
- Implicit implementation makes it hard to locate the implementation of a physical characteristics in case a change is necessary due to evolution. This in turn, increases maintenance costs and might reduce reliability (if not all relevant code is updated consistently).
- Because of implicit implementation, there are no clear boundaries between physical models and other application logic. This may lead to tangling and scattering of physical models in embedded control software. Example 2.1 shows tangling of physical models in the industrial case studies. As frequently claimed in the Aspect-Oriented Programming literature [49], tangling and scattering of *concerns* can lead to higher maintenance costs through reduced comprehensibility and evolvability of the tangled concerns.
- Implicit implementation hinders domain-specific analysis of the implemented physical characteristics for detection of faults, as domain-specific abstractions are lost. This makes it harder to ensure the reliability of the control software.

This chapter deals with the first three issues, Chapter 3 with the fourth issue.

Example 2.1 Tangling of Physical Characteristics

Listings 2.1 and 2.2 show for the two industrial case studies introduced in Section 1.3 an example implementation of the physical models in control software.

```
1  double PI_rad_I = 0d;
2
3  /* Control loop */{
4    // Get data using interfaces
5    double currentV = io.getV();
6    double Tph = io.getTph();
7    double Tbelt = io.getTbelt();
8    double newV = paperPath.getV();
9
10   // Print Quality and Belt Temp. physical characteristics
11   double TcontactSP = c1*newV - c2*Tph + c3;
12   double Tcontact = c4 * Prad / sqrt(currentV) + Tbelt;
13
14   // PI control logic
15   double deviation = TcontactSP - Tcontact;
16   PI_rad_I = PI_rad_I + deviation;
17   double PI_rad = c6 * deviation + c7 * PI_rad_I;
18
19   // Actuate Prad
20   double Prad = Math.min(Math.max(PI_rad, 0.0), 1500d);
21   io.setPrad(Prad);
22 }
```

Listing 2.1: Code fragment of a Radiator Controller module

Listing 2.1 is part of the Warm Process case study, which was introduced in Section 1.3.1. The listing shows an implementation of the control loop of the `Radiator Controller`. The physical characteristics that represent print quality (Equation 1.1) and belt temperature (Equation 1.2) have been implemented within the control loop on Lines 11 and 12 respectively. Lines 15 to 17 show the main functionality of the control loop: the feedback control logic (proportional-integral feedback control) that controls $T_{contact}$ to its setpoint using the actuator P_{rad}.

```
1  /* Control loop */{
2    double pulseCount = io.getPulseCount();
3
4    // Rotation physical characteristic
5    double xPos = pulseCount / CpulsesPerRev * CgearTransmission *
         CdrumCircumference;
6
7    // determine zPos:
8    double zPos = (c_ratio * xPos) % (2 * c_maxZPos);
9    if (zPos > c_maxZPos)
10     zPos = 2 * c_maxZPos - zPos;
11
12   // Shuttling physical characteristic
13   double stepPos = zPos / CdegreesPerStep / CzMovementPerDegree;
14
15   io.setStepPos(stepPos);
16 }
```

Listing 2.2: Code fragment of a Shuttling Controller module

Listing 2.2 is part of the Drum Shuttling case study, which was introduced in Section 1.3.2. The listing shows an implementation of the control loop of the `Shuttling Controller`. The physical characteristics that represent the rotation relationship (Equation 1.3) and the shuttling relationship (Equation 1.4) have been implemented within the control loop on Lines 5 and 13 respectively. Line 8 to 10 show the main functionality of the control loop, the logic that determines the required $zPos$ based on the current $xPos$. $zPos$ is linear in $xPos$, but reversing at the boundaries ($zPos = 0$ and $zPos = c_{maxZPos}$).

Both examples show the tangling of physical models with other control logic. Tangling reduces maintainability and comprehensibility of software systems [49].

2.2.3 Development based on DSMLs

Domain-specific modeling languages reduce or eliminate accidental complexity that is introduced by translating domain-specific abstractions to abstractions in the implementation language. This improves the maintainability of software, as has been proven in several publications such as [71, 103]. For instance, Matlab Simulink [20] and 20-Sim [1] provide DSMLs that are suitable to model physical characteristics, as well as continuous control logic. The accompanying tooling offers the possibility to compile models into executable software [1, 20]. The tooling can also apply domain-specific analysis to detect faults prior to code generation.

However, control software for embedded systems usually contains other

application-specific functionality that does not fall in the categories of physical models and continuous control logic. Examples of such functionality are managing system states (e.g., idle, start-up, available), scheduling of (discrete) tasks, recovering from errors, monitoring the available resources (e.g., the amount of toner, number of sheets of paper in the paper tray), processing of user input, security, communication, maintenance tasks, interaction with third-party libraries, etc. It is impractical to design or adopt a DSML to express these types of functionality, which are commonly expressed in a GPL.

As a result, GPL software modules generated from domain-specific models should be composed with other GPL software modules. This is an overwhelming task to perform manually because the interaction points of models with software are scattered throughout the software modules and the generated code is not (supposed to be) readable and maintainable. Even if the generated code modules are used in embedded control software, these modules become black boxes within the software architecture. Software engineers cannot easily understand and maintain them. Moreover, the software architecture gets constrained by the generated code: the other GPL modules have to be fitted around the generated modules, work-arounds need to be implemented if not all required interactions with the physical model are possible with the generated code, etc. Because of these limitations, in current practice the physical models are usually implemented and maintained in the GPL. DSML models are usually created by domain engineers for documentation and simulation purposes only.

To prevent the afore mentioned problems and to be able to exploit the created domain models in software, the transparent composition of DSML models with GPL modules should be facilitated.

2.2.4 Composing DSML and GPL Artifacts

Based on the observations noted in the previous section, it can be concluded that DSMLs are the preferred method to specify physical models. However, DSMLs are not used to implement physical models in embedded control software because of the limited ability to compose physical models specified in a DSML with other software modules specified in a GPL and the strict and tightly coupled interfaces resulting from existing code generation techniques.

An improvement of the current situation would be possible with an approach that offers flexible composition between physical models specified in a DSML and software modules written in a GPL, on the abstraction level of both the DSML and the GPL. Such a method combines the benefits of a DSML to implement physical models with the freedom of a GPL to implement other application logic. Furthermore, using a DSML provides separation of physical models from other application logic. Together with flexible composition operators, this separation prevents tangling and scattering of physical models with/through other application logic. Composition at the abstraction level of both languages eliminates the need to first translate DSML models to GPL modules (i.e., code generation), before the physical models can be composed with other software modules. This prevents tight integration caused by the limitations of generated interfaces.

2.3 Approach Overview

Figure 2.2 schematically shows our approach to deal with the issues introduced in the previous section.

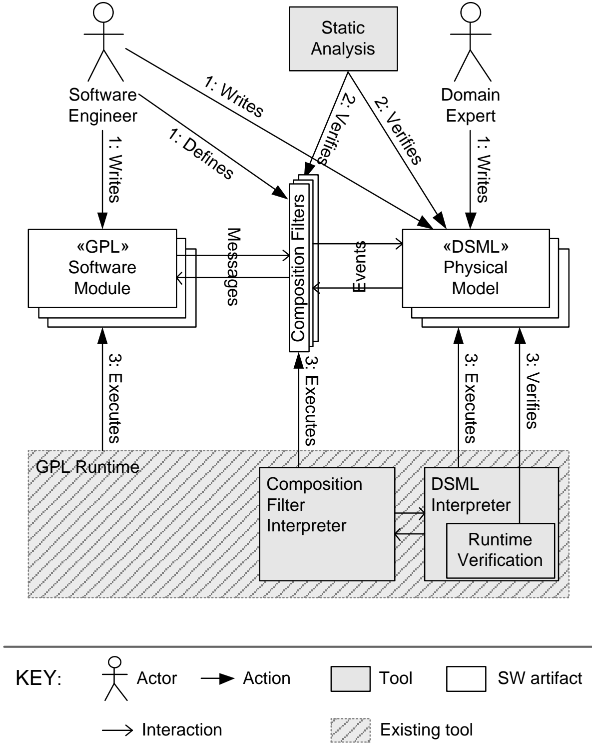


Figure 2.2: Detailed overview of our approach

The figure shows three different types of artifacts: software modules written in a GPL, physical models specified in a DSML and composition filters. The GPL software modules are written by software engineers. They are executed in a runtime environment for the GPL.

Physical models are specified in a DSML by software engineers and/or domain experts. We adopt the DSML SIDOPS+ from the 20-Sim toolset to define physical models. 20-Sim is a widely applied toolset with an extensive set of functions suitable for modeling and simulating physical systems [1, 23]. In the following we call a SIDOPS+ specification a *physical model*, or just *model*. We defined execution semantics for physical models in embedded control software[2] and implemented an interpreter to execute the physical models. Section 2.4 explains in detail how physical models are specified and executed. The DSML interpreter also facilitates runtime verification. This is discussed in Chapter 3.

[2]There are some small differences with the execution semantics of physical models in the 20-Sim simulation tooling. These differences are discussed in Section 2.9.5.

To compose physical models with software modules, we apply the Composition Filters model. To facilitate the application of the Composition Filters model, an event model has been defined on top of the execution semantics of physical models. This event model defines several types of events that occur during the execution of a physical model, and it defines the properties of these events. Specified *composition filters* filter events and execute certain behavior when an event matches. This behavior is for example the dispatch of a certain message to a GPL software module or logging the occurence of certain events. The dispatch functionality of the Composition Filters model provides a transformation mechanism between events and messages.

The benefits of using the Composition Filters model to compose physical models with software modules, compared to for example implementing event handlers in the GPL, are:

- The Composition Filters model provides a language suitable for event filtering and selection.
- The Composition Filters model facilitates interaction on the abstraction level of both languages: domain-specific events in the execution of physical models are translated to messages targeted to GPL software modules[3], and vice versa.
- Composition filters provide loose coupling between physical models and GPL modules: their interaction is specified separately.
- The Composition Filters model enables aspect-oriented quantification over physical models and events within these physical models. This enables the specification of generic interaction, for example logging of certain events in all physical models.
- The Composition Filters model offers a declarative language to specify event filtering and selection. This facilitates static analysis of the behavior of composition filters.

The application of the Composition Filters model to compose physical models with GPL software modules is explained in Section 2.5.

2.4 Specifying and Executing Models of Physical Characteristics

This section explains how physical models are represented in embedded control software using the SIDOPS+ language from the 20-Sim toolset and how physical models are executed in embedded control software.

[3]Here, we assume that the GPL is a language that adopts the concept of message dispatching between software artifacts. Examples of language classes that support message dispatching are procedural languages and object-oriented languages.

2.4.1 Introduction to 20-Sim/SIDOPS+

The 20-Sim toolset is used to model and simulate physical systems. Part of the 20-Sim toolset is the language SIDOPS+. With this language it is possible to mathematically define physical models. The SIDOPS+ language also offers a composition mechanism to compose smaller physical models into larger physical models. Besides the SIDOPS+ language, the 20-Sim toolset also provides a modeling environment to model physical systems with iconic diagrams and bond graphs. For brevity these features are not discussed in this thesis, as they result in equivalent representations [73].

Listing 2.3 shows an example specification in the SIDOPS+ language. This specification contains three types of definition blocks: `constants`, `variables` and `equations`. SIDOPS+ provides more language constructs to model physical processes, but because they are not used in this thesis, they are not explained here.

```
1 constants
2   real pulsesPerRev=24.0;
3 variables
4   integer global pulseCount=0;
5   real global motorRotation {Rotation, rev};
6 equations
7   motorRotation=pulseCount / pulsesPerRev;
```

Listing 2.3: 20-Sim example specification

Constants are defined in the `constants` definition block. The example shows the definition of the constant `pulsesPerRev` of type `real`. SIDOPS+ supports a number of different types, such as `integer`, `real` and `boolean`. Constant definitions always have a value assignment. In the example, the value `24.0` is given to `pulsesPerRev`.

The `variables` block defines the physical variables. The example shows the definition of two physical variables: `pulseCount` and `motorRotation`. Variables have a type. They can also have the modifier `global`, which indicates that the same variable can also be used in other models. In a 20-Sim simulation, this means that if this variable is defined in multiple submodels, composed into a larger model, they all represent the same variable.

The example shows that a variable may have an initial value assigned. This assignment is optional. Furthermore, it is also possible to attach the name and unit of the corresponding physical quantity to the variable, as the example shows for the variable `motorRotation`: within curly braces the quantity name `Rotation` and unit `rev` are given. The definition of the quantity name and unit is optional, but can be used to check whether defined equations are consistent. The quantity name and unit can be attached to constant definitions in the same way.

Equations are defined in the `equations` block. Equations specify mathematical relationships between variables. An equation is composed of two expressions, separated by an equality (`=`) sign. An expression can contain variables, constants, operators and predefined functions. The example shows how the variables `motorRotation` and `pulseCount` relate to each other.

For further detail about the SIDOPS+ language, we refer to the 20-Sim documentation in [73].

2.4.2 Definition: Dependency Graph

A specific representation of a physical model (as specified using SIDOPS+ specifications) that we will use in this thesis is called a *dependency graph*. A dependency graph is a directed graph structure that relates variables and equations in the physical model to each other.

Definition 2.4.1 (Dependency Graph) *A dependency graph is an ordered pair* (vN, eN, E)*, where:*

- vN *is a set of* variable nodes*.*
- eN *is a set of* equation nodes*.*
- $vN \cap eN = \emptyset$
- *The total set of nodes* N *is defined as* $N = vN \cup eN$*.*
- E *is a set of edges, which are ordered pairs of nodes:*$E \subseteq N \times N$*.*

Furthermore, a proper dependency graph fulfills the following constraints:

- *There are no edges between two* equation nodes[4]*:* $E \cap eN \times eN = \emptyset$*.*
- *There are no self-edges:* $E \cap \{(n, n) | n \in N\} = \emptyset$*.*

In addition, the edges attached to equation nodes adhere to the following rules:

- *The equation node has edges to and/or from those and only those variable nodes that correspond to the variables in the equation.*
- *If the left-hand side of an equation contains only one variable, then the edges between the equation node and the variable nodes are directed:*
  - *For all variable nodes that correspond to variables on the right-hand side of the equation, the edge is directed from the variable node to the equation node.*
  - *For the single variable node that corresponds to a variable on the left-hand side of the equation, the edge is directed from the equation node to the variable node.*
- *If the number of variables on the left-hand side of the equation is not equal to* 1*, there is no directional relationship between the equation node and the variable nodes. This is represented in the dependency graph as two edges, one in each direction*[5]*.*

[4]Edges between two *variable nodes* are allowed. For certain applications of dependency graphs this is used to indicate that two connected variables have the same value.

[5]The visual representations of the dependency graph use two-headed single arrows to represent the existence of two edges, one in each direction.

Creating a Dependency Graph for a Physical Model

We will now describe how a dependency graph is created for a given physical model. Suppose we are able to obtain the following meta-information from the physical model specified in SIDOPS+:

- $model$: refers to a given physical model.
- $variables(model)$: results in a set containing all variables in a given physical model ($model$). $variables(model).size$ gives the number of elements in the set.
- $equations(model)$: results in a set containing all equations in a given physical model ($model$). $equations(model).size$ gives the number of elements in the set.
- $lhsVars(equation)$: results in a set containing all variables on the left hand side of a given equation. $lhsVars(equation).size$ gives the number of elements in the set.
- $rhsVars(equation)$: results in a set containing all variables on the right hand side of a given equation. $rhsVars(equation).size$ gives the number of elements in the set.

Procedure `createDependencyGraph` describes how a dependency graph is created, using the meta-information from the physical model. First, for each variable in the physical model a *variable node* is created. A mapping from the variable to the corresponding *variable node* is maintained in $varMap$. Next, for each equation in the physical model an *equation node* is created. Edges are created between a created *equation nodes* and the *variable nodes* corresponding to the variables in the equation. These variable nodes are obtained using the $varMap$ mapping. The *if-statement* selects between the creation of directed edges (when there is a single variable on the left-hand side of the equation) or the creation of undirected edges.

Procedure `createDependencyGraph(`*PhysicalModel model*`)`

1 $vN := eN := E := varMap := \emptyset$
`// Create variable nodes`
2 **foreach** $variable \in variables(model)$ **do**
3     $n :=$ new Node(variable)
4     $vN := vN \cup \{n\}$
5     $varMap := varMap \cup \{(variable, n)\}$
6 **end**
`// Create equation nodes`
7 **foreach** $equation \in equations(model)$ **do**
8     $n :=$ new Node(equation)
9     $eN := eN \cup \{n\}$
10   **if** $lhsVars(equation).size = 1$ **then**
11   **foreach** $variable \in lhsVars(equation)$ **do**
12             $E := E \cup \{(n, varMap(variable))\}$
13   **end**
14   **foreach** $variable \in rhsVars(equation)$ **do**
15             $E := E \cup \{(varMap(variable), n)\}$
16   **end**
17   **end**
18   else
19   **foreach** $variable \in lhsVars(equation) \cup rhsVars(equation)$ **do**
20             $E := E \cup \{(n, varMap(variable)), (varMap(variable), n)\}$
21   **end**
22   **end**
23 **end**
24 **return** (vN, eN, E)

Example 2.2 Dependency Graph

Listing 2.4 shows an example SIDOPS+ specification. The corresponding dependency graph is shown in Figure 2.3 (the numbers in the *equation nodes* correspond to the comments in the listing).

```
1 equations
2   v3 = 0.5 * v1 * v2;           \\ Eq1
3   v6 = 10 * v5 / v4;            \\ Eq2
4   v7 = v3 + 3 * v6;             \\ Eq3
5   v10 = v8 * sqrt(v9) / 3;      \\ Eq4
6   v11 = v7 / v10 / 2;           \\ Eq5
7   v12 = v3 * v6;                \\ Eq6
8   v2 = power(v12, 2);           \\ Eq7
9   v3 * v7 / 3 = v11 + 2 * v13;  \\ Eq8
```

Listing 2.4: Example physical model

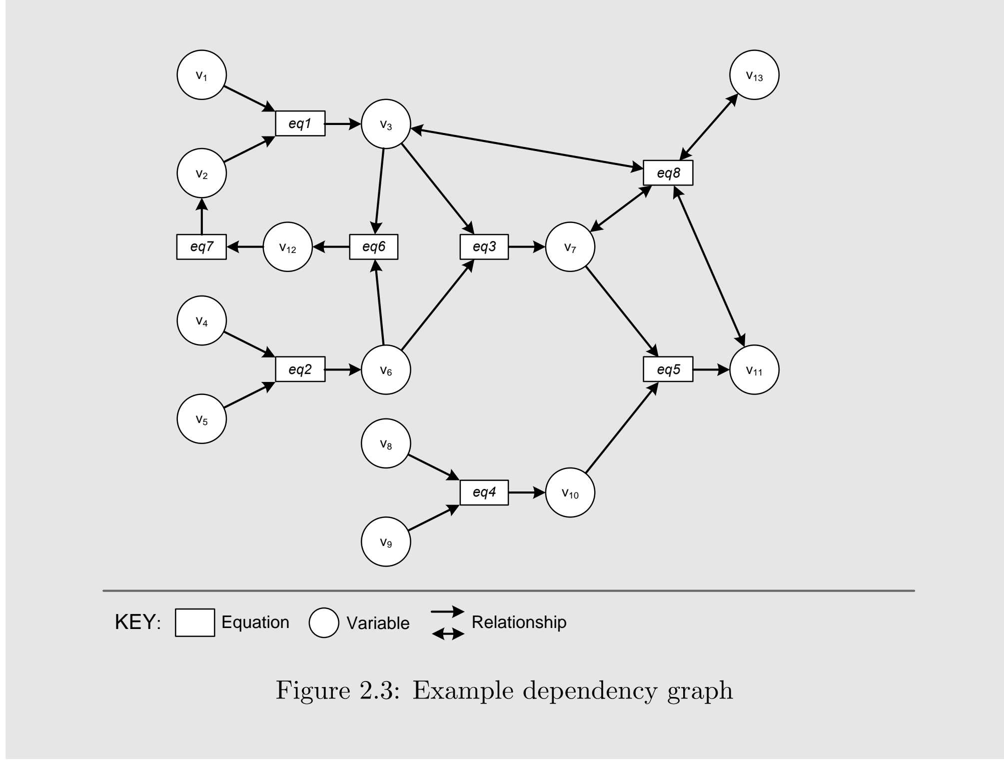


Figure 2.3: Example dependency graph

2.4.3 Instantiation of Physical Models

A physical model defined in SIDOPS+ describes mathematical relationships between physical variables. But, a physical model does not represent the values of the physical variables at a certain moment in time. We call the valuation of the physical variables in a physical model at a certain moment in time a *physical state*. A physical model then defines a set of possible physical states, i.e., all physical states in which the relationships in the model are valid.

To use physical models in embedded control software, the physical state should be represented in software. Furthermore, the consistency of the physical state with the corresponding physical model should be maintained. This means that the mathematical relationships in the physical model are valid for the given values of the physical variables in the physical state. To represent the physical state and maintain its consistency with the physical model, we introduce the concept of a *physical model instance*. A physical model instance maintains a physical state for a given physical model. The physical state maintained by a physical model instance can be manipulated. The runtime tries to maintain consistency of the physical state with the physical model, although this may not always be possible (e.g., because multiple variables may have been updated inconsistently). The approach contains mechanisms to handle inconsistencies, as will be explained in Section 2.5. Figure 2.4 shows schematically the relationship between the physical model and the physical model instance.

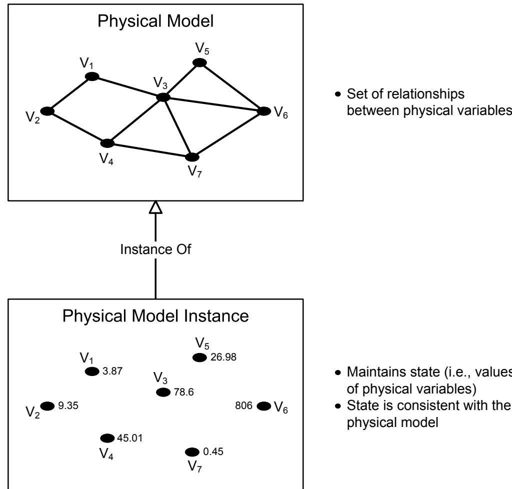


Figure 2.4: Instantiation of a physical model

Use-Cases for Physical Model Instances

Depending on the context in which physical model instances are used, they model different states of the system, for example:

- **Current system state:** The most straightforward application of physical model instances is to model the current system state. Sensor readings are used to update the state in the physical model instance and to maintain its consistency with the real physical machine. For example, in the Drum Shuttling case study this can be applied to determine the current *xPosition* of the drum.

- **Desired system state:** A second application of physical model instances is to model a desired system state. For example, this physical model instance can be used by higher-level controllers to communicate a desired value of a certain physical variable. Using the mathematical relations in the physical model, the desired value of this physical variable is translated into desired values of other physical variables, which might act as setpoints for lower level controllers.

  For example, in the Drum Shuttling case study, this can be applied to determine the desired *stepPosition* of the stepper motor, based on the desired *zPosition* of the drum.

- **A state representing control constraints** This application of physical model instances is a combination of the other two example applications.

The physical state is a mixture of the current system state and desired system state, to determine setpoints for certain controllers. This can be used to enforce constraints on the state of the physical system. These constraints are basically physical models that determine desired values for certain physical variables based on the current values of other physical variables. For example, suppose a physical model contains the relationship $a = 2 * b$. Suppose that the value of b in the physical model instance is the current value in the system and is updated using a sensor. Suppose that the value of a in the physical model instance (which is obtained using the equation) is the setpoint for a controller that controls a. Then, the equation is a constraint on the system that ensures, using the sensor and the controller for a, that a equals $2 * b$.

Such an application of physical model instances can be used in the Warm Process case study, to determine the required value $T_{contact}$ (i.e., $T^{sp}_{contact}$), based on the current values of T_{PH} and v.

2.4.4 Composing Physical Models

Multiple models, resulting from different SIDOPS+ specifications, can be composed into larger models, expressing more complex physical systems. Composition is basically performed by including multiple SIDOPS+ specifications in a physical model. The physical variables with modifier `global` provide the interaction points between the submodels in the composition. This means that if multiple submodels in the model define a variable with the same typing and the same name and they have the modifier global, then this represents a single variable in the composed model. If a submodel defines a variable without the modifier `global`, then in the composed model this variable is different from any variable defined with the same name and typing in another submodel. Example 2.3 illustrates composition of physical models.

Example 2.3 Composition of Physical Models

Listings 2.5, 2.6 and 2.7 show three SIDOPS+ specifications for respectively the physical components *motor*, *gears* and *drum* from the Drum Shuttling case study. These SIDOPS+ specifications will be composed into a physical model of the interaction between the motor, gears and drum.

```
1 constants
2   real pulsesPerRev=24.0;
3 variables
4   integer global pulseCount=0;
5   real global motorRotation {Rotation, rev};
6 equations
7   motorRotation=pulseCount / pulsesPerRev; // eq1
```

Listing 2.5: SIDOPS+ model of motor rotation (`motor`)

```
1 constants
2   real gearTransmission=15.0 / 126.0;
3 variables
4   real global motorRotation {Rotation, rev};
5   real global gearRotation {Rotation, rev};
6 equations
7   gearRotation=gearTransmission * motorRotation; // eq2
```

Listing 2.6: SIDOPS+ model of gear rotation (`gears`)

```
1 constants
2   real drumCircumference=350.0 {Distance, mm};
3 variables
4   real global gearRotation {Rotation, rev};
5   real global drumRotation {Rotation, rev};
6   real global xPosition {Distance, mm};
7 equations
8   drumRotation = gearRotation; // eq3
9   xPosition = drumRotation * drumCircumference; // eq4
```

Listing 2.7: SIDOPS+ model of drum rotation (`drum`)

The listings show that the variable *motorRotation* provides the interaction point between the submodels `motor` and `gears`. Variable *gearRotation* provides the interaction point between the submodels `gears` and `drum`. Figure 2.5 shows these interaction points schematically, using the three dependency graphs that correspond to the three SIDOPS+ specifications.

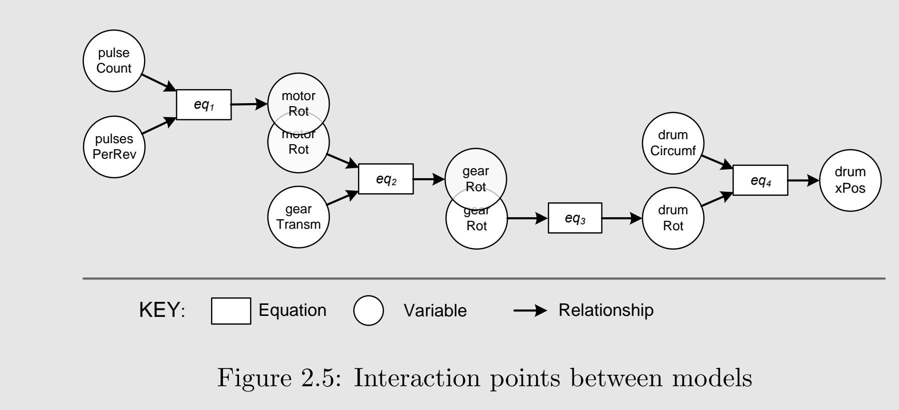


Figure 2.5: Interaction points between models

2.4.5 Executing Physical Models

At runtime, the values in the physical state of a physical model instance can be changed, for example based on sensor readings (Section 2.5 describes the interface to the physical model instance, which enables software modules to change values of physical variables). When the values of certain variables in the physical state change,

the physical model instance ensures that the physical state remains consistent with the corresponding physical model. This means that the physical model instance executes the physical model to determine a new value of each physical variable, using the equations in the physical model and the values of the physical variables that have been changed. This section explains in detail how the physical model instance executes this behavior.

The algorithm that executes a physical model has been seperated into a *main procedure* and several *sub-procedures*. These procedures are described next. All the procedures have access to the following information[6]:

- The derivation graph corresponding to the physical model: $g = (vNodes, eNodes, edges)$.
- The $currentValues : vNodes \rightarrow \mathbb{R}$ mapping, which contains the current physical state as a mapping from the *variable nodes* in the dependency graph to a value.
- The $updatedValues : vNodes \rightarrow \mathcal{P}(\mathbb{R})$ mapping, which contains the updated parts of the physical state as a partial mapping from the *variable nodes* in the dependency graph to a set of values. When values in the physical state are changed using the interface of the physical model instance, these changes are not directly reflected in the $currentValues$ mapping, but first stored in the $updatedValues$ mapping. The reason for this is that the solving algorithm does not necessarily run immediately, but for example at a fixed frequency in a control loop. The result of the mapping is a set of values, because the evaluation algorithm may determine multiple new values for a variable, if there are multiple equations in the physical model from which the value of the variable can be determined.
- The set *updated*, which contains all *variable nodes* that already have been updated. This set is initialized with those *variable nodes* for which the value has been updated using the interface (i.e., for which the mapping $updatedValues$ has been defined).
- The set *solved*, which contains all *equation nodes* that have been solved. Initially, this set is empty.

Main Procedure

Procedure `updateModelInstance()` is the main procedure that updates a physical model instance if some of its variables have been updated.

Updating the physical state is divided into three different techniques:

- **Forward Solving:** Forward solving applies to *equation nodes* that have one outgoing edge and one or more incoming edges (such an *equation node* corresponds to an equation in the SIDOPS+ specification that has one variable on

[6]For brevity this information is not supplied as arguments to the procedures.

Procedure `updateModelInstance`

```
1  change := false
2  repeat
3  |  repeat
4  |  |  repeat
5  |  |  |  change := forwardSolve()
6  |  |  until ¬change
7  |  |  change := backwardSolve()
8  |  until ¬change
9  |  change := noUpdateSolve()
10 until ¬change
```

the left hand side). If the values of the variable nodes attached to all incoming edges are known, the value of the *variable node* on the outgoing edge can be determined by evaluating the equation. Forward solving has the highest preference[7].

- **Backward Solving:** If forward solving (i.e., solving a single variable on the left-hand side using known values for the variables on the right-hand side) cannot be applied anymore, the next option is to use *backward solving*. This means that one unknown variable in an equation is determined by solving the equation using the known values of all other variables.

- **No-update Solving:** If backward solving cannot be applied anymore, the last technique is *No-update solving*. With this technique it is assumed that one of the *variable nodes* that is not updated yet will keep the same value and is considered to be updated. This *variable node* is strategically chosen, so that the other two techniques can be applied again on (some of) the other *variable nodes* that are not updated yet.

Each of these three techniques is explained in detail next. First, Example 2.4 introduces an example that is used to explain the three techniques described next.

Example 2.4 Physical Model Execution

To illustrate the execution of physical models, we use the physical model introduced in Example 2.2. At the start of the execution, the values of three variables (v_1, v_2 and v_7) have been changed. This is illustrated in Figure 2.6. From this start state, the new values for the other variables will be determined using the techniques explained next.

[7] The evaluation order of equations in a physical model can be ambiguous. By giving preference to certain equations based on how they are structured the developer is able to influence the execution order.

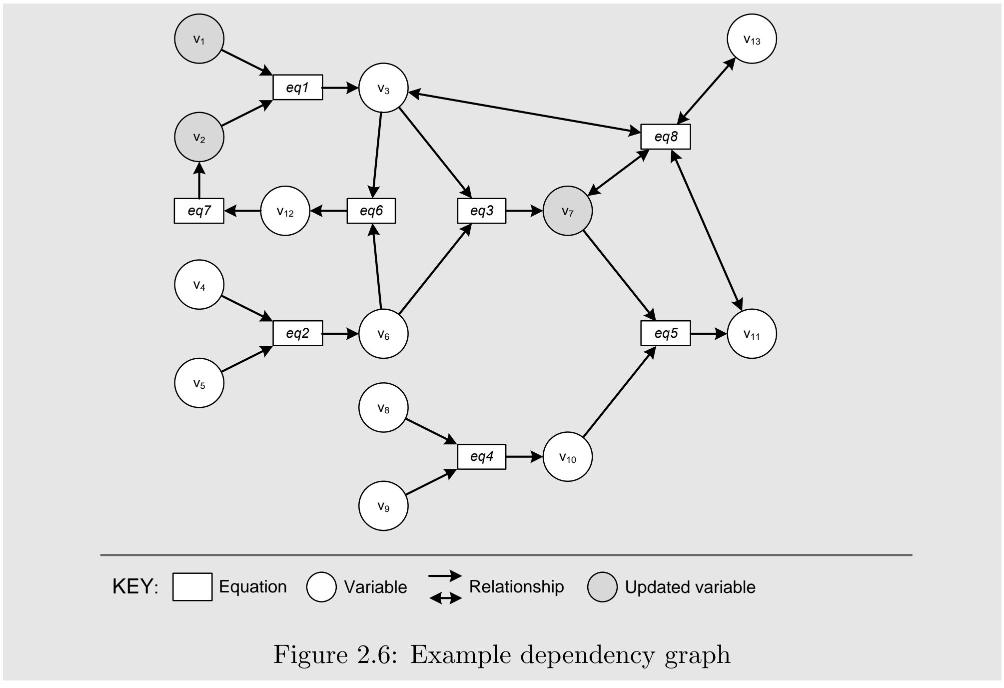


Figure 2.6: Example dependency graph

Forward Solving

Procedure `forwardSolve`

1 $change := false$
2 **foreach** $eNode \in eNodes \setminus solved$ **do**
3   **if** $\forall_{vNode_1 \in vNodes}((vNode_1, eNode) \in edges \rightarrow vNode_1 \in updated) \wedge$ $\exists_{vNode_2 \in vNodes}((eNode, vNode_2) \in edges \wedge (vNode_2, eNode) \notin edges)$ **then**
4   Let $vNode_2$ be the node identified in the existential quantification
5         $process(eNode, vNode_2)$
6         $change := true$
7   **end**
8 **end**
9 **return** $change$

The preferred technique to update the state of a physical model instance is called *forward solving*. This technique is applied on those equations in a SIDOPS+ specification that have one variable on the left-hand side. If the values of the variables on the right-hand side are already updated, the value for the single variable on the left-hand side can be determined.

Procedure `forwardSolve` shows the implementation of forward solving. For each *equation node* that is not already solved, it is checked whether all *variable nodes* attached to incoming edges are updated (i.e., are in the set *updated*) and there is a

variable node attached to the *equation node* with only one edge from the *equation node* to the *variable node* (this is the single variable on the left-hand side of the equation)[8]. In this case, the equation corresponding to the *equation node* can be processed, which means that the equation will be solved to determine the value of the variable on the left-hand side of the equation.

Example 2.4 — Continued

Figure 2.7 shows the updated nodes after forward solving has been applied. Forward solving could only be applied to $eq1$, resulting in the update of v_3.

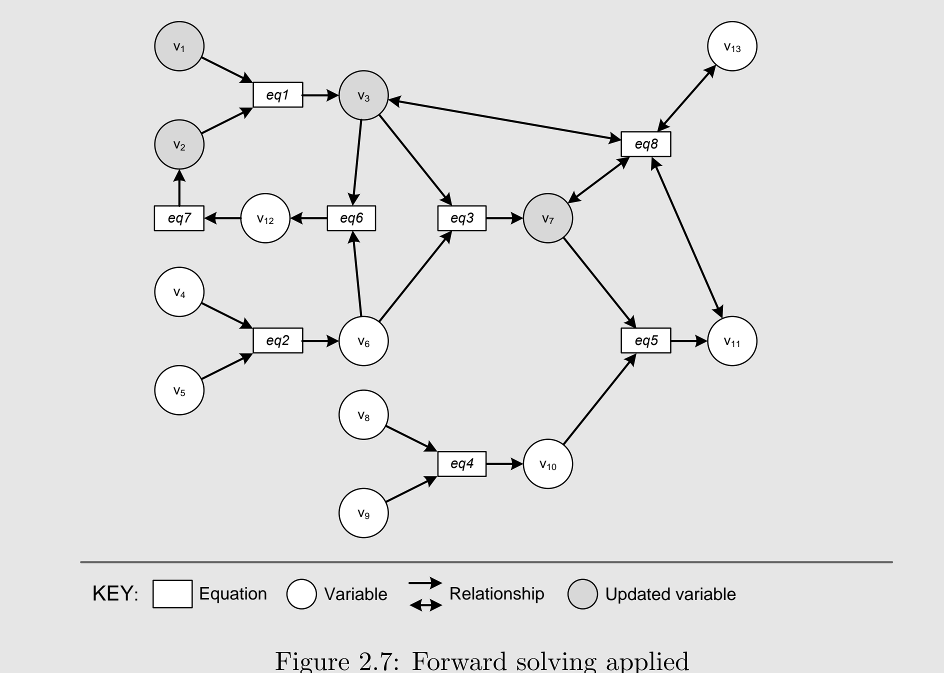


Figure 2.7: Forward solving applied

Backward Solving

If forward solving cannot be applied anymore, the next technique that is used is called *backward solving*. This technique is applied to equations for which all variables except one are updated[9]. The single variable that is not updated can be determined by solving the equation. Procedure `backwardSolve` shows how this is performed.

For each *equation node* that is not already solved, it is checked whether there exists a connected *variable node* that is not updated and it is checked that all the

[8]Note that this variable might already be updated, for example because it is on the left-hand side of another equation. In this case, a second value is determined.

[9]Note that this is less strict than in forward solving, which requires all variables on the right-hand side of the equation to be already updated and which then derives the value of the single variable on the left-hand side of the equation.

Procedure `backwardSolve`

1 $candidates := \emptyset$
2 $eqMapping := \emptyset$
3 **foreach** $eNode \in eNodes \setminus solved$ **do**
4 **if** $\exists_{vNode_1 \in vNodes}(connected(vNode_1, eNode) \wedge vNode_1 \notin updated \wedge \forall_{vNode_2 \in vNodes \setminus \{vNode_1\}}(connected(vNode_2, eNode) \rightarrow vNode_2 \in updated))$ **then**
5 Let $vNode_1$ be the node identified in the existential quantification
6 $candidates := candidates \cup \{vNode_1\}$
7 $eqMapping := eqMapping \cup \{(vNode_1, eNode)\}$
8 **end**
9 **end**
10 $rootNodes := \{n_1 \in candidates | \forall_{n_2 \in candidates} \neg Reachable(n_1, n_2)\}$
11 **foreach** $vNode \in rootNodes$ **do**
12 $process(eqMapping(vNode), vNode)$
13 **end**
14 **return** $rootNodes \neq \emptyset$

other connected *variable nodes* are updated (two nodes are connected iff there is an edge between them). In this case, the *variable node* that is not updated yet is added to the set *candidates* and the relationship between that *variable node* and the *equation node* is added to $eqMapping$.

One may wonder why the *equation nodes* that match the condition are not immediately solved. The reason for this is that the unknown variable in the equation may also be solvable with another equation using forward solving in a later stage. To determine this, it is checked for each *variable node* that end up in *candidates*, whether this node is reachable from any of the other nodes in *candidates*. $Reachable(n_1, n_2)$ is satisfied iff there is a path from n_2 to n_1 in the (directed) dependeny graph. A *variable node* in *candidates* that is reachable from another *variable node* in *candidates* may be solved using forward solving in a later stage, after the other *variable node* has been updated.

The set $rootNodes$ contains all *variable nodes* in *candidates* that are not reachable by any other node in *candidates*. For each *variable node* in $rootNodes$, the attached *equation node* is obtained from the $eqMapping$ and the equation corresponding to this *equation node* is solved to determine the new value for the variable.

Example 2.4 — Continued

Figure 2.8 shows the updated nodes after backward solving has been applied. Backward solving could only be applied to $eq3$, resulting in the update of v_6. Note that the figure informally shows *derivation edges*, indicating the direction of the derivation (different than the dependency direction).

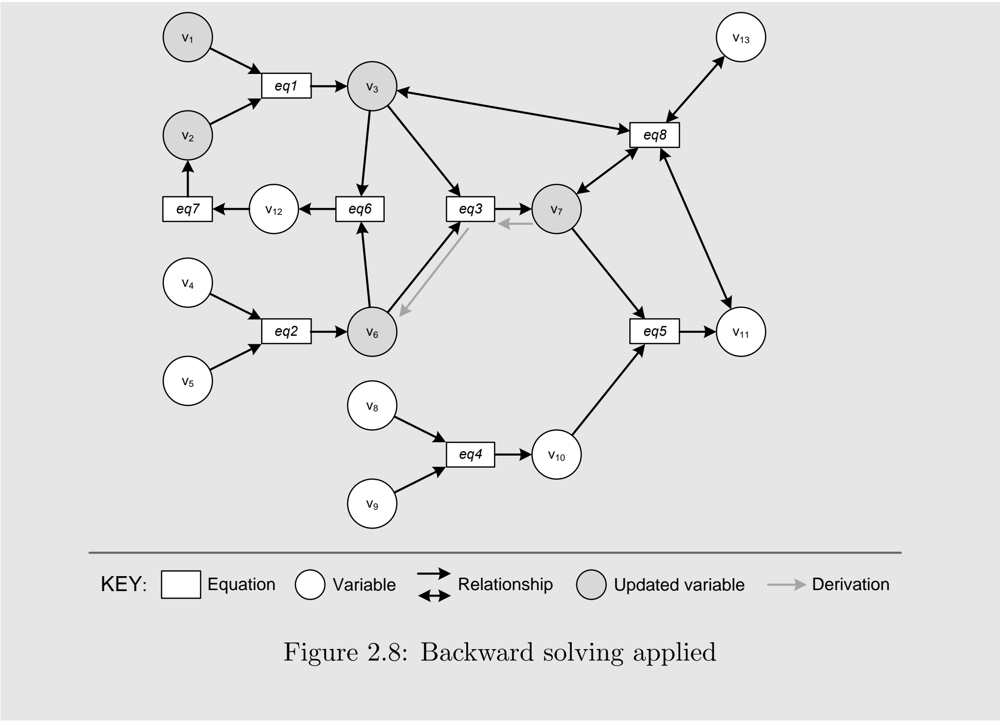


Figure 2.8: Backward solving applied

No-update Solving

If both forward solving and backward solving cannot be applied anymore, the next technique is *no-update* solving. With this technique, it is assumed that the value of one of the *variable nodes* that has not been updated yet, will not change. With this assumption, it is possible to continue the solving procedure. Procedure `noUpdateSolve ()` shows how no-update solving is performed.

Procedure `noUpdateSolve`

1 $candidates := vNodes \setminus updated$
2 $rootNodes := \{n_1 \in candidates | \forall_{n_2 \in candidates} \neg Reachable(n_1, n_2)\}$
`// Randomly select one node from the set rootNodes`
3 Randomly select $vNode \in rootNodes$
4 $updateValue(vNode, currentValues(vNode))$
5 **return** $rootNodes \neq \emptyset$

The set *candidates* contains all *variable nodes* that are not updated yet. The set *rootnodes* is determined in the same way as with backward solving, to eliminate the candidates that can be updated using forward solving.

From the set *rootnodes*, randomly one *variable node* is selected and its value is updated with the current value of the *variable node*. Only one *variable node* is selected

from *rootnodes*, because it is now possible that the other *variable nodes* are solvable using backward solving.

Example 2.4 — Continued

Figure 2.9 shows that after two forward solving steps, solving *eq*5 and *eq*7, both forward solving and backward solving cannot be applied anymore.

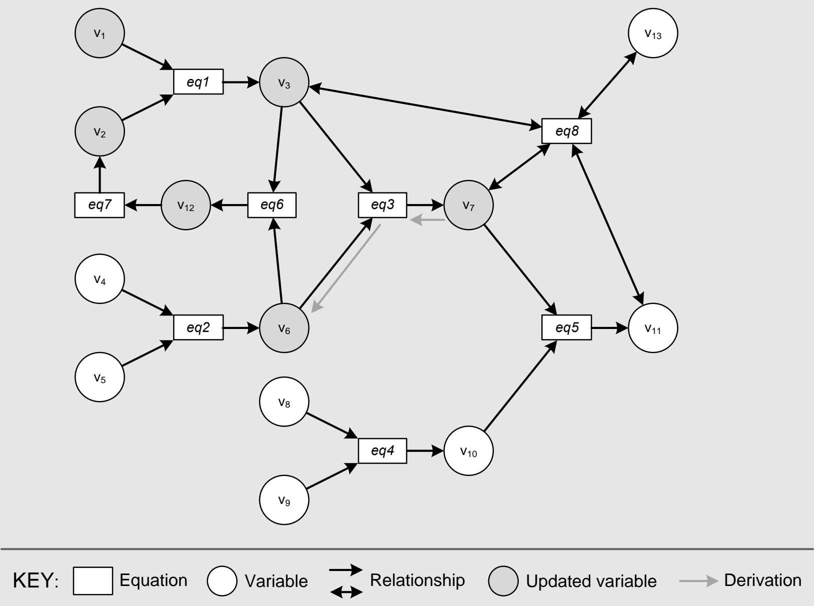


Figure 2.9: Before no-update solving

In this case no-update solving determines a *variable node* that keeps the same value. The set *rootnodes* contains v_4, v_5, v_8 and v_9. Randomly, v_4 is selected as the variable that keeps the same value. This is shown in Figure 2.4.5. After this step, the algorithm continues with backward-solving to update v_5. Next, no-update solving provides values for v_8 and v_9. Then forward solving updates v_{10} and v_{11}. The last step is a backward solving step to update v_{13}

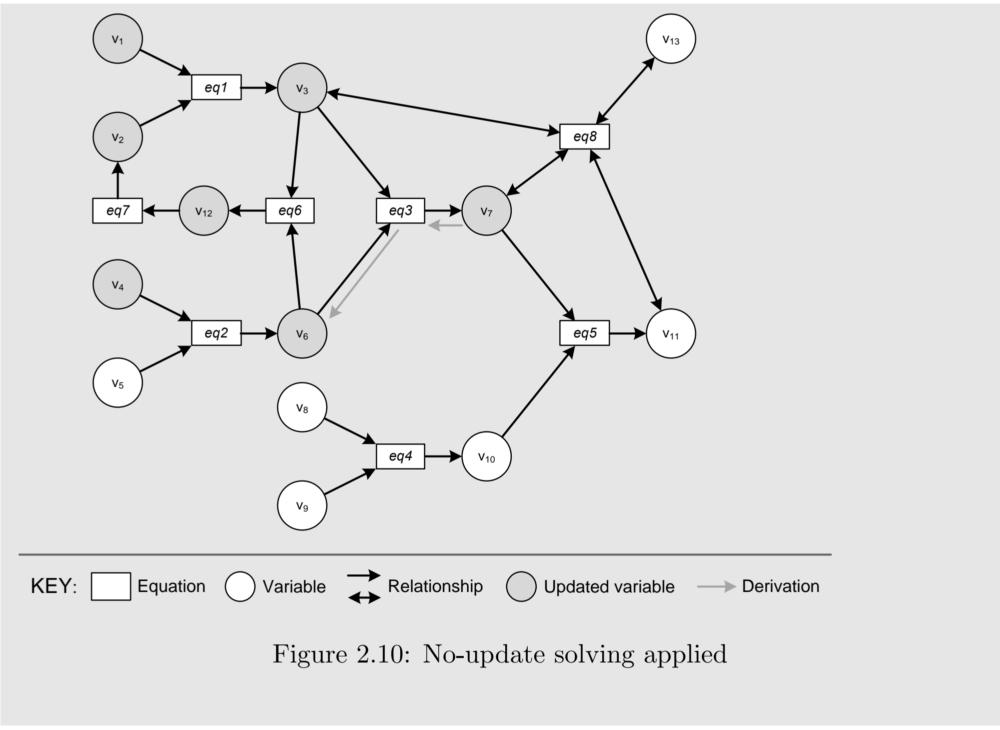


Figure 2.10: No-update solving applied

Procedures `process` and `updateValue`

The previously described procedures use procedures `process` and `updateValue`.

Procedure `process`(*EquationNode eNode, VariableNode vNode*)

1 $value := solve(eNode, vNode)$
2 $updateValue(vNode, value)$
3 $updated := updated \cup \{vNode\}$
4 $solved := solved \cup \{eNode\}$

The procedure `process` processes a given *equation node* to update the value of a given *variable node*, which is connected to the equation node. The procedure `process` calls the procedure `solve`, which performs an equation solving algorithm (an example of such an algorithm is given in Section 2.6). Furthermore, the procedure `process` updates the mapping $updatedValues$ and the sets *updated* and *solved*.

The procedure `updateValue` adds a new value for a given *variable node* to the mapping $updatedValues$. Because it is possible to define multiple equations to derive the value of a specific variable, the mapping $updatedValues$ maps a *variable node* to a set of values. This set contains all values derived using multiple equations. Procedure `updateValue` adds a new value to $updatedValues$ by first checking whether the mapping has already been defined for the given *variable node*. If this is the case,

Procedure `updateValue(` *VariableNode vNode, double newValue* `)`

1 **if** $\exists_{(vNode_1,values)\in updatedValues} vNode_1 = vNode$ **then**
2 $\quad oldValues := updatedValues(vNode)$
3 $\quad updatedValues := (updatedValues \setminus \{(vNode, oldValues)\}) \cup \{(vNode, oldValues \cup \{newValue\})\}$
4 **end**
5 **else**
6 $\quad updatedValues := updatedValues \cup \{(vNode, \{newValue\})\}$
7 **end**

the new value is added to the set of values corresponding to the given *variable node*. Otherwise, a new mapping from the given *variable node* to the new value is added to $updatedValues$.

When to Perform Execution

There are different possibilities as to when a physical model instance should be executed. These possibilities are called *evaluation modes*. Some examples of evaluation modes are:

External trigger
An event external to the physical model instance and evaluator determines when the physical model instance is evaluated. This is for example useful if there is an external clock to which the evaluation should be synchronized.

Scheduled frequency
Evaluation of the physical model instance is scheduled by the evaluator using a predetermined evaluation frequency.

On value request/update
Evaluation is done when a value is requested/updated in the physical model instance.

The type of the application determines which evaluation mode to choose.

2.4.6 Definition: Derivation Graph

When backward solving is applied, the direction in which values of variables are derived from values of other variables is different from the dependency direction in the dependency graph. This so-called *derivation direction* is represented by a *derivation graph*.

Definition 2.4.2 (Derivation Graph) *A derivation graph of a physical model is a directed graph that reflects how the values of the physical variables in the physical*

model are derived from the values of other physical variables after the execution of a physical model instance. A derivation graph is an ordered pair (vN, eN, E)*, where:*

- vN *is a set of* variable nodes.
- eN *is a set of* equation nodes.
- $vN \cap eN = \emptyset$
- *The total set of nodes* N *is defined as* $N = vN \cup eN$.
- E *is a set of edges, which are ordered pairs of nodes:*$E \subseteq N \times N$.

Example 2.5 Derivation Graph

Applying the evaluation algorithm leads to the derivation graph shown in Figure 2.11. This graph shows the order of evaluation for the different physical variables in the physical model. The algorithm determines the value of `v6` using backward solving from equation `eq3` and the values of `v3` and `v7`. This is shown in the derivation graph by the changed direction of the edges `(v7,eq3)` and `(eq3,v6)`, compared to the dependency graph. The value of `v5` is also determined using backward solving, as is shown in the derivation graph by the change of direction of the edges `(v6,eq2)` and `(eq2,v5)`. The bi-directional edges attached to `eq8` in the dependency graph have been replaced in the derivation graph by uni-directional edges that indicate that the value of `v13` is derived from the values of `v3`, `v7` and `v11`.

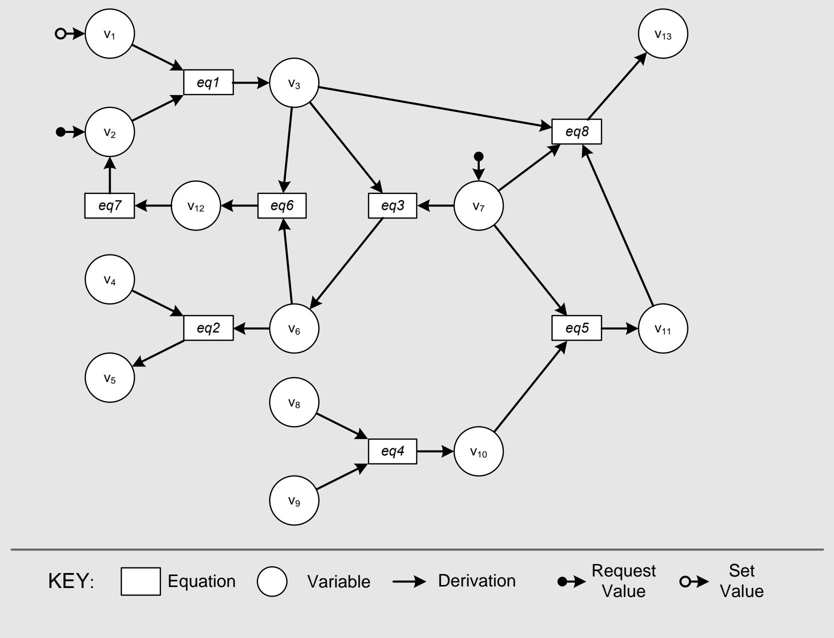


Figure 2.11: Example derivation graph

2.5 Composition using the Composition Filters Model

In the previous section we explained how physical models are represented in software. In this section we describe how physical models interface with other software modules using the *Composition Filters model*.

2.5.1 Background: The Composition Filters Model

In this subsection we give a short introduction in the Composition Filters model. For further information on the Composition Filters model and the Compose* language, we refer to [34, 82, 101, 102].

A key design goal of the *Composition Filters model* is to improve the composability of programs written in object-based programming languages. The Composition Filters model has evolved from the first (published) version of the Sina language in the late 1980s [10, 11], to a version that supports language independent composition of crosscutting concerns [19, 101].

The Composition Filters model can be applied to object-based systems. In such a system, objects can send *messages* between each other, e.g. in the form of method calls or events. In the Composition Filters model, these messages can be filtered using a set of *filters*, as shown in Figure 2.12.

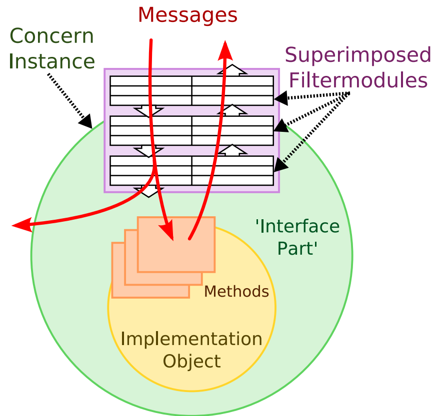


Figure 2.12: Overview of the Composition Filters model

Each *filter* has a *filter type*, which defines the behavior that should be executed if the filter accepts the message and the behavior that should be executed if the filter rejects the message. The matching behavior of a filter is specified by *filter expressions*, which offer a simple declarative language for state and message matching. Filters defining related functionality are grouped in so-called *filter modules*. Such filter modules can also encapsulate some internal state or share state with other objects.

To indicate which filter modules should be applied (*superimposed*) to which objects, we use *superimposition selectors*. A superimposition selector selects a set of

classes using a Prolog-based selector language. A specified filter module is applied to this selected set of classes. The result is that all messages sent to and received by all instances of those selected classes, have to pass through the filters within the filter module.

The Composition Filters model can be applied to many different languages, and we have done so e.g. to SmallTalk [104], Java [108] and C++ [52]. The most recent implementation of the Composition Filters model is the *Compose** toolset, which not only supports .NET, but also Java and C. Example 2.6 shows an example object-oriented application that uses composition filters.

Example 2.6 Pacman with Composition Filters

In this example we apply the Composition Filters model to implement scoring functionality in a Pacman game.

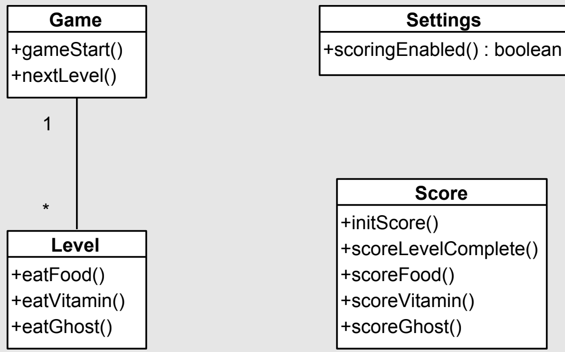


Figure 2.13: Some classes in a Pacman game

Figure 2.13 shows the class diagram of part of a Pacman implementation. The diagram contains the classes `Game` and `Level`, which manage respectively the game and the levels in the game. `Game` contains method `gameStart`, which initializes a new game, and method `nextLevel`, which initializes a new level after a previous level has been completed. The class `Level` contains methods `eatFood`, `eatVitamin` and `eatGhost`, which manage respectively Pacman eating a piece of food, Pacman eating a vitamin and Pacman eating a ghost.

The game includes an option to maintain a score. The class `Settings` contains a flag that indicates whether a score should be maintained. Scores are given for various actions of Pacman: eating a piece of food, eating a vitamin, eating a ghost and finishing a level. Furthermore, scoring should be initialized/reset at the start of a new game. The class `Score` contains a method to initialize scoring (`initScore()`) and methods to add a score when a certain action has happened (`scoreLevelComplete()`, `scoreFood()`, `scoreVitamin()` and `scoreGhost()`). Because of the crosscutting nature of scoring functionality with the classes `Game` and `Level`, composition filters are used to compose `Score` with these classes.

Listing 2.8 shows the composition filters specification that composes the scoring functionality with the Pacman game. The listing shows the definition of the *concern* `ScoringConcern`. This concern consists of one *filter module* definition and one *superimposition* definition.

```
concern ScoringConcern in pacman{
  filtermodule scoring{
    externals
      score: pacman.Score = pacman.Score.instance();
      settings: pacman.Settings = pacman.Settings.instance();
    conditions
      enabled : settings.scoringEnabled();
    inputfilters
      scoreF : After = (enabled & selector=="gameStart")
                 {target=score; selector="initScore"}
        cor (enabled & selector=="eatFood")
                 {target=score; selector="scoreFood"}
        cor (enabled & selector=="eatVitamin")
                 {target=score; selector="scoreVitamin"}
        cor (enabled & selector=="eatGhost")
                 {target=score; selector="scoreGhost"}
        cor (enabled & selector=="nextLevel")
                 {target=score; selector="scoreLevel"};
  }

  superimposition{
    selectors
      classes = { C | isClassWithNameInList(C, ['pacman.Game',
          'pacman.Level']) };
    filtermodules
      classes <- scoring;
  }
}
```

Listing 2.8: Composition filters specification for `Scoring` concern

Filter Module Definition Lines 2 to 19 show the definition of the filtermodule `scoring`. Two external objects, `score` and `settings`, are referenced in the definition of the *externals* on Lines 3 to 5. Line 7 defines a *condition*, which is used in the filter specification. The filter module defines one *filter*, on Lines 9 to 18. The filter consists of several different parts, as indicated below:

$$\overbrace{scoreF}^{identifier} : \overbrace{After}^{filter\ type} = \overbrace{(enabled\ \&\ selector == "gameStart")}^{matching\ part}$$

$$\overbrace{\{target = score; selector = "initScore"\}}^{substitution\ part}$$

$$cor\ \overbrace{(\ldots)\ \{\ldots\}}^{filter\ element}$$

$$\ldots$$

The *identifier* is the name of the filter in the filter module. The *filter type* specifies the type of the filter. In this example, the type is `After`, which means that an additional message is sent after the original message has been further processed. In this way behavior can be added after the original behavior. In the example, this behavior is to perform scoring. Examples of other filter types are `Dispatch`, which performs a dispatch of the message to a given target instead of the original target, and `Logging`, which performs logging of the given message.

Filters contain one or more *filter elements*. The filter `scoreF` contains five filter elements. Filter elements define message matching and substitution. The five filter elements in the example are composed with a *conditional-or* (`cor`) operator, meaning that if a filter element accepts, the filter accepts without evaluating the next filter elements. If a filter element rejects, the next filter element is processed.

A filter elements consists of a *matching part* and a *substition part*. The matching part defines a matching condition on the messages. Only if the matching condition is satisfied, the filter element accepts and the substitution part is executed. The substitution part changes certain properties of the message. When a filter element accepts a message, the filter of which the filter element is part accepts the message, and the behavior corresponding to the filter type is executed.

The example filter on Lines 9 to 18 of Listing 2.8 show five filter elements. Each of these filter elements only match when the condition `enabled` is `true`. Furthermore, each of these filter elements match a different selector (i.e., method call) and specifies a different selector in `score` to which a message is sent after the execution of the original message. For example, when a message with selector `"eatVitamin"` is processed, first `Level.eatVitamin` executes, followed by a call to the method `scoreVitamin` in `Score` (as shown on Lines 13 and 14), to apply the scoring that corresponds to eating a vitamin.

Superimposition Definition Lines 21 to 26 show the *superimposition* definition. A superimposition definition specifies which filter modules are placed (i.e., superimposed) on which artifacts (e.g., classes). The given superimposition definition places the filter module `scoring` on classes `Game` and `Level`.

2.5.2 Composition Overview

To compose physical model instances with software modules, physical model instances provide an interface. This interface is divided into two parts:

- A *base interface* to request and update values of variables in the state of the physical model instance.
- An *event model* that defines *events* that can occur during the execution of a physical model instance. The Composition Filters model can be applied to filter and match certain events and define computational logic to be executed when a specific event occurs.

Figure 2.12 on Page 50 showed how a concern instance that is based on objects in an object-oriented language is separated into an *interface part* and an *implementation*

object. Analogously to that figure, Figure 2.14 shows how a concern instance based on a physical model instance, is separated into an interface part and an implementing physical model instance.

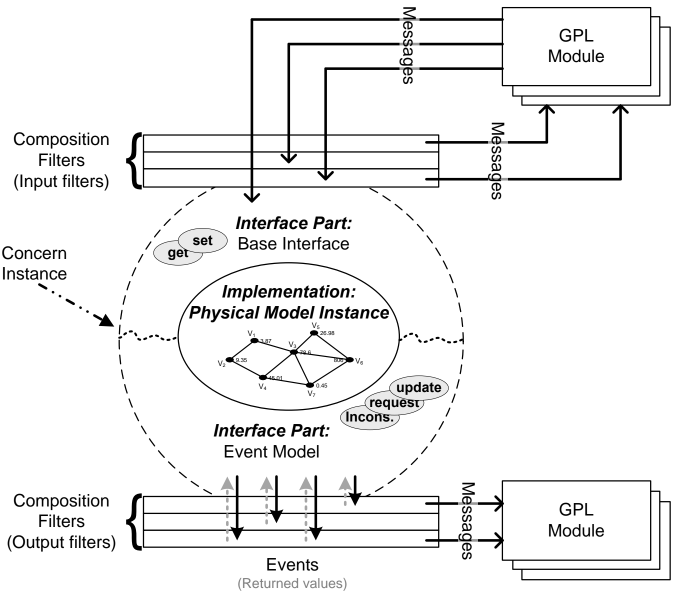


Figure 2.14: Composition filters applied to a physical model instance

The interface part is separated into a base interface and an event model. The base interface is accessible from software modules by sending messages to the physical model instance. The Composition Filters model directly applies to the base interface; filter modules containing input filters can be superimposed on a physical model instance to filter messages that are send to the physical model instance, as shown in the top part of Figure 2.14.

We generalized the Composition Filters model in such a way that not only messages can be filtered and matched, but also events conforming to a certain *event model*. We defined such an event model for the execution of physical model instances. Examples of events that can occur in the execution of physical model instances are a request of the value of a physical variable, the update of the value of a physical variable and the detection of an inconsistency in the physical model. Filter modules, containing output filters, can be superimposed on physical model instances to filter and match events occuring during the execution of the physical model instance. The filters in these filter modules can specify behavior to execute when an event is matched. This behavior can for example be to dispatch a message to a software module to execute certain computational logic. The bottom part of Figure 2.14 shows the *event model* interface part and the application of composition filters.

2.5.3 Base Interface

The physical model instance is in an object-oriented language accessible as an object. This object has an interface to obtain values from the physical model instance and to change values in the physical model instance. Furthermore, the interface contains a method to start the evaluation of the physical model instance. This enables the designer to control when a physical model instance is evaluated (e.g., as part of a control loop). Figure 2.15 shows the base interface of physical model instances.

PhysicalModelInstance
+getValue(variableName : string) : double +updateValue(varName : string, value : double, id : string) : void +evaluate()

Figure 2.15: Base interface of physical model instances

2.5.4 The Event Model

The event model defines the types of events that can happen within the execution of a physical model instance and that can be matched within a composition filter. It is a generic definition for physical model instances, not specific for a particular physical model instance. Events have certain properties. Table 2.1 shows the different properties of events.

Event properties	
$eventType$	
$variableName$	
$value$	
$returnValue$	
$returnIdentifier$	
$values$	Only defined if $eventType =' Inconsistency'$
$margin$	Only defined if $eventType =' Inconsistency'$
$enforceReturn$	Only defined if $eventType =' Inconsistency'$

Table 2.1: Event properties

Property: eventType

The `eventType` property conveys the type of the event. The possible types are:

- *Request*: The value of a variable is requested using the base interface of the physical model instance.
- *Update*: The value of a variable is updated using the base interface of the physical model instance.

- *CheckUpdate*: At the start of evaluation of a physical model instance, for each variable it is checked whether the value has been updated. This eventType indicates this check and provides the possibility to define behavior that updates the value.
- *Change*: The value of a variable has changed during the evaluation of the physical model instance. This type of event occurs after the evaluation algorithm, when the current state is changed to reflect the new state.
- *Inconsistency*: An inconsistency has been detected during the evaluation of the physical model instance. Inconsistencies occur when there are multiple ways to determine the value of a physical variable. If the multiple outcomes are not equal, this leads to an *Inconsistency* event.

Property: variableName

This property contains the name of the variable that is the subject of the event.

Property: value

This property contains a value, which differs based on the type of the event:

- *Request* and *CheckUpdate*: The current value of the variable in the physical model instance.
- *Update*: The updated value of the variable, as provided using the call to the base interface.
- *Change*: The new value of the variable. Actually, this is the current value of the variable in the physical model instance; the current value changed during the evaluation of the physical model instance.
- *Inconsistency*: One of the multiple (inconsistent) values for the variable, which are determined by the evaluation algorithm. The single value is non-deterministically selected.

Property: returnValue

Can be used by filter actions to return a value. The semantics of this returned value differs based on the type of the event:

- *Request*: A `getValue` call has been made to the base interface of the physical model instance, triggering this type of event. Normally, the returned value of a `getValue` call is the current value of the variable. But using the `returnValue` property, a composition filter can change the returned value. The current value of the variable in the physical state is not affected.
- *CheckUpdate*: The property can be used to return an updated value of the corresponding physical variable. This updated value will be used by the evaluation algorithm to update the physical state.

- *Update*: The property can be used to change the updated value provided by the update-call to the base interface that initiated the event.
- *Change*: The property is not used.
- *Inconsistency*: The property contains the value that should be used in the physical state (to resolve the inconsistency).

Property: returnIdentifier

This property is only defined for events with type *CheckUpdate*. It can be used to provide an identifier for the update.

Property: values

In case the event type is *Inconsistency*, this property contains all values for the corresponding variable that are derived by the evaluation algorithm. The property provides a mapping from a `String` identifier to the corresponding value. The identifier indicates the source of the value.

Property: margin

In case the event type is *Inconsistency*, this property contains the largest difference between the values in the property `values`. The `margin` property can be used, for example, to filter *Inconsistency* events for which the difference between the values is larger than a certain threshold value.

Property: enforceReturn

In case the event type is *Inconsistency*, this property contains a Boolean value indicating whether the value in the property `returnValue` should be enforced upon the state or not. Enforcing a value means that the values of other variables are made consistent, according to the physical model, with the provided value in the `returnValue` property. This will be explained in detail in Section 2.5.5.

Example 2.7 Composition Example

Listing 2.9 shows an example composition filters specification. This specification is applied to the physical model instance of the Drum Shuttling case study that contains the drum rotation model specified in Example 2.3 on Page 38.

In this case, the `pulseCount` is retrieved using a sensor. If the `xPosition` is changed, the new value is given to the `Shuttling Controller` module, which implements the control logic to translate the *xPosition* of the drum to the desired *zPosition* of the drum.

```
1  filtermodule RotationIO{
2    externals
3      pcSensor: Sensor = IO.getPulseCountSensor();
4      shControl: ShuttlingController =
           Control.getShuttlingController();
5    outputfilters
6      pcFilter: Dispatch = (event.variableName=='pulseCount' &
           event.eventType=='CheckUpdate')
7        {msg.target=pcSensor; msg.selector='getValue'};
8      xPosFilter: Dispatch = (event.variable=='xPosition' &
           event.eventType=='Change')
9        {msg.target=shControl; msg.selector='setXPos'};
10 }
11
12 superimposition{
13   selectors
14     models = { M | isModelWithNameInList (M, ['motor', 'gears',
           'drum']) };
15   filtermodules
16     models <- RotationIO;
17 }
```

Listing 2.9: Composition filters specification to handle drum rotation

Lines 1 till 10 show the definition of the filter module `RotationIO`. This filter module consists of two references to external objects; `pcSensor` is the external `Sensor` object to provide `pulseCount`. `shControl` is the external `ShuttlingController` object to which a new `xPosition` can be given.

The filter module also defines two composition filters. On Lines 6 and 7 the `pcFilter` has been defined. The type of this filter is `Dispatch`, which indicates that when the filter matches, a message is dispatched to a GPL module. The matching part of the filter shows that the filter matches for *CheckUpdate* events concerning variable *pulseCount*. After the matching part, this filter has an assignment part (shown on Line 7). In this assignment part, the target and selector of the to be dispatched message are provided. The target is the *pcSensor* external object; the selector is the method `getValue`. If the filter matches, a dispatch will be done to this method, and the resulting value will automatically be stored in the `resultValue` property. This value is used by the evaluation algorithm to update the physical state.

Lines 6 and 7 show the definition of the second composition filter, which matches on *Change* events of *xPosition* and dispatches to the method `setXPos` in the *shControl* external object.

Lines 12 until 17 show a superimposition definition. In this superimposition definition, the defined filter module `RotationIO` is placed on the physical model instance of the physical model that contains one or more of the SIDOPS+ specifications `motor`, `gears` and `drum`.

2.5.5 Resolving Inconsistencies

During the evaluation of a physical model instance, the value of a physical variable can be derived in several possible ways:

- An update using the base interface has been performed.
- The event with event type *CheckUpdate* resulted in a new value.
- The value is derived from the values of other physical variables using an equation in the physical model.

If multiple of the above ways to update the value of a physical variable are present, the evaluation algorithm will evaluate all of them, resulting in multiple new values for the physical variable. We call the existence of multiple ways to determine the new value of a physical variable *redundancy* in the physical model instance. Example 2.8 shows an example of a physical model instance containing redundancy.

Example 2.8 Redundancy in Physical Model Instances

```
1  // The SIDOPS+ file is named 'exampleModel'
2  variables
3    real global a;
4    real global b;
5    real global c;
6    real global d;
7  equations
8    b = 2*a;      \\ eq1
9    d = 3*a;      \\ eq2
10   d = c + 5;    \\ eq3
```

Listing 2.10: Example SIDOPS+ specification

Suppose that b and c have been updated using the base interface and that a received an update after the *CheckUpdate* event. The following figure shows the dependency graph of the physical model. The figure also shows the nodes that have status `Updated` before the evaluation algorithm starts. Additional markers are added to the dependency graph to indicate the source of these updates.

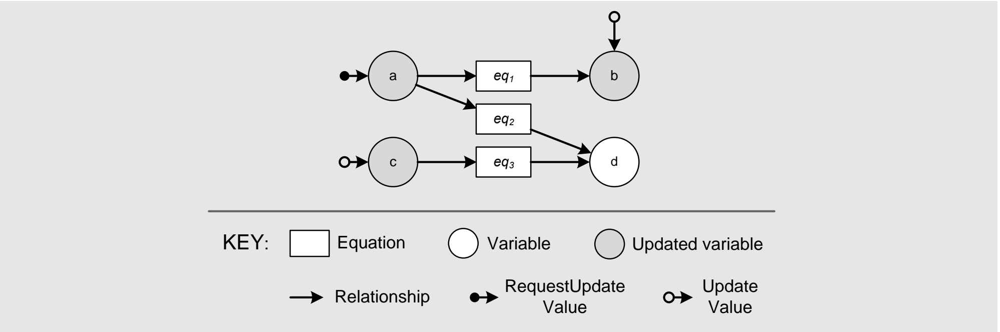


Figure 2.16: Dependency graph showing multiple ways to update certain variables

After the evaluation algorithm has been executed, there are two new values for b: one value that has been set using the base interface and one value that has been derived from a using $eq1$. There are also two new values for d: one value that has been derived from a using $eq2$ and one value that has been derived from c using $eq3$.

Example 2.8 illustrated the possible existence of redundancy in the physical model, leading to multiple newly derived values for a physical variable. However, eventually one value should be selected from these multiple derived values. If all derived values for a physical variable are the same, there is no problem. But, if the values differ, it is not clear which value is the correct one. In this case, an *inconsistency* has been detected, resulting in an *Inconsistency* event.

The `values` property of an *Inconsistency* event provides all derived values as a mapping from an identifier to the corresponding value. The identifiers are determined as follows:

- In case the value originates from an `Update` call to the base interface: the identifier is provided with the `Update` call.
- In case the value originates from the `returnValue` property of a *CheckUpdate* event: the identifier is provided by the `returnIdentifier` property of the event.
- In case the value is derived using an equation: the identifier is the name of the physical model (SIDOPS+ file) that contains the equation. If there are multiple values in the `values` property that are derived using different equations from the same physical model, the name of the physical model is indexed using the ordering of the equations in the SIDOPS+ file.

Composition filters can be defined to match *Inconsistency* events and provide behavior when an event matches. This behavior is for example the selection of the correct value from the provided *values* mapping or error handling. Example 2.9 gives an example of composition filters that handle *Inconsistency* events in the physical model instance from Example 2.8.

Example 2.9 Composition filter to resolve redundancy

```
1  filtermodule InconsistencyHandler{
2    outputfilters
3      filter1: Return = (event.variableName=='b' &
           event.eventType=='Inconsistency')
4        {event.returnValue=event.values['bSensor']};
5      filter1: Return = (event.variableName=='d' &
           event.eventType=='Inconsistency')
6        {event.returnValue=event.values['exampleModel[2]']};
7  }
```

Listing 2.11: Filter module specification that retrieves the sensor value for Tbelt

Listing 2.11 shows the definition of two composition filters that handle inconsistencies. The first filter, specified on Lines 3 and 4, matches *Inconsistency* events for variable b. The filter is of type `Return`, which means that the substitution is executed and the filter returns without additional behavior. The filter shows that the value provided using the base interface with the given identifier $'bSensor'$ is used to resolve the inconsistency.

The second filter, specified on Lines 5 and 6, matches *Inconsistency* events for variable d. The substitution part shows that the value to resolve the inconsistency is the value that is derived using an equation from the physical model $exampleModel$. Actually, the second equation in this physical model is used to derive values for variable d (i.e., the equation $d = c + 5$, as the first one is $d = 3 * a$).

Enforcing Return Value

In case of an *Inconsistency* event, if the property `enforceReturn` is set to *true*, the returned value must be enforced in the state of the physical model instance. This means that the values of other variables should be made consistent, according to the physical model, with the returned value. We will now describe the algorithm to perform this.

The procedures described next all have access to the following information[10]:

- The derivation graph corresponding to the physical model: $g = (vNodes, eNodes, edges)$.
- The $newValues : vNodes \rightarrow \mathbb{R}$ mapping, which contains the new values for the physical state after the evaluation algorithm as a mapping from mapping from the *variable nodes* in the dependency graph to a value.
- The set $enforced$, which contains all *variable nodes* that already have been enforced. This set is initialized with those *variable nodes* for which an *Inconsistency* event returned with the property `enforceResult` set to *true*.
- The set *solved*, which contains all *equation nodes* that have been solved. Initially, this set is empty.

[10]For brevity this information is not supplied as arguments to the procedures.

Procedure `enforceReturn()` is the main procedure to enforce returned values after *Inconsistency* events.

Procedure `enforceReturn`

```
1 change := false
2 repeat
3   repeat
4     change := strictEnforce()
5   until ¬change
6   change := looseEnforce()
7 until ¬change
```

The enforcing algorithm is divided into two different techniques:

- **Strict enforcement:** This is the preferred technique. In this case an *equation node* is solved if there is only one connected *variable node* that is not yet in the set $enforced$. The value of this *variable node* can be derived using the equation and the values of the other connected *variable nodes* (which are already in the set $enforced$).
- **Loose enforcement:** If *strict enforcement* cannot be applied anymore, *loose enforcement* is applied. This technique propagates enforced values by solving equations nodes for which at least one incoming edge has an attached *variable node* that is in the set $enforced$ and the *variable node* attached to the single outgoing edge is not in $enforced$. The value for the *variable node* attached to the outgoing edge is determined by solving the equation. This technique performs only forward propagation of enforced values.

Each of these two techniques is explained in detail next.

Procedure `strictEnforce`

```
1 change := false
2 foreach eNode ∈ eNodes \ solved do
3   if ∃_{vNode1∈vNodes}(connected(vNode1, eNode) ∧ vNode1 ∉ enforced ∧
    ∀_{vNode2∈vNodes\{vNode1}}(connected(vNode2, eNode) → vNode2 ∈ solved))
    then
4     Let vNode1 be the node identified in the existential quantification
5     processEnforce(eNode, vNode1)
6     change := true
7   end
8 end
9 return change
```

Procedure `strictEnforce` shows the implementation of the *strict enforcement* technique. The *equation nodes* that are not solved yet are checked whether all connected

variable nodes except one is in the set *enforced*. If this is the case, the value for the connected *variable node* that is not in the set *enforced* is determined by processing the equation.

Procedure `looseEnforce`

1 $candidates := \emptyset$
2 $eqMapping := \emptyset$
3 **foreach** $eNode \in eNodes \setminus solved$ **do**
4 | **if** $\exists_{vNode_1 \in vNodes}((vNode_1, eNode) \in edges \wedge vNode_1 \in enforced) \wedge \exists_{vNode_2 \in vNodes}((eNode, vNode_2) \in edges \wedge (vNode_2, eNode) \notin edges \wedge vNode_2 \notin enforced)$ **then**
5 | | Let $vNode_2$ be the node identified in the second existential quantification
6 | | $candidates := candidates \cup \{vNode_2\}$
7 | | $eqMapping := eqMapping \cup \{(vNode_2, eNode)\}$
8 | **end**
9 **end**
10 $rootNodes := \{n_1 \in candidates | \forall_{n_2 \in candidates} \neg Reachable(n_1, n_2)\}$
11 **foreach** $vNode \in rootNodes$ **do**
12 | $processEnforce(eqMapping(vNode), vNode)$
13 **end**
14 **return** $rootNodes \neq \emptyset$

Procedure `looseEnforce` shows the *loose enforcement* technique. This technique first selects the candidate *equation nodes* to be solved[11]. The candidate *equation nodes* are those *equation nodes* that have at least one incoming edge for which the connected *variable node* is in *enforced* and the *variable node* connected to the single outgoing edge of the *equation node* is not in *enforced*.

Similar to the procedure `backwardSolve`, which was explained on Page 44 as part of the evaluation algorithm, from the set *candidates* those nodes are selected that are not reachable from other nodes in *candidates*. This is because nodes that are reachable from other nodes in *candidates* may be solved in a later stage using the *strict enforcement* technique.

Procedure `processEnforce(`*EquationNode eNode, VariableNode vNode*`)`

1 $value := solve(eNode, vNode)$
2 $newValues.put(vNode, value)$
3 $enforced := enforced \cup \{vNode\}$
4 $solved := solved \cup \{eNode\}$

The procedure `processEnforce` processes a given *equation node* to enforce the value of a given *variable node*, which is connected to the *equation node*. The procedure

[11] Actually, the set *candidates* contains the *variable node* that can be solved. The corresponding *equation node* can be derived using the mapping *eqMapping*.

`processEnforce` calls the procedure `solve`, which performs an equation solving algorithm (an example of such an algorithm is given in Section 2.6). Furthermore, the procedure `processEnforce` updates the mapping *newValues* and the sets *enforced* and *solved*.

2.5.6 Default Composition Filter

When an event occurs in the execution of the physical model instance, the value of the physical variable that is the subject of the event may be influenced. For example, if the event type is *Update*, the update value provided to the base interface with the `updateValue` call can be changed by a composition filter. If the event type is *Inconsistency*, the proper value should be selected.

Using the `resultValue` property, a composition filter can give a new value for the physical variable. But if no composition filter has been defined to provide a result value, there is a default composition filter that performs this task. The default composition filter is always superimposed to a physical model instance. The behavior of the default composition filter is to return the current value in the physical state as the new value. Listing 2.12 shows the implementation of the default composition filter.

```
1 filtermodule defaultModule{
2   outputfilters
3     defaultFilter: Result = (True) {event.resultValue=event.value};
4 }
```

Listing 2.12: Default composition filter

The default composition filter consists of a matching part that always matches (all events are matched) and a substitution part that gives the property `resultValue` the value of the `value` property. For *Update* events, this is the value provided by the `updateValue` call to the base interface. For *Inconsistency* events, this is a random value from the set containing multiple inconsistent values for the variable.

2.6 Design

In this section we describe design details of the DSML interpreter and the connection with the Compose* toolset to execute the composition filters.

2.6.1 Structure of the DSML Interpreter

Figure 2.17 shows a class diagram of the DSML interpreter. The class diagram contains the following classes:

SIDOPSModel
: Instances of this class represent a SIDOPS+ model that has been parsed from a SIDOPS+ file.

PhysicalModel
: Instances of this class represent a physical model. A `PhysicalModel` has a reference to one or more `SIDOPSModels` (i.e., model composition). A `PhysicalModel`

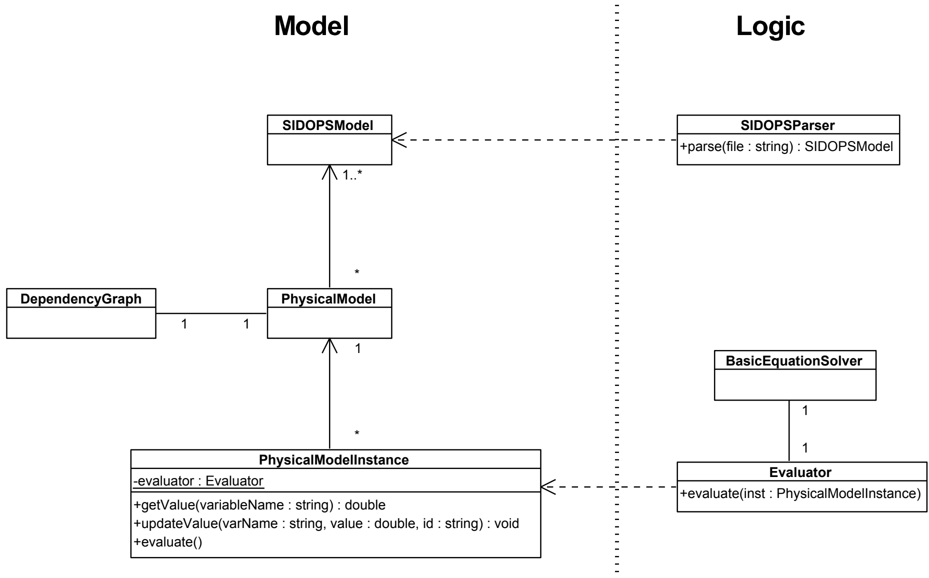


Figure 2.17: Class diagram of the DSML interpreter

also has a reference to a `DependencyGraph`, which is the dependency graph that corresponds to the physical model.

DependencyGraph
Instances of this class represent a dependency graph.

PhysicalModelInstance
Instances of this class represent a physical model instance.

SIDOPSParser
Parser of SIDOPS+ files.

Evaluator
Implements the evaluation algorithm presented in Section 2.4.5 and the algorithm to resolve inconsistencies presented in Section 2.5.5.

BasicEquationSolver
An implementation of an equation solver that can solve a limited set of basic equations. This equation solver is used for experimentation and as a proof of concept. Powerful equation solvers are (commercially) available. More detail about the basic equation solver is given in Section 2.6.3.

2.6.2 Connection with the Compose* Runtime

Figure 2.18 shows a class diagram representing the connection between the DSML interpreter and the Compose* runtime. This class diagram contains the following

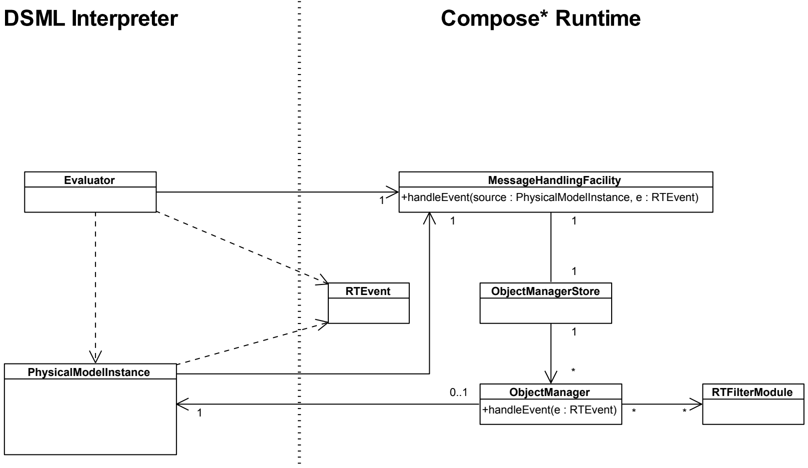


Figure 2.18: Class diagram showing the connection with Compose*

classes[12]:

MessageHandlingFacility
Component in the Compose* runtime that handles messages. We extended this class to be able to also handle events occuring in the execution of a physical model instance. This class contains the method `handleEvent` to handle events. This method is called from the `Evaluator` and `PhysicalModelInstance` classes in the DSML Interpreter when events are occuring.

ObjectManager
The Compose* runtime creates an object manager for each object that has composition filters superimposed. The object manager registers the superimposed filter modules and implements handling of messages for the object. We extended the Compose* runtime so that an object manager is also created for each physical model instance. We extended `ObjectManager` so that object managers can handle events occuring in the corresponding physical model instance.

ObjectManagerStore
The object manager store contains all object managers.

RTFilterModule
The runtime representation of a filter module.

RTEvent
The runtime representation of an event.

Evaluator
See also Section 2.6.1. This class evaluates physical model instances. It calls the message handling facility in the Compose* runtime when an event is occuring in the execution of a physical model instance.

[12]Only limited information about the classes in the Compose* runtime is given here. For more information about the Compose* runtime we refer to [101].

PhysicalModelInstance

See also Section 2.6.1. This class implements physical model instances. It calls the message handling facility in case an event that is related to access of the base interface of the physical model instance occurs (i.e., events with event types *Request* and *Update*).

The classes `Evaluator` and `PhysicalModelInstance` communicate the occurence of an event to the class `MessageHandlingFacility`. This happens in the following cases:

- **Request:** When the `getValue` method in `PhysicalModelInstance` is called, the physical model instance communicates the occurence of a *Request* event to the message handling facility.
- **Update:** When the `update` method in `PhysicalModelInstance` is called, the physical model instance communicates the occurence of an *Update* event to the message handling facility.
- **CheckUpdate:** When the execution of `updateModelInstance()` (see Section 2.4.5) is started, it is checked for each variable whether the value has been updated. The `Evaluator` communicates the occurence of *CheckUpdate* events to the message handling facility.
- **Change:** When the execution of procedures `updateModelInstance()` (see Section 2.4.5) and `enforceReturn()` (see Section 2.5.5) has ended, the set $currentValues$ in the physical model instance is updated with the new values. For each variable that has changed (i.e., the new value is different from the current value), the evaluator communicates the occurence of a *Change* event to the message handling facility.
- **Inconsistency:** If there are variables with inconsistent values after the execution of the procedure `updateModelInstance()` (see Section 2.4.5), for each of these variables the evaluator communicates the occurence of an *Inconsistency* event to the message handling facility.

2.6.3 Basic Equation Solver

We implemented a basic solver to solve the equations in the physical model. It cannot solve all possible physical models in the SIDOPS+ language. It is limited to solving equations in which each physical variable occurs only once. This limitation excludes for example differential equations. This is sufficient for demonstration on the example cases. More powerful solvers, such as those used in the 20-Sim toolset or the Matlab Simulink toolset, can be implemented in the future.

The solver uses the *abstract syntax tree* (AST) of the corresponding equation. Figure 2.19 shows an example AST for the following equation[13]:

$$T_{contact} = c_4 \frac{P_{rad}}{\sqrt{v}} + T_{belt}$$

[13] Note that this is Equation 1.2 from the Warm Process case study.

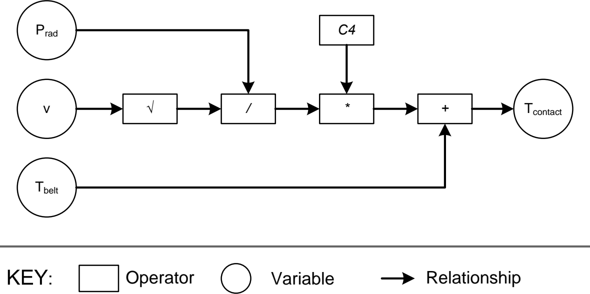


Figure 2.19: The abstract syntax tree of Equation 1.2

The used AST representation contains *operator nodes* to represent operators and *variable nodes* to represent variables.

Operator Nodes For each operator in the supported language, it is defined how the value of an unknown edge of the corresponding operator node can be determined if the values of all other edges of the node are known. Figure 2.20 shows this for three operators. Each operator in the figure has a number of inputs $I_1 \dots I_n$ and one output/result R. The formulas at each input/output edge indicate how the value on this edge can be determined from the values on the other edges. For example, for the division operator ('/'), the value of R can be determined by dividing I_1 by I_2, the value of I_1 by multiplying R and I_2, the value of I_2 by dividing I_1 by R.

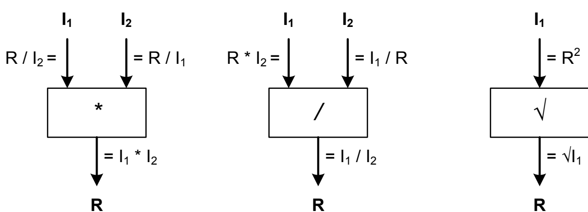


Figure 2.20: Operator definitions

The algorithm to solve the value of a variable now informally works as follows:

1. For each variable node of which the value is known, provide this value to all edges of the variable node.

2. For each operator node: If the value of one attached edge is unknown and the values of all other edges are known, then the value on the unknown edge can be determined from the values of the other edges, through the operator definition.

3. For the variable node for which the value is not known, if there is an attached edge that has a known value, the value of the variable node is set to this value. The algorithm ends, as the value has been resolved.

4. Repeat steps 2 and 3 until the algorithm ends.

After application of this algorithm, the value of the unknown variable has been resolved.

Example 2.10 Example Equation Solving

Figure 2.21 gives an example of a solving problem, using the AST of Equation 1.2. In this example, the values of $T_{contact}$, T_{belt} and P_{rad} are known, and the value of v is unknown. For this example, the constant c_4 has the value 2.

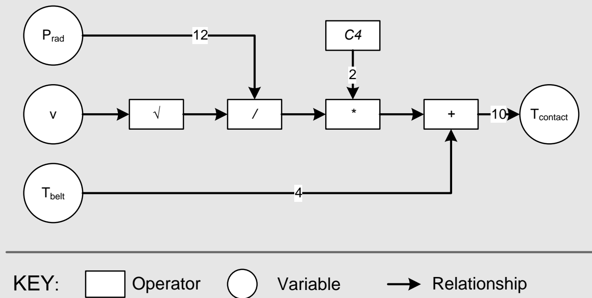


Figure 2.21: Example of an equation solving problem

Applying the solving algorithm gives the result shown in Figure 2.22. The value of v is determined to be 16.

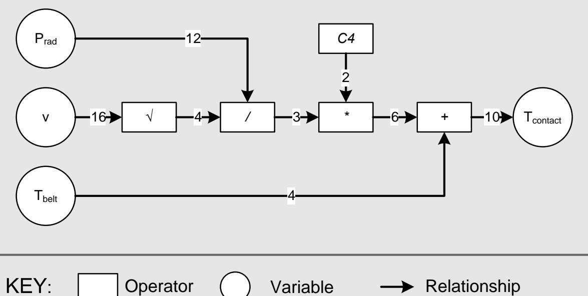


Figure 2.22: Solution after applying the basic equation solving algorithm

2.7 Example Control Software Designs

Example 2.11 Control Software Design of the Warm Process Case Study

Figure 2.23 shows the software structure and data-flow between the artifacts for the Warm Process case study. The design includes two physical model instances. The instance of the `PrintQuality` physical model ensures correct print quality, by providing a setpoint for $T_{contact}$ to the `RadiatorController` software module. The SIDOPS+ specification of the `PrintQuality` physical model is given in Listing 2.13. The instance of the `BeltTemperature` physical model provides the current $T_{contact}$ to the `RadiatorController` software module. The SIDOPS+ specification of the `BeltTemperature` physical model is given in Listing 2.14.

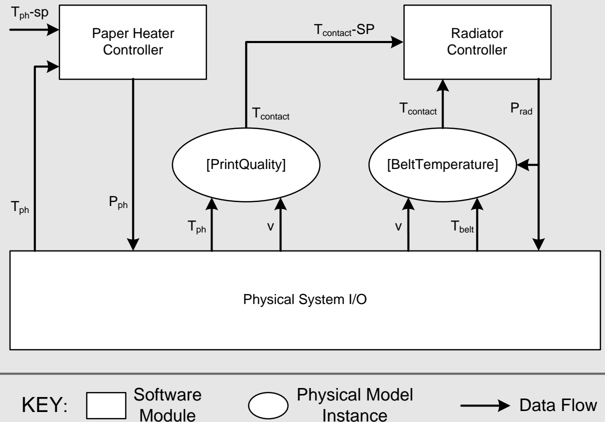


Figure 2.23: Warm Process software structure and data flow

```
1 constants
2   /* definition of c1, c2 and c3 */
3 variables
4   real global v {Velocity, m/s2};
5   real global Tph {Temperature, K};
6   real global TcontactSP {Temperature, K};
7 equations
8   Tcontact = c1*v - c2*Tph + c3;
```

Listing 2.13: PrintQuality SIDOPS+ specification

Listing 2.15 shows part of a composition filters specification that is superimposed on the `PrintQuality` physical model instance. The specifications shows two filters. The first filter (`ioFilter`) gets the sensor readings for T_{ph} and v from `PhysicalSystemIO`. The second filter (`spFilter`) changes the setpoint in the `RadiatorController` when the corresponding value has changed in the physical model instance. For the `BeltTemperature` physical model instance similar filters can be defined.

Figure 2.24 shows how the different artifacts relate, using the overview figure of the approach, which was shown in Figure 2.1 (with an extension shown in Figure 2.2).

The figure shows which artifacts are the GPL software modules, which artifacts are the composition filters and which artifacts are the DSML models. The figure also shows the specific events generated by the DSML models and filtered by the composition filters. Furthermore, the figure shows which messages are dispatched from the composition filters to the GPL software modules.

```
1  constants
2    real c4=/* some value */;
3  variables
4    real global Prad {Power, W};
5    real global v {Velocity, m/s2};
6    real global Tbelt {Temperature, K};
7    real global Tcontact {Temperature, K};
8  equations
9    Tcontact = c4 * Prad/sqrt(v) + Tbelt;
```

Listing 2.14: BeltTemperature SIDOPS+ specification

```
1  ...
2  outputfilters
3    ioFilter: Dispatch = (event.variableName=='Tph' &
         event.eventType=='CheckUpdate')
4      {msg.target=io; msg.selector='getTph'}
5      cor (event.variableName=='v' & event.eventType=='CheckUpdate')
6      {msg.target=io; msg.selector='getV'};
7    spFilter: Dispatch = (event.variable=='Tcontact' &
         event.eventType='Change')
8      {msg.target=radControl; msg.selector='setTcontactSP'};
9  ...
```

Listing 2.15: Composition filters for the PrintQuality physical model instance

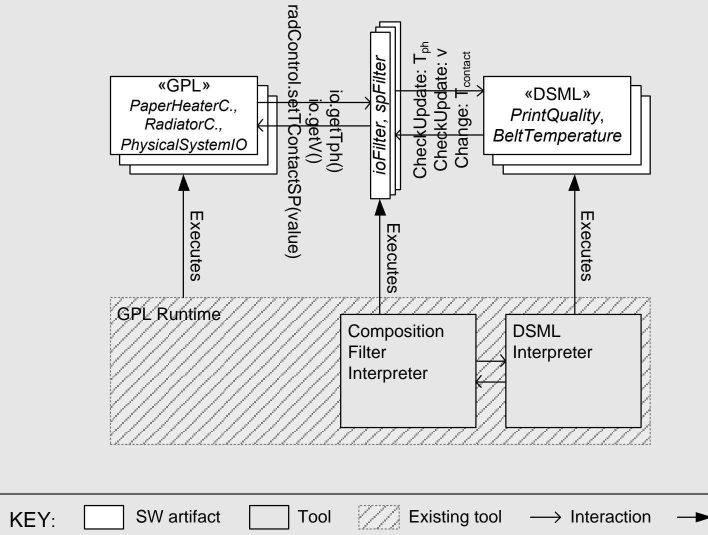


Figure 2.24: Different artifacts shown in the overview figure of the approach

Example 2.12 Control Software Design of the Drum Shuttling Case Study

Figure 2.25 shows the software structure and data-flow between the artifacts for the Drum Shuttling case study. This design also utilizes two physical model instance. One physical model instance is an instance of a physical model that is a composition of the `motor`, `gears` and `drum` SIDOPS+ specifications. This instance provides the current *xPos* to the `ShuttlingController` software module. The second physical model instance is an instance of a physical model that is a composition of the `stepperMotor` and `cam` SIDOPS+ specifications. This instance is used to model a desirable system state. The `ShuttlingController` module provides a desired *zPos* to this model. The model translates this to a desired *stepPos*, and provides this value as a setpoint to the `StepperController` software module.

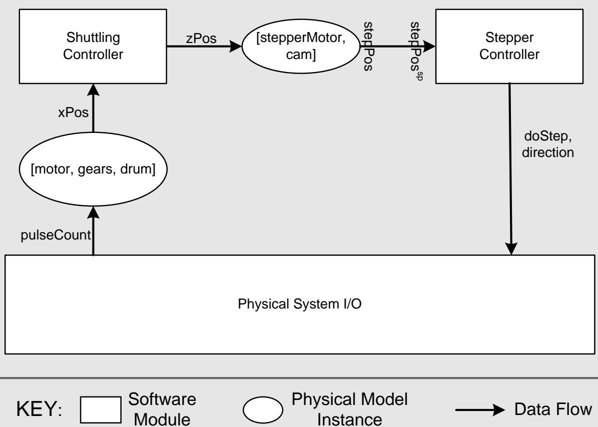


Figure 2.25: Drum Shuttling software structure and data flow

The SIDOPS+ specifications of the physical models `motor`, `gears` and `drum` have been presented in Example 2.3. Listings 2.16 and 2.17 show the SIDOPS+ specifications for the physical models `stepperMotor` and `cam`.

```
1  constants
2    real degreesPerStep=/* some value */;
3  variables
4    integer global stepPosition=0;
5    real global stepRotation {Rotation, deg};
6  equations
7    stepRotation=stepPosition * degreesPerStep;
```

Listing 2.16: stepperMotor SIDOPS+ specification

Example 2.7 on Page 58 already gave the composition filter specification for the `motor,gears,drum` physical model instance. A similar composition filter specification can be defined for the `stepperMotor,cam` physical model instance.

```
1 constants
2   real movementPerDegree=/* some value */;
3 variables
4   real global stepRotation {Rotation, rev};
5   real global zPosition {Distance, mm};
6 equations
7   zPosition=stepRotation * movementPerDegree;
```

Listing 2.17: cam SIDOPS+ specification

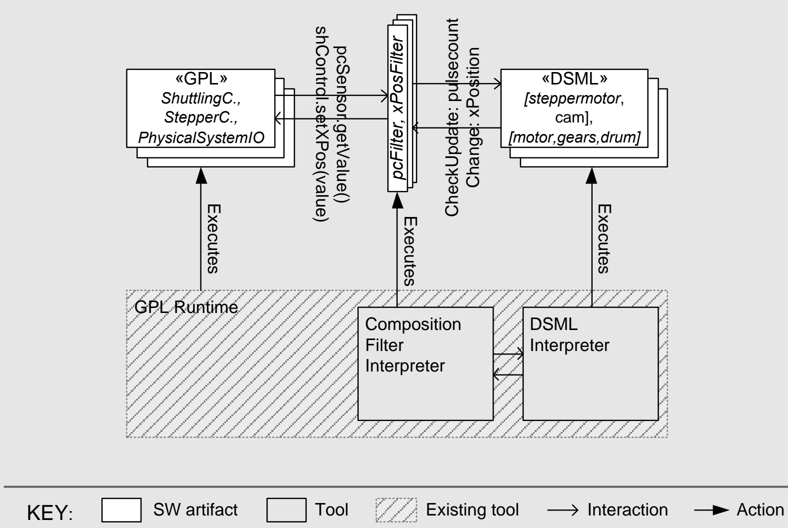


Figure 2.26: Different artifacts shown in the overview figure of the approach

Just as in Example 2.11, Figure 2.26 shows how the different artifacts relate using the overview figure of the approach.

2.8 Evaluation

Our approach enables the composition of physical models specified in a DSML with other software modules implemented in a GPL. This provides the benefits of using a DSML to specify physical models with the freedom of a GPL to implement other functionality. In this section we evaluate the benefits of our approach using a number of evolution scenarios.

2.8.1 Implementing Physical Models with a DSML

Benefits of a DSML

With a DSML it is possible to specify physical models using domain-specific abstractions. This reduces or elliminates accidental complexity that is introduced when domain-specific abstractions are translated to implementation abstractions in a GPL. Less accidental complexity results in an improvement of software quality characteristics such as comprehensibility, maintainability and reusability [24].

Separation of Physical Models

By applying our approach, we separated the physical characteristics from the GPL modules that implement control logic. This increases the cohesion of these GPL modules and lowers their coupling. Increasing cohesion and lowering coupling has a proven benefit on development effort. According to Darcy et al. the combination of improved cohesion and reduced coupling in general leads to reduced development effort [29].

In current practice, system engineers already specify physical models using 20-Sim/SIDOPS+, for simulation and analysis purposes. Our approach provides the possibility to integrate these 20-Sim/SIDOPS+ specifications with software, rather than re-implement the equations from these specifications in software. This provides a further reduction in development effort.

Evolvability of Physical Models

During its lifecycle, an embedded system product type will be subject to evolution. Some types of evolution have an impact on the physical characteristics of the system. Some examples of these evolution scenarios are:

- **Addition or removal of a component in the physical system:** The system consists of multiple components. It is likely that during the lifecycle of a product type a specific component is added or removed from the design, for example to extend its functionality or to lower costs.

- **Change of the physical structure of the system:** The physical structure is subject to change during the lifecycle of the product type. With physical structure is meant the location and arrangement of the different components of the system.

- **Change of operating conditions of the system:** During the design of the system, certain operating conditions, like temperature of the environment, are assumed to be (approximately) between certain fixed margins. However, during the lifecycle, the assumed operating conditions may change. For example, a digital document printing system is first targeted at a market in a moderate climate. Later on, the product becomes targeted at markets in a hot climate, which means a higher environmental temperature.

- **Addition or removal of sensors in the physical system:** During the lifecycle of a product type, new sensors can be added to the system to increase the amount of information available to the control software. Or sensors can be removed from the system, for example because the information that these sensors provide can be derived from other sensors.

These evolution scenarios have an impact on the physical characteristics of an embedded system, and as such these evolution scenarios can have an impact on the implemented models of physical characteristics in embedded control software. With these evolution scenarios, the advantages of using a DSML to explicitly specify physical models in embedded control software become clear. The physical characteristics affected by an evolution scenario are easier to locate in embedded control software when they are separated in a domain-specific model instead of when they are tangled with software modules that also implement other computation logic (e.g., control logic). Furthermore, their domain-specific representation makes them easier to comprehend and modify.

2.8.2 Application of Composition Filters

To compose physical model instances with software modules, we applied the Composition Filters model. This resulted in a number of benefits which we will now discuss.

Separation of Interaction

Composition filters provide the possibility to separate the interaction between physical model instances and software modules from the implementation of the physical model instances and software modules. This provides so-called loose coupling between physical model instances and software modules [48]. Loose coupling improves software quality attributes, such as reusability and comprehensibility.

Interaction at the Abstraction Level of Both Languages

We extended the Composition Filters model to allow filtering and matching of events. We defined a domain-specific event model that specifies events of interest in the manipulation and execution of physical model instances. This event model can be used to specify composition filters that filter and match specific events. The Composition Filters model allows the dispatch of messages to software modules.

In this way, the interaction between physical model instances and composition filters takes place at the abstraction level of the DSML, by using domain-specific

events. The message dispatching facility of composition filters provides the means to transform events in the execution of physical model instances to messages that are dispatched to software modules in the GPL. As such, the interaction between composition filters and GPL modules occurs at the abstraction level of the GPL.

Traditional code generation approaches to compose models specified in a DSML with software modules written in a GPL lead to a number of issues; these approaches often create black boxes in software that have strict interfaces, causing tight coupling and tailoring of other software modules around the generated code. Furthermore, the domain-specific physical models from which the code is generated are often not included with the embedded control software or with its documentation, making the generated code harder to understand. By providing rich interaction at the abstraction level of both languages, we reduce or prevent the problems that traditional code generation approaches have.

Aspect-Oriented Quantification

An additional benefit provided by the Composition Filters model is the possibility to perform aspect oriented quantification over physical models and events in these physical models. This enables the specification of more generic interaction: instead of matching a specific event in a specific physical model instance, it is possible to match a class of events in multiple physical model instances. For example, logging of all inconsistencies in all physical model instances can be performed.

Declarative Composition Language

The Composition Filters model provides a declarative language to specify event filtering and matching. A declarative language improves the analyzability of the composition. This provides several benefits, such as better error detection and efficient compilation of the composition between physical model instances and software modules. Chapter 3 explores the analysis of the composition of physical model instances with software modules in detail.

2.9 Discussion

In this section we discuss some additional aspects of our approach.

2.9.1 Applicability on Systems with Tight Timing Constraints

An important aspect in the design of embedded software is dealing with tight timing constraints [76]. Therefore, programming languages and tools aiming at embedded software should support the creation of software that is able to meet timing constraints under all circumstances. This support includes efficient software execution, software execution that is deterministic in time and memory and the ability to precisely analyze the software with respect to its timing behavior.

The implementation of our approach contains a runtime infrastructure with an interpreter to execute the SIDOPS+ models and the composition filters. Such an

interpreter-based approach raises the question whether the resulting execution of the embedded software is efficient and deterministic in time, i.e., whether the embedded software is able to meet its timing constraints under all circumstances. In this chapter, we did not address this concern. We mainly focused on combining a DSML for physical modeling with a GPL, to improve the quality characteristics *Maintainability* and *Reliability*. With respect to these quality characteristics, an interpreter-based execution environment is sufficient to demonstrate our approach.

To enable industrial application of our approach, efficiency and deterministic operation are definitely concerns that should be addressed. Efficient compilation instead of applying an interpreter is part of our future work. Efficient compilation algorithms for aspect-oriented languages, such as the Composition Filters model, are known in literature, e.g., in [21, 33, 34]. The 20-sim tooling contains code generators to compile 20-sim/SIDOPS+ models to efficient and deterministic C code [1]. Future work involves how these different compilation approaches can be combined, to be able to efficiently compile the composition (using the Composition Filters model) of the SIDOPS+ models with the GPL modules.

2.9.2 Separating Design Rationale from Control Logic

The design of a controller depends heavily on the physical characteristics of the system being controlled. Control engineers study these physical characteristics and decide on the type of controller to use (e.g., PI controller, PID controller) and the parameters for this controller, to obtain certain desired characteristics, such as stability, small error margins, reaction time, etc. As such, these physical characteristics are part of the design rationale of the control logic.

A variety of work has been performed on specifying control logic on a higher abstraction level, independent of specific physical characteristics of the system. The actual control logic is then generated from the higher-level specification and a model of the physical characteristics. In this way, the specification of the control logic is less vulnerable for changes in the physical system. An example of such work is the work on supervisory control synthesis by Rooda et al., e.g., in [83, 100].

The described method in this chapter does not aim at specifying and separating this design rationale from the control logic. Also, this chapter did not introduce new types of control logic. This chapter presented techniques to modularize and compose models of physical characteristics that are being applied in the control logic, as computational functionality, not as design rationale of the control logic. Control engineers still decide on which models of physical characteristics are part of the computational logic and in what ways these models interact with other control modules. The models of physical characteristics are for example used to model the actual system state, to model a desired system state or to model constraints on the state of the system.

2.9.3 Control Logic in a DSML

In this chapter we made a distinction between physical characteristics and control logic. Physical characteristics describe aspects of the physical system that are valid independent of the control software, such as a natural relationship between certain

physical variables (e.g., the relationship between T_{ph}, $T_{contact}$ and v that defines acceptable print quality). Control logic are those computations designed to actively influence the state of the system, such as a PID controller that calculates a certain output signal based on some input signals and the controllers state.

Besides being applied to model physical systems, toolsets such as 20-Sim and Matlab Simulink can also be applied to model continuous control logic. The computational model for continuous control logic is similar to the computational model for physical characteristics. For example, in the SIDOPS+ language, continuous control logic can be specified using equations. Therefore, our approach can also be applied to SIDOPS+ models containing continuous control logic. This provides the additional benefits of the Composition Filters model, such as the possibility to implement aspect-oriented functionality on continuous control logic.

In this chapter we did not investigate and utilize the capability of the 20-Sim toolset to model the control logic. We only used the 20-Sim toolset for the domain-specific modeling of the physical characteristics being applied in the control software. This was done to illustrate the possibility to model certain concerns of the control software with a DSML and to illustrate that the models specified in a DSML can be composed with modules specified in a GPL that implements other aspects of the system. If we also applied the DSML to model the control logic, the amount of GPL code in the example cases would be limited. However, this is not the case in general; the limited amount of GPL code is caused by the fact that the example cases are limited, and only contain a subset of the concerns of realistic control software. Examples of concerns that are part of realistic control software are management of system states (e.g., idle, start-up, available), scheduling of (discrete) tasks, recovering from errors, monitoring the available resources (e.g., the amount of toner, number of sheets of paper in the paper tray), processing of user input, security, communication, maintenance tasks, etc. These concerns were left out of the example case studies, as they would require and extensive introduction of the example case studies and of the domain knowledge needed to understand them. Toolsets such as 20-Sim and Matlab Simulink are not designed to handle these concerns. Therefore, still a substantial amount of GPL code is needed.

2.9.4 Risks of Physical Model Decomposition

In this chapter we presented techniques to modularize physical models used in control software and to compose the physical models with other software modules. The techniques ensure that a given modularization and composition is correct from a language perspective (i.e., it results in executable software). However, this chapter did not attempt to provide techniques that ensure the correctness of the modularization and composition of physical models from the perspective of the application. It is up to the engineers of the system to ensure the correctness of the modularization and composition from the application perspective.

If the modularization and composition of physical models in software is not performed with care, undesirable side effects can occur in the behavior of the control software and as such in the behavior of the physical machine. Examples of undesirable side effects are unforeseen interactions between physical models or between

a physical model and a GPL module and deviation from desirable behavior. These problems are related to modularization, as the internal working of a module can be hidden for the developer of another module. If the interfaces of the modules are not properly specified, unforeseen interactions can occur. Note, however, that these issues concerning modularization are general and not specific for the modularization of physical models in control software.

2.9.5 Difference between 20-Sim Simulation and Physical Model Execution

In Section 2.4.5 we presented the algorithm used to execute physical models in software. Physical modeling tools, such as 20-Sim [1], also execute physical models to simulate the behavior of the modeled system. The equation solving procedure in our implementation is largely the same as the procedure used in 20-Sim. However, there are some differences.

These differences are caused by the fact that the execution of physical models in software relies on externally provided updates of physical variables. We introduced the possibility to have multiple ways to determine a value, possibly leading to inconsistencies in the physical model. We added a mechanism to solve inconsistencies. However, the possibility of having multiple ways to determine a value leads to differences in which algebraic loops are solved and to differences in the execution order of equations.

Algebraic Loops

Because there are multiple ways to determine a value, the handling of algebraic loops is different:

- If the value of one of the variables (referred to as variable a) in the loop is also provided from another source (e.g., a sensor or a different physical relationship not part of the algebraic loop), then the loop is solved using this value. This leads to multiple values for the variable a, because of the loop. This way of solving can be emulated in 20-Sim in the following way:
  - Replace the physical variable that can have multiple values with multiple physical variables, one for each way to determine the value (thus breaking the loop).
  - Add logic to compare the values of the multiple physical variables derived from the original physical variable.
- Otherwise, the loop is solved using methods also applied by 20-Sim (not implemented in our current tooling).

Evaluation Order

Because there can be multiple sources for values, the evaluation order of equations can be ambiguous. To aid the designer, we implemented a specific evaluation order, giving preference to forward solving of equations before backward solving.

Replacing a physical variable that can have multiple values with multiple physical variables eliminates this ambiguity. In this way, the 20-Sim execution method can be used.

2.9.6 Dynamic Adaptation of Physical Models

The application of physical model instances can be made more flexible, if the physical model it refers to is flexible. If it is for example possible to add, remove, replace submodels from the physical model, the behavior of the physical model instance can vary at runtime, to reflect for example discrete changes in the physics of the system or changing constraints. An example of changing constraints is the constraint between $T_{contact}$ and T_{PH} in the Warm Process case. The constraint presented by Equation 1.1 on Page 16 is only required in the running state of the printing machine. If the printing machine is for example in the idle or startup state, the constraint is different. By making it possible to manipulate the physical model at runtime, such examples of adaptive behavior can be easily implemented.

2.10 Related Work

2.10.1 Domain-specific Models in Embedded Control Software

Domain-specific models are commonly used in the development of embedded software. For example, the Giotto approach [59] is a domain-specific language that aims to separate timing from functionality in embedded control software. A timing specification, called Giotto timing program, can be generated from a simulink control model. The timing program supervises the functionality programs, which implement the control functionality. The timing program is independent of a given implementation platform, supporting portability between different platforms. The Giotto approach provides tooling to verify whether a given implementation platform can handle the timing constraints given in the Giotto timing program. The Giotto approach compares to our approach as it also applies a DSML to implement part of the computational logic for embedded control software. Interesting in the Giotto approach is that the DSML used is a DSML to model physical characteristics and continuous control logic, i.e. a DSML similar to the SIDOPS+ language. In this way, the Giotto approach can be an addition to our approach.

There are approaches that apply aspect-oriented techniques to domain-specific modeling approaches. An example of this is the C-SAW approach [55]. C-SAW provides a technique and tooling to add aspects to software models, which are higher abstraction levels of the software implementation. This differs from our approach in two ways. The first difference is that the models referred to in our approach are not software models, but models of physical characteristics (in general, models of domain concepts). These models of physical characteristics are applied in control software to provide part of the computational logic, which is realized by adding execution semantics to the modeling language. The second difference is that our approach does not aim to express aspects in the domain-specific models themselves, but uses

aspect-oriented composition technology to compose the models in the DSML with the modules in the GPL on a conceptual level.

2.10.2 Interaction Based Approaches

One usage of physical models in embedded control software is to implement the interaction between different control modules. The application of our approach to the Drum Shuttling case study contains an example of such a usage of physical models: the `stepperMotor,cam` physical model instance provides the interaction between the `ShuttlingController` and `StepperController`. In this section we compare our approach to existing interaction and coordination approaches.

Contracts [58, 63] were introduced to explicitly model interactions among a group of objects. These constructs capture behavioral dependencies specified with a set of preconditions and invariants [58]. The contract principle was also applied to component-based systems, to extend component interfaces with behavioral constraints [18]. Hereby, contracts are used to check the compliance of components. Similarly, contracts that provide formal semantics regarding object interactions enable conformance checking at compile-time [58]. As such, these approaches support maintenance and reuse of software systems. In our work, we use the formal specification not only for domain-specific analysis and checking for compliance but we also compose the specified models with other software modules and execute them. So, the specification becomes a part of the implementation.

Meta-object protocols (MOP) [70] have been introduced as supplemental constructs to programming languages by means of which the language's behavior can be modified. As such, semantics of a program becomes open and extensible. MOPs, as well as reflection [80] mechanisms, could be used to explicitly model the interaction between modules that is caused by the physical characteristics. In our work, we have proposed a domain-specific solution to specify these physical characteristics in a declarative way and to support domain-specific analysis. Furthermore, we apply the aspect-oriented Composition Filters model to compose DSML models with software modules. Such aspect-oriented techniques can be supported using reflection and meta-object protocols [22].

There have been efforts to abstract away and encapsulate the coordination among a number of computational entities (e.g., objects, components, etc.). These efforts led to coordinated behavior abstractions [12], and later on to coordination models and languages [90]. These approaches improve reusability by enabling explicit modeling of interaction, enforcement of invariant behavior, and separation of interaction details from the computational concerns. Modeling coordination is not the aim of this work and as such we have not proposed an alternative model or language for this. Instead, we have introduced a modular extension to state-of-the-practice languages for explicitly modeling interaction caused by physical characteristics. The models that deal with the specified interaction are composed with the existing GPL modules using the Composition Filters model.

Service orchestration is used to compose multiple (web-)services in a meaningful way to create a larger application. The orchestration language deals with issues as sequencing of the services to call, parallel calls to different services, branching in

the services to call, etc. Examples of languages to perform service orchestration are BPEL [8] and Orc [72]. Service orchestration is specifically aimed at coordination among services. Our approach can capture interaction caused by physical characteristics, using domain-specific abstractions.

2.10.3 Connection with System Modeling Approaches

System architects use tools to model different aspects of the physical system, such as the physical structure, physical behavior and interaction between components in the system, state of the system etc. There are several approaches to integrate such models and describe the system from different perspectives so that consistency can be maintained and changes in one model can be automatically reflected in other models. Example of such approaches are the Knowledge Intensive Engineering Framework (KIEF) [111] and the Speeds approach [92]. Such efforts can be extended to include software specifications, because software functionality is heavily based on aspects of the physical system, such as the structure and the physical interaction between components. Furthermore, such an integration can provide traceability between the physical relationships and the corresponding structures in the system models. This can be helpful in case of evolution, as the impact of changes in the system to the physical relationships can be automatically traced within the embedded control software.

KIEF

Forbus describes in [50] the concept of qualitative process theory. This is the analysis and specification of the qualitative relationships between different physical variables in a physical system. Such a specification can be used to predict behavior in the system. This theory has been incorporated in KIEF in the form of parameter networks. Parameter networks are graph structures that describe the qualitative relationships between physical variables [111]. They can be derived from other system models, like structural models.

These parameter networks are similar to the concept of dependency graphs that we introduced in Section 2.4.2. The difference between them is that dependency graphs provide a quantification on these relationships, while parameter networks only qualitatively describes the relationships. So, parameter networks reflect the graph structure of dependency graphs, but without the equation nodes in the graph.

Although the parameter networks do not describe quantitative relationships, they can be used as a starting point to derive the quantitative relationships needed to create the physical models. First, the parameter network can be used to check whether certain variables are solvable. Secondly, if the quantitative mapping is possible, the part of the parameter network that reflects this mapping can be used as the starting graph structure for the dependency graph. The relationships in this graph should then be quantified to create the dependency graph.

Speeds

The Speeds approach [92] is an embedded system design methodology that supports the composition of heterogeneous subsystems using semantic-based modeling meth-

ods. To compose different models of the system, the approach defines the concept of *heterogeneous rich-component* model that can represent different functional and architectural abstractions in embedded system models, such as timing and safety properties. Although the Speeds approach offers composition of different models, the Speeds approach differs from our approach as it provides an integrated approach for embedded system modeling instead of an approach to apply such heterogeneous (domain-specific) models in embedded control software and to compose these models with GPL software modules.

The integration of the Speeds approach with our approach is interesting future work; this integration can provide composition between heterogeneous domain-specific models in embedded control software, in addition to the composition of DSML models with GPL modules.

2.10.4 Heterogeneous Composition of Computational Models

The Ptolemy approach [44] provides a way to model *embedded computational systems* (not necessarily limited to software) that consist of heterogeneous types of components (i.e., types of components that differ in how they communicate and interact). Arbitrarily composing heterogeneous components can lead to ambiguities in the interaction between the components (due to the differences in the way these components communicate and interact), resulting in unexpected emerging behavior. To prevent such problems, the Ptolemy approach uses hierarchical nesting of different types of components, resulting in a homogeneous composition at each hierarchical level.

The basic building block in the Ptolemy approach is called an actor. Actors can communicate with each other through ports. A system of multiple communicating actors can be encapsulated into a higher-level actor. In this way, hierarchical structures can be created. The way data flow and control flow between actors is performed at a certain hierarchical level is not defined by the actors themselves, but by a separate model of computation (MoC). Different composite actors can have different MoCs. Some MoC implementations (also called domains) that have been realised are: Communication Sequential Processes (CSP), Continuous Time (CT), Discrete Event (DE), Process Network (PN) and Synchronous Dataflow (SDF).

Our approach of combining the semantics of the DSML for physical models with the semantics of the GPL is similar to the Ptolemy approach. The semantics of physical models is encapsulated in the physical model instance. This semantics is well-defined and there is no interference with the semantics of the GPL. The modules in the GPL are oblivious to this internal semantics of the physical model instance. The interaction between the physical model instance and the modules specified in the GPL is performed using the base interface and the Composition Filters model, which is natural from the perspective of the GPL. This mechanism can be seen as two layers of hierarchical heterogeneous composition from the Ptolemy approach (the physical model semantics is encapsulated into physical model instance actors, which can be composed with other GPL modules/actors). This prevents ambiguity in the control flow and data flow, thus preventing unexpected emerging behavior.

2.11 Conclusion and Future Work

In this chapter we discussed the implementation of physical models in embedded control software. In current practice there are two common approaches to implement physical models. The first approach is to use a general-purpose programming language (GPL) to implement physical models. However, this means that domain-specific abstractions of the physical models need to be transformed to implementation abstractions in the GPL. This transformation introduces accidental complexity, which reduces software quality considering certain software quality characteristics such as comprehensibility, maintainability and reusability.

The second approach is to use a domain-specific modeling language (DSML) to specify and execute physical models. However, embedded control software usually also contains other, application-specific functionality, which cannot easily be implemented with a DSML for physical models. Examples of other functionality in embedded control software is scheduling of (discrete) tasks, recovering from errors, processing user input, security and communication. Usually, a GPL is used to implement this application-specific functionality. Code-generation techniques are applied to compose physical models specified in a DSML with GPL modules. However, code generation usually leads to black boxes with inflexible and tightly integrated interfaces, reducing the comprehensibility and maintainability of the embedded control software.

The main contribution of this chapter is a novel approach to represent physical models in embedded control software using a DSML and to compose physical models with GPL modules at the abstraction level of both the DSML and the GPL using the Composition Filters model. As such, this approach provides the benefits of a DSML to specify physical models with the freedom of a GPL to implement other application logic, without the drawbacks of code generation approaches.

We adopted the SIDOPS+ language from the 20-Sim toolset as the DSML to specify physical models. The 20-Sim toolset is commonly used to model and simulate physical systems and control logic. To execute physical models that are specified in SIDOPS+ we introduced the concept of *physical model instance*. A physical model instance maintains the state of the physical variables in the physical model. We specified and implemented executions semantics for physical model instances in an evaluation algorithm, which is divided into three different evaluation strategies. Applying a DSML to specify and execute physical models provides the following benefits:

- Reduction of accidental complexity that is introduced when domain-specific abstractions have to be transformed to implementation abstractions in a GPL. This improves comprehensibility, maintainability and reusability of embedded control software.

- Separation of physical models from other application logic, reducing tangling and scattering of physical models in embedded control software.

- Explicit representation of physical models, improving the comprehensibility and evolvability of embedded control software.

To compose physical models specified in SIDOPS+ with GPL modules we applied and extended the Composition Filters model. The Composition Filters model makes

it possible to specify interaction between physical model instances and GPL modules in terms of messages (GPL) and execution events (physical model instance). In this way, the Composition Filters model provides the following benefits:

- Interaction between physical model instances and GPL modules at the abstraction level of both the GPL and the DSML, due to the combined message and event mechanism. This provides a richer interaction mechanism than traditional code generation techniques provide, improving the comprehensibility, maintainability and reusability of embedded control software.
- Loose coupling between physical model instances and GPL modules, because their interaction is separated from their implementation. This improves the comprehensibility, maintainability and reusability of embedded control software.
- Aspect-oriented quantification. This enables more generic interaction, which reduces tangling and scattering of certain concerns. This improves the comprehensibility, maintainability and reusability of embedded control software.
- A declarative composition language. This enhances the analyzability of the composition, which reduces the possibility of errors remaining undetected.

We implemented our approach in a toolset and demonstrated our approach on the two industrial case studies. Furthermore, we provided a qualitative evaluation of our approach. In Chapter 5 the toolset and techniques will be further evaluated, as part of the integration with the other research presented in the coming chapters.

Future work includes the integration of our approach with system engineering tooling, such as the KIEF approach [111]. Furthermore, additional research will be performed on efficient compilation techniques for our approach. The combination of our approach with other DSMLs, such as the Giotto approach [59], is going to be investigated, as well as the possibility to integrate the Speeds approach [92].

Verification & Analysis of Physical Models in Embedded Control Software[1][2]

3.1 Introduction

Many embedded systems today are largely controlled by software. For example, the control of overall system behavior in digital document printing systems is mainly performed by software. The size and complexity of such embedded software systems is continuously increasing due to the demand for new functionality, increasing variety in the types of hardware that need to be controlled and more refined and optimized control. Despite this trend, the control software must be kept reliable, even in varying circumstances such as changing environmental conditions, changing usage profile and evolution of system properties.

We have seen in Chapter 2 that models of physical characteristics can be part of embedded control software. The accuracy of such physical models is crucial for correct and optimal functioning of the system. However, errors can occur in these physical models and in their composition with GPL modules due to a number of reasons, e.g., *i)* data received from the sensors is inaccurate, *ii)* there are undiscovered faults in the implementation of the physical model, and *iii)* the working context of the system or system properties have changed, for example because of wear and tear of the hardware, resulting in invalid implementations of the physical models. To detect such errors, the accuracy of the physical models and the correctness of their composition with GPL modules needs to be verified.

The proper functioning of physical models depends on the physical circumstances in which the system is used. At design time, the system is extensively tested under a wide variety of circumstances. However, testing the system in all possible circumstances is impractical, if not impossible. Therefore, it is necessary to detect and cope with remaining defects at runtime. To detect and cope with defects at runtime, the behavior of the software and the system needs to be observed. Monitoring the behavior of the software and the system at runtime to detect defects is called runtime

[1] Part of the content in this chapter is based on a paper published at the 5th International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2011 [36].

[2] A version of this chapter is under review for publication in a journal.

verification [16]. We apply runtime verification to verify the accuracy of physical models in control software, as a complementary approach to testing for increasing the reliability of the embedded system.

Observers and monitors are essential ingredients of runtime verification. In traditional runtime verification techniques, the properties that need to be verified are specified in terms of software states and events. From these specified properties, observers and monitors that perform runtime verification are generated. However, this approach is inadequate for verifying the physical models implemented in software; when physical models are verified, it is checked whether they are consistent with physical reality. This consistency is not directly apparent from the encountered states and events during the software execution. As such, it is not straightforward to specify monitors in terms of the traditional concepts of states and events and properties on these states and events that need to be verified.

In this chapter, we propose a novel approach to verify physical models at runtime, exploiting redundancy in the physical model, e.g., in the form of multiple relationships for the same physical variable or additional sensors. This approach is inspired by control engineering literature on state observers [98].

As explained in Chapter 2, physical models are composed with software modules using the Composition Filters model. The Composition Filters model provides a declarative way to specify the filtering and matching of messages and events. This enables powerful static analysis of the behavior of a set of composition filters. We call this analysis *filter reasoning*. The obtained information can be used for various purposes, which includes consistency analysis, redundancy detection and GPL code generation. In this chapter, we present the techniques to perform filter reasoning.

To summarize, this chapter provides the following contributions:

- A method to verify the consistency of physical models in control software with physical reality at runtime, and diagnose encountered problems.
- Static analysis techniques to verify the set of composition filters that composes the physical models with other software modules specified in the GPL.

This chapter is organized as follows. First, the problems and challenges concerning both runtime verification of physical models and analysis of the composition filters are discussed in detail. Then, the approach to verify physical models at runtime is explained. This is followed by the approach to analyze the composition filters used to compose physical models with GPL modules. Then, an application of the runtime verification approach is explained: calibration of the physical state within the physical model instance and corresponding detection of a malfunctioning sensor or system component. Finally, a discussion of relevant issues and concerns is provided, related work is discussed and a conclusion is given.

3.2 Problem Statement

This section motivates why physical models need to be verified at runtime and explains the challenges that arise when we want to perform this verification. Furthermore,

this section motivates why the composition of the physical models with GPL modules should be analyzed.

3.2.1 Verifying Physical Relationships

The previous chapter introduced a technique to compose physical models specified in a DSML with other software modules implemented in a GPL. The physical models that are composed with GPL modules, however, might not always be, or remain, accurate or correct. There are different reasons for this, which include:

1. The underlying assumptions on which the physical models are based might not be accurate enough. For example, the physical relationships might not accurately describe the physical reality.
2. The system might be used in different operational conditions than envisioned during design. For example, a printer system might be applied in different environmental conditions than expected, a new type of paper might be used, etc.
3. Physical characteristics might change over time, because of, for instance, wear and tear of physical components in the system.
4. Physical characteristics might change because of evolution. For example, changes in the physical hardware influence the physical characteristics. If they are not updated accordingly in the implemented physical models, this can cause failures.
5. The engineer implementing a physical model might introduce a fault.

Incorrectness in an implemented physical model might lead to failures in the behavior of the system. In the embedded software domain it is not possible to test the software in all possible physical conditions. Therefore, certain faults might remain undetected. To increase the reliability of the system, runtime verification is essential.

3.2.2 Failures Observable in Physical System Behavior

Faults in implemented physical models lead to observable failures in the physical behavior of the system, but not necessarily to observable failures in software behavior. This hinders the application of common runtime verification techniques that focus on monitoring software behavior only.

Common runtime verification techniques usually consist of three parts [37]. First, there is a data and/or event model in which the software can be described. Second, there is a logic to specify properties of such models. These are properties that have to be valid or properties that should never occur. Examples of specification logics are regular expressions and temporal logics. Third, there is a specification of the actions that need to be performed when the specified properties are violated.

In embedded control software we want to verify whether the implemented physical models correspond to physical reality; we want to verify whether the actions performed

in the control software lead to the correct behavior of the physical system. Failures in the implemented physical model become apparent in the behavior of the physical system, e.g., in reduced print quality, but are not always observable by monitoring software behavior only; the same state or sequence of events in software behavior might in one case lead to correct behavior of the physical system while under different circumstances it leads to incorrect behavior. As such, common runtime verification techniques are insufficient to perform this kind of verification, as their data and event models only include the software state and software actions; the state of the physical system is not explicit in the software implementation.

3.2.3 Analyzing the Composition

The Composition Filters model provides a declarative language to specify filtering and matching of messages and events. A declarative language enables powerful static analysis of the behavior of a set of composition filters. We call this type of analysis *filter reasoning*. The obtained information can be used for various purposes, which includes consistency analysis, redundancy detection and GPL code generation.

Consistency Analysis

A filter set consists of a number of filter modules. Filter modules consist of a number of filters. The specific matching and selection of composition filters, the combination of filters into filter modules and the ordering of the filter modules in the filter set might cause certain consistency conflicts in the filter set, such as unreachable filters and filters that never accept a message or event. An example of a filter set with a consistency conflict is shown in Listing 3.1. This listing shows two composition filters, which are superimposed on the `BeltTemperature` model of the Warm Process case study. The first filter is of type `Result`, which means that if the filter accepts, the `resultValue` property is set and the control flow returns. Because of this behavior, events that are accepted by the first filter never reach the second filter. However, the set of events accepted by the second filter is a subset of the set of events accepted by the first filter. Because all events in this set never reach the second filter, the second filter can never accept[3].

```
1 tbeltHandler: Result = (event.variableName=='Tbelt' &
      event.eventType=='Inconsistency')
      {event.resultValue=event.values['Tbelt_sensor']};
2 tbeltLog: Logging = (event.variableName=='Tbelt' &
      event.eventType=='Inconsistency' & event.margin > 0.2);
```

Listing 3.1: Example of an inconsistent set of composition filters

The inconsistency in this example might have been caused by an ordering problem; if the filters are specified in different filter modules that both are superimposed on the `BeltTempererature` physical model, two orderings are possible. The presented ordering leads to consistency conflicts. The other ordering (in which `tbeltLog` comes before `tbeltHandler`) does not lead to consistency conflicts.

[3]Note that other events can reach the second filter, namely those events that are not accepted by the first filter.

Consistency conflicts that are caused by the ambiguous nature of filter module orderings fall in the category of *aspect interaction* problems of aspect-oriented languages. General information about aspect interaction problems and how they can be detected and resolved can be found in [39].

Redundancy Detecting

Redundancy in the physical model is defined as the existence of multiple ways in which the value of a variable is updated before and during the evaluation algorithm that is explained in Section 2.4.5. These multiple values can come from the following sources:

- From `update` calls to the base interface of the physical model instance, before the execution of the evaluation algorithm. We call a value from this source a *set value*.
- From a provided result by a composition filter that matches the corresponding *CheckUpdate* event. We call a value from this source a *request value*. The existence of this source of values for a specific variable can be detected by analyzing the filter set using filter reasoning.
- From an equation during the execution of the evaluation algorithm. We call a value from this source an *equation value*.

Code Generation

Currently, the execution of a physical model instance and the execution of the composition filters that compose the physical model instance with GPL modules are performed using an interpreter. One disadvantage of an interpreter is its performance overhead. By compiling the physical model instance and the composition filters into GPL code, efficiency can be improved. To translate a filter set into GPL code, the behavior of the filter set needs to be known. Filter reasoning can provide this information.

3.3 Runtime Verification of Physical Models

This section presents a novel approach to verify the correctness of physical models at runtime.

3.3.1 Using Redundancy to Verify Correctness

To verify physical models at runtime, we need to check whether these physical models correspond to actual physical reality. To do this, we generalize the techniques used for state observers [98] in control engineering. State observers are implementations of physical models that provide more information about the system's state than is available from sensors. They are kept consistent with the system's state by calibrating the model's state with (redundant) information known from sensors [98].

State observers are one application of physical model instances. Redundancy can be present in physical model instances. In this case redundancy means that certain physical variables in the physical model have multiple sources for their values. Examples of such sources are:

- *Additional sensors to determine the value of a physical variable, besides an already existing relationship in the physical model to calculate the value for the same physical variable.* This is a situation that is rare at system deployment, as the physical models and the physical relationships within the models are introduced because there is no available sensor information. One application of such sensors is to calibrate the physical state in the physical model instance, for which the sensor only occasionally gives a reading. The Drum Shuttling case study introduced in Section 1.3.2 contains such a sensor.

- *Redundancy in the physical relationships in the physical model.* If there are multiple physical relationships in the physical model that calculate the value of the same physical variable, the results can be compared.

The following example shows an extension of the Warm Process case study that adds redundancy to the `BeltTemperature` model.

Example 3.1 Redundancy in the BeltTemperature Physical Model

The description of the Warm Process case study in Section 1.3.1 shows that there is a sensor to measure the belt temperature. Listing 3.2 shows the composition filters specification that retrieves the sensor value when there is a *CheckUpdate* event. A composition filter specification is used, instead of the base interface, to separate the interaction between the `IO` software module and the `BeltTemperature` physical model from the implementation of this software module and physical model, as was explained in Chapter 2.

```
1 filtermodule BeltTemperatureSensor{
2   externals
3     bTSensor: Sensor = IO.getBeltTemperatureSensor();
4   outputfilters
5     bTFilter: Dispatch = (event.variableName=='Tbelt' &
          event.eventType=='CheckUpdate') {target=bTSensor;
          selector='getValue'};
6 }
```

Listing 3.2: Filter module specification that retrieves the sensor value for Tbelt

Suppose that the `BeltTemperature` physical model also contains the following equation:

$$T_{belt} = c_5 \cdot (T_{contact} + c_6 \cdot T_{ph}) \cdot \sqrt{v} \tag{3.1}$$

This equation determines the temperature at the sensor location (T_{belt}) from other physical variables ($T_{contact}$, T_{ph} and v). Listing 3.3 shows the resulting SIDOPS+ specification of the `Belt Temperature` model, also including Equation 1.2 to determine $T_{contact}$.

```
1  ...
2  equations
3    Tcontact = c4 * Prad / sqrt(v) + Tbelt; \\Eq 1.2
4    Tbelt = c5 * (Tcontact + c6 * Tph) * sqrt(v); \\Eq 3.1
```

Listing 3.3: SIDOPS+ specification of the belt temperature model

So, in this example there are multiple sources for the value of T_{belt}: the Equation 3.1 and the composition filter that returns a sensor reading after a *CheckUpdate* event. Figure 3.1 shows the dependency graph corresponding to this example. This graph clearly illustrates multiple incoming edges to the *Tbelt* node.

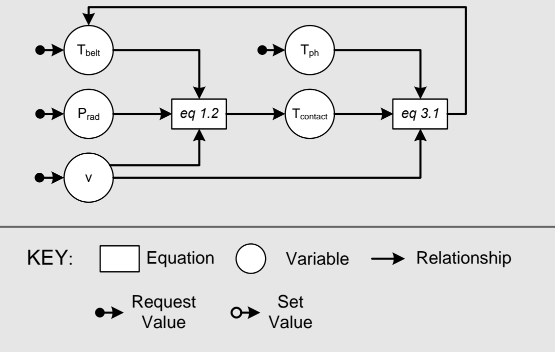


Figure 3.1: Dependency graph of the `BeltTemperature` model

Extension of Dependency Graphs

For runtime verification, the dependency graph of a physical model is extended at runtime with certain *source edges*. Source edges are edges that have no start-point and the end-point is a variable node. Such edges indicate that the value of the variable corresponding to the variable node has been updated from outside the physical model. There are two types of source edges: *request value edges* and *set value edges*. Request value edges indicate that the value of the variable has been updated by a composition filter that handled the corresponding *CheckUpdate* event. Set value edges indicate that the value of the variable has been updated using an `update` call to the base interface of the physical model instance. Figure 3.1 shows request value edges for the variable nodes T_{belt}, P_{rad}, v and T_{ph}.

Detecting Redundancy in the Physical Model

Redundancy is easily recognizable in dependency graphs; there is a redundant calculation if a variable node has multiple incoming edges. This means that the corresponding variable can be calculated in multiple ways. The dependency graph in Figure 3.1 contains one redundant calculation. The variable T_{belt} can be determined in two ways; it can be derived from a sensor input and through Equation 3.1. Note that there is actually a cycle in the graph, in which T_{belt} depends on itself through

Equations 1.2 and 3.1[4].

Checking the Consistency of the Physical Model

Analogous to state observers, the redundancy in the physical model instance can be utilized to verify whether the corresponding physical model is consistent with physical reality. The physical model is consistent if all determined values for the variable are equal. If not all values are equal, one of the value sources is not correct. The `Evaluator` component in the interpreter evaluates all possible sources of values for the physical variables. It also checks whether the multiple outcomes are consistent. If not, it issues an *Inconsistency* event, as was explained in Section 2.5.5.

Handling Inconsistency

To handle inconsistencies in the physical model, composition filters can be specified to filter and match the issued *Inconsistency* events. Listing 3.4 shows an example of a composition filter that matches *Inconsistency* events of physical variable *Tbelt*. The type of the filter is `Logging`, which means that it performs a logging operation when it matches.

```
1 filtermodule TbeltLogging{
2   outputfilters
3     tbeltLog: Logging = (event.variableName=='Tbelt' &
          event.eventType==Inconsistency & event.margin > 0.1);
4 }
```

Listing 3.4: Composition filters specification to log inconsistencies in *Tbelt*

Coping with Deviation in Outcomes

In reality, there are often small deviations between the outcomes, because of small error-margins in the sensors or formulas, for which the system has been designed to be robust. This means that when the difference between the redundant values is within a certain safe range, there is no indication of a failure. Therefore, to do monitoring and checking, such error-margins should be taken into account. The event model introduced in Section 2.5 provides means to ignore small deviations by using the `margin` property. The example in Listing 3.4 demonstrates this property; it only matches for *Inconsistency* events of *Tbelt* if the margin is larger than 0.1.

3.3.2 Diagnosing Faults

When the different values for a physical variable are inconsistent, this indicates that there is a failure. The next step is diagnosing the cause of the failure. There are multiple possible causes for failures, e.g., a malfunctioning sensor or actuator, a failing physical component or a fault in the implementation of the physical relationship in the

[4] The evaluation algorithm introduced in Section 2.4.5 can cope with cycles in the dependency graph, as it evaluates each equation node at most once. As such, the evaluation algorithm evaluates cycles differently than the 20-Sim simulation environment, in which cycles are considered to be algebraic loops that need to be solved [73].

model. Possible causes can be determined using the derivation graph of the physical model. The definition of a derivation graph was introduced in Section 2.4.6.

Systematic fault diagnosis starts in the derivation graph at the variable node that corresponds to the inconsistent physical variable (v_2 in the example). In a systematic way, the derivation graph is traversed backward, where each type of graph structure encountered provides fault diagnosis information. Table 3.1 shows the different possible graph structures that can be encountered. In black, the encountered graph structure is shown. In gray, the graph structure that is traversed next is shown. The fault diagnosis information that can be derived from each graph structure is explained next.

Structure	Name
v_i	Variable node
eq i	Equation node
	Request value edge
	Set value edge
eq i	Equation result edge
v_8	Variable value edge
Key: Black = Encountered structure; Gray = Next traversed structure	

Table 3.1: Diagnosis encountered structures

Variable Node

A variable node leads to the following fault diagnosis information:

- The value of one of the incoming edges is incorrect.

If the variable node has been visited before, this branch of the traversal of the graph ends. Otherwise, the traversal continues to all incoming edges.

Note that a variable node only has multiple incoming edges if it has multiple sources of information. As the traversal algorithm cannot detect from the derivation graph which source of values is used as the output value, it has to traverse all incoming edges. A record of which incoming value is used could be

maintained during the execution of the evaluation algorithm. The availability of such information can make the diagnosis more precise, as only the single edge that is the source of information needs to be analyzed. Note also that the systematic fault diagnosis always starts at a variable node, namely the inconsistent variable node.

Equation Node

The equation node leads to the following fault diagnosis information:

- The equation itself is incorrect;
- Or, the value of one of the incoming edges is incorrect.

If the equation node has been visited before, this branch of the traversal of the graph ends. Otherwise, the traversal continues to all incoming edges.

Request Value Edge

The request value edge leads to the following fault diagnosis information:

- The outcome of the filter evaluation is incorrect.

At this point, the traversal of the derivation graph stops. But fault diagnosis could continue with the filter set, deriving the source of the value, e.g., a software module that provides a sensor reading. This could mean that the software module does not work correctly, or the sensor is broken. Section 3.4.3 explains how such analysis of the filter set can be performed.

Set Value Edge

The set value edge leads to the following fault diagnosis information:

- The provided value is incorrect.

This is also an end-point of the traversal of the derivation graph. GPL code analysis, such as the analysis of the call-graph [94], could be performed to determine the source of the provided value.

Equation Result Edge

The equation result edge leads to the following fault diagnosis information:

- The outcome of the equation is incorrect.

The traversal continues with the corresponding equation node.

Variable Value Edge

The variable value edge leads to the following fault diagnosis information:

- The value of the variable is incorrect.

The traversal continues with the corresponding variable node.

The next example shows the result of applying systematic fault diagnosis to the case from Example 3.1.

Example 3.2 Fault Diagnosis

Suppose that in the Warm Process case study the sensor reading for variable T_{belt} gives a different value than the evaluation of Equation 3.1. The dependency graph shown in Figure 3.1 is in this case also the derivation graph. This derivation graph is used for the systematic fault diagnosis, leading to the following possible causes:

1. The *request value* edge is incorrect. This may be caused by a malfunctioning T_{belt} sensor.
2. The outcome of Equation 3.1 is incorrect. This may be caused by:
   (a) Equation 3.1 itself is incorrect.
   (b) The value of T_{ph} is incorrect. This may be caused by:
      i. The *request value* edge is incorrect: This may be caused by a malfunctioning T_{ph} sensor.
   (c) The value v is incorrect. This may be caused by:
      i. The *request value* edge is incorrect: This is the actuator value. It may indicate a problem in the actuation/controlling of v, either in software or in hardware.
   (d) The value of $T_{contact}$ is incorrect $\rightarrow$ the outcome of Equation 1.2 is incorrect. This may be caused by:
      i. Equation 1.2 itself is incorrect.
      ii. The value of T_{belt} is incorrect. We already encountered the T_{belt} variable node (there is a cycle in the graph), so evaluation stops here.
      iii. The value of P_{rad} is incorrect. This may be caused by:
         A. The *request value* edge is incorrect. This indicates a problem in the actuation/controlling of the radiator, either in software or in hardware.
      iv. The value of v is incorrect. This case has already been taken into account in point 2c.

Such derived fault diagnosis information and possible causes can be provided to service engineers to enable them to find the causes of failures more effectively.

3.3.3 Example Applications

This subsection presents a number of example applications of the runtime verification techniques presented in the previous subsections.

Detecting Inconsistencies at Runtime

Inconsistencies in the values for physical variables may indicate that the physical model does not correspond to physical reality. Composition filters can be implemented to monitor for inconsistencies in the physical variables that have a redundant value source. The aspect-oriented features of composition filters make it possible to define a single filter that monitors and handles all inconsistencies. Listing 3.5 shows an example of such a composition filter specification. Line 3 shows the definition of a `Logging` filter that matches all *Inconsistency* events that have a significant margin. All occurrences of these events are logged. Lines 6 to 11 show that the filter module is superimposed on all physical model instances in the system.

```
1  filtermodule InconsistencyLogging{
2    outputfilters
3      inconLog: Logging = (event.eventType=='Inconsistency' &
           event.margin > 0.5)
4  }
5
6  superimposition{
7    selectors
8      models = { M | isModelInstance (M) };
9    filtermodules
10     models <- inconsistencyLogging;
11 }
```

Listing 3.5: Composition filters specification to log inconsistencies

Monitor Wear and Tear

One advantage of the runtime verification approach is that it becomes possible to monitor for wear and tear in the system. We show this using the Warm Process case study and Example 3.1.

The printing system in the Warm Process case study contains a radiator component to heat the toner belt. Over time, the radiator component may get polluted and therefore less efficient. If not detected early enough, this can cause damage to the system.

Detection of radiator pollution has been implemented using the redundancy introduced in Example 3.1. In this example, a second equation was introduced into the `BeltTemperature` model of the Warm Process case study. In this way, the physical variable T_{belt} has two sources for its value: a sensor and the added equation. With a clean radiator, the two sources provide the same value. But when the radiator gets polluted, the two values begin to differ. This can be detected by the composition filter specification shown in Listing 3.6. Lines 5 and 6 show the `radWear` filter. This filter matches *Inconsistency* events of *Tbelt* with a margin larger than 0.2. If the filter matches, it dispatches the message `makeServiceCall` to the `ServiceModule` in the GPL[5].

```
1  filtermodule RadiatorWear{
2    externals
3      serviceModule : ServiceModule = ServiceModule.instance();
4    outputfilters
5      radWear: Dispatch = (event.variableName=='Tbelt' &
           event.eventType=='Inconsistency' & event.margin > 0.2)
6        {target=serviceModule; selector='makeServiceCall'};
7  }
```

Listing 3.6: Composition filters specification to detect radiator wear

Monitor Acceptable Ranges

Besides monitoring for inconsistencies, the Composition Filters model also provides means to perform other runtime verification tasks within physical models. One exam-

[5]Note that wear and tear might not be the only cause for the inconsistency; full diagnosis of the inconsistency has been performed in Example 3.2.

ple is monitoring whether a physical variable stays within an acceptable range. An example of this is shown in Listing 3.7. The specific physical variable in this example is T_{ph}. The `eventType` is *Change*, reflecting the fact that checking is done when the value of the variable changes. The acceptable range for the physical variable is between 60 and 100. If the value is outside this range, the composition filter matches and the event is logged.

```
1 filtermodule TphMonitoring{
2   outputfilters
3     tphMonitor: Logging = (event.variableName=='Tph' &
        event.eventType=='Change' & (value < 60 | value > 100));
4 }
```

Listing 3.7: Composition filters specification to monitor the acceptable range of *Tph*

3.4 Composition Analysis

Using our approach introduced in Chapter 2, physical models specified in a DSML are composed with GPL software modules using composition filters. This section describes how static analysis of the behavior of such composition filters can be performed. Such static analysis is called *filter reasoning*. A number of applications of filter reasoning are described.

3.4.1 Background: Composition Filter Reasoning

A set of composition filters (filter set) that is superimposed on an object or physical model instance defines message and event matching and the behavior to be executed when a message or event matches. The filter set can define different behavior for different messages and events, by matching and selecting based on the properties of the message or event. The behavior of a specific message or event in a given filter set can be statically determined.

Analyzing the Execution of a Filter Set for a given Message

Static analysis of the behavior of messages in a filter set has been extensively investigated in [33]. The general approach given in [33] is:

1. Transform the abstract syntax tree (AST) of the filter set into the corresponding flowchart. This flowchart represents the flow semantics of the filter set. The flow semantics defines how messages can flow through the filter set. As the message matching and filtering semantics is not represented by the flowchart, the flowchart does not represent the behavior of a specific message in the filter set. To analyze this, the next step is performed.

2. Simulate the execution of the filter set for a specific message, using the flowchart for the flow semantics. The message matching and filtering semantics has been implemented in the simulation algorithm. The result of the simulation is a state space of the execution of the message; it contains all reachable execution states

for the given message. An execution state is the combination of the program counter (pointing to the current location of execution) and the properties of the message (which can change during the execution of the filter set). As the filter set can contain conditional branching (other than through property matching), the resulting state space can contain branching. Each path in the state space represents one possible execution of the message.

Analyzing all Possible Executions of a Filter Set

For certain applications it is necessary to analyze all possible execution paths through the filter set, independent of the specific message. All possible execution paths can be obtained by simulating the filter set for all possible messages. However, because there are an infinite amount of possible messages, this is not possible. Still, all possible execution paths can be determined by taking the following steps:

1. Equivalence classes of messages that have the same behavior in the filter set are identified by analyzing the matching parts of filters.
2. For each equivalence class a representative message is selected.
3. The filter set is simulated for each representative message.
4. The resulting state spaces are combined into one state space, representing all possible executions of the filter set.

Applications of Filter Reasoning

In [33] composition filter reasoning is performed for the following purposes:

- To detect consistency conflicts in the filter set, for example unreachable filters and matching parts that never match.
- To analyze interface changes: if composition filters are superimposed on objects, they can change the interface to these objects.
- To compile the filter set to GPL code, for efficient execution.

3.4.2 Extending Filter Reasoning to the Event Model

Event Simulation

The main approach for event simulation is equivalent to the approach of message simulation: the AST of the filter set is transformed to the corresponding flowchart and this flowchart is used to simulate the execution of a specific event. The only addition to the message simulation approach is the semantics of event property matching. For example, the semantics of the inequality matching of the property `margin` is added to the simulation algorithm, e.g., to simulate the execution of the matching expression `event.margin < 0.5`.

Constructing Event Equivalence Classes

The filter reasoning approach given in [33] analyzes all possible executions of a filter set by constructing equivalence classes of messages that have the same behavior in the filter set. To simulate the behavior of a filter set for all possible events in the event model introduced in Section 2.5, we need to construct equivalence classes of events. This section explains how such equivalence classes are created.

The first step to create the equivalence classes for events is to create the equivalence classes for each property of the events. The method to create the equivalence classes for a given property depends on the type of the property. Table 3.2 shows the types of the different event properties. The method to create the equivalence classes for each property type is explained next.

Property	Property type
eventType	Enumeration type
variableName	String type
value	Numeric type
returnValue	Numeric type
returnIdentifier	String type
values	String → Numeric type
margin	Numeric type
enforceReturn	Enumeration type

Table 3.2: Event property types

String type

Properties of this type contain `String` values. An example is the property `variableName`. Matching is performed using `String` equivalence comparison, e.g., `event.variableName='Tph'`. The `String` value, e.g., `'Tph'`, to which the value of the property is compared in a given matching part always gives different behavior in the filter set than other `String` values. The reason for this is that it matches in the given matching part, while other values do not match. We call such a `String` value a *distinguishable value*. As the behavior of this value in the filter set is different from the behavior of all other possible `String` values in the filter set, this value represents an equivalence class. The set of all distinguishable values for a given property p and filter set fs is represented by $Dist_p^{fs}$. There is one more equivalence class, which contains all values that are not applied in a matching part to compare the property's value against. As there is no matching part that matches these values, their behavior in the filter set is the same. We use the symbol ϵ as the representation for this equivalence class. The set of all equivalence class representatives for the given property p and filter set fs can be created as follows: $Repr_p^{fs} = Dist_p^{fs} \cup \{\epsilon\}$.

Enumeration type

Certain properties are of an `Enumeration` type, such as property `eventType`. The finite set of all possible values for the `Enumeration` type of property p is repre-

sented by T_p. Each value in T_p for which there is a matching part that selects based on this value forms an equivalence class, as its behavior is different from the behavior of the other values in T_p. Such a value is again called a *distinguishable value*. For example, if there is a matching part `event.eventType=='Change'`, then the behavior of the value `'Change'` for the property `eventType` is different from the behavior of all other possible values for the property `eventType`. Therefore, the value `'Change'` forms one equivalence class.
Besides the equivalence classes represented by all distinguishable values $Dist_p^{fs}$, there is one other equivalence class: the set containing all values of the `Enumeration` type for which there is no matching part that selects based on that value. This set is empty if all possible values of the `Enumeration` type are applied in a matching part. Thus, the set of all equivalence class representatives for the given property p and filter set fs is defined as follows:

$$Repr_p^{fs} = \begin{cases} Dist_p^{fs} \cup \{\epsilon\} & \text{if } Dist_p^{fs} \neq T_p, \\ T_p & \text{otherwise.} \end{cases}$$

Numeric type

For `Numeric` types, matching is performed by testing the equality or inequality of the property with a given value. This value is a *distinguishable value*. An example is testing of the margin property, for example $margin > 2$. In this example, 2 is a distinguishable value. To create the equivalence classes for `Numeric` types, we make no assumptions about the matching operators; this simplifies the construction of equivalence classes[6]. The equivalence classes for `Numeric` types are created in the following way:

- Create the ascendingly ordered list of all distinguishable values for the given property in the given filter set, with each value occurring only once in the list.
- Suppose this list contains the following n elements: $[el_1, el_2, \ldots, el_n]$. The set of equivalence classes (EC_p^{fs}) contains these elements and the intervals between each element, as follows[7]:

$$EC_p^{fs} = \{(-\infty, el_1), \{el_1\}, (el_1, el_2), \{el_2\}, \ldots, \{el_n\}, (el_n, \infty)\}$$

The reasoning behind this is that around a distinguishable value the behavior of the filter set differs, as matching is performed using the distinguishable value. Because we make no assumptions about the type of matching (e.g., equality, inequality, greater than), we need to assume that the set containing all values smaller than the distinguishable value, the set containing the distinguishable value and the set containing all values larger than the distinguishable value all have different behavior in the filter set, so all are equivalence classes. Because in general there are multiple of these distinguishable values, this creates

[6]However, taking the matching operator into account could reduce the number of equivalence classes, making filter reasoning more efficient. Therefore, taking the matching operators into account is considered to be future work.

[7]The standard interval notation as defined by ISO 80000-2 [66] is applied.

equivalence classes for each interval between each consecutive distinguishable value[8].

The set of representatives $Repr_p^{fs}$ contains all distinguishable values and one value from each interval. For example, if $EC_p^{fs} = \{(-\infty, 1), \{1\}, (1, 2), \{2\}, (2, \infty)\}$, then one possible $Repr_p^{fs}$ is: $\{0.5, 1, 1.5, 2, 2.5\}$.

String → Numeric type

The property `values` is a mapping of `String` values to `Numeric` values. Matching is performed by testing the equality or inequality of the `values` property, indexed with a certain `String` key, with a given value. An example of a matching statement is `event.values['sensor'] < 1.5`, in which the key is $'sensor'$ and the value is 1.5. The set of equivalence classes for the `values` property is created as follows.

- First, a list containing all distinguishable keys in the filter set is constructed. This list, containing n elements, is represented as: $Keys = [key_1, key_2, \ldots, key_n]$.
- For each distinguishable key key_i, the distinguishable values are identified. These are the values in matching parts that match against the property $event.values[key_i]$. From this set of distinguishable values, the set of equivalence class representatives is created in the same way as with a `Numeric` type. The set of representatives for key_i is represented by $Repr_{values[key_i]}^{fs}$.
- The set of representatives for the equivalence classes for the `values` property is constructed using the following equation:

$$Repr_{values}^{fs} = \{\{(key_1, val_1), (key_2, val_2), \ldots, (key_n, val_2)\}| \\ \text{for } i = 1 \ldots n\text{: } val_i \in Repr_{values[key_i]}^{fs}\}$$

For example, if $Keys = [a, b]$, $Repr_{values[a]}^{fs} = \{1, 2\}$ and $Repr_{values[b]}^{fs} = \{5, 6\}$, then:

$$Repr_{values}^{fs} = \{\{(a, 1), (b, 5)\}, \{(a, 1), (b, 6)\}, \{(a, 2), (b, 5)\}, \{(a, 2), (b, 6)\}\}$$

Constructing the Event Equivalence Classes

As the event is defined by the valuation of its properties, the set of equivalence classes for events is the Cartesian product of the sets of equivalence classes for all properties. The set of representative values for the event equivalence classes can therefore be created by taking the Cartesian product of the sets of representative values for all properties:

[8]Note that this actually creates pseudo-equivalence classes, which means that there might be two equivalence classes that result in the same behavior of the filter set. This is caused by the fact that we do not take the type of matching into account. The disadvantage of pseudo-equivalence classes is that it results in redundant states in the generated state space by the filter reasoning procedure. Redundant states have no effect on the outcome of filter reasoning; they only make filter reasoning less efficient. It is possible to take the type of matching into account, but this results in a less straightforward way to create the equivalence classes.

$$Repr^{fs} = \prod_{p \in properties} Repr_p^{fs}$$

For each element in $Repr^{fs}$ the filter set is simulated. The resulting state spaces are combined to create the state space with all possible executions.

Example 3.3 Event Equivalence Classes

Listing 3.8 shows an example filter set. We will construct the set of equivalence classes for this filter set.

```
1  tbeltHandler: Result = (event.variableName=='Tbelt' &
       event.eventType=='Inconsistency')
       {event.resultValue=event.values['Tbelt_sensor']};
2  tbeltLog: Logging = (event.variableName=='Tbelt' &
       event.eventType=='Inconsistency' & event.margin > 0.2);
3  tContactChange : Dispatch = (event.variableName=='Tcontact' &
       event.eventType=='Change') {target=radControl;
       selector='setTcontact';};
```

Listing 3.8: Composition filters specification to log inconsistencies

Property: variableName The only values applied in the filter set for the property `variableName` are $'Tbelt'$ and $'Tcontact'$. Therefore, the set of representative values for the equivalence classes of the property `variableName` is as follows:

$$Repr^{fs}_{variableName} = \{'Tbelt', 'Tcontact', \epsilon\}$$

For the property `eventType`, the values applied in the filter set are $'Inconsistency'$ and $'Change'$. Therefore, the set of representative values is:

$$Repr^{fs}_{eventType} = \{'Inconsistency', 'Change', \epsilon\}$$

For the property `margin`, the only value applied in the filter set is 0.2. The set of equivalence classes is:

$$EC^{fs}_{margin} = \{(-\infty, 0.2), \{0.2\}, (0.2, \infty)\}$$

The set of the following representative values is derived:

$$Repr^{fs}_{margin} = \{0.1, 0.2, 0.3\}$$

The set of representatives for the event equivalence classes, containing 27 elements, is as follows:

$$\begin{aligned} Repr^{fs} &= Repr^{fs}_{variableName} \times Repr^{fs}_{eventType} \times Repr^{fs}_{margin} \\ &= \{('Tbelt', 'Inconsistency', 0.1), \\ &('Tbelt', 'Inconsistency', 0.2), \ldots, \\ &(\epsilon, \epsilon, 0.3)\} \end{aligned}$$

The other properties in the event model are assumed to have a default value and are not shown in the representatives for the event equivalence classes, as they are not used in a matching expression of the filter set.

3.4.3 Applications of Filter Reasoning

Filter reasoning has a number of applications. In this section we summarize some important applications related to the composition of physical models.

Detect Consistency Conflicts in the Filter Set

A first application of filter reasoning is to check the consistency of the filter set. For example, there can be unreachable filters in the filter set, filters that never accept, etc. Using the state space that contains all possible executions of the filter set, these conflicts can be detected and their cause can be analyzed [33].

Example 3.4 Consistency Conflicts

Listing 3.8 in Example 3.3 contains a consistency conflict: the second filter never accepts. This conflict is detected using filter reasoning in the following way. Filter reasoning detects that the first filter always accepts for the following set of representative events:

$$Repr^{accept1} = \{'Tbelt'\} \times \{'Inconsistency'\} \times Repr^{fs}_{margin}$$

When the first filter accepts, it returns the *execution flow*, which means that the second and third filter are not executed anymore. This means that for all events in the equivalence classes represented by $Repr^{accept1}$, the second and third filter are not reachable.

The second filter accepts the following set of representative events:

$$Repr^{accept2} = \{'Tbelt'\} \times \{'Inconsistency'\} \times \{0.3\}$$

But, as $Repr^{accept2} \subset Repr^{accept1}$, such events never reach the second filter. Therefore, the second filter never accepts. Note that the third filter does not have this problem, as the set of messages accepted by that filter is not a subset of the set of messages accepted by the first filter. So, some of the messages that the third filter accepts can still reach the third filter.

Detect Redundancy in the Physical Model

There can be multiple ways to determine the value of a certain physical variable in a physical model. One source of values for physical variables is from the filter set execution of a *CheckUpdate* event. Using filter reasoning it can be determined for which physical variables a *CheckUpdate* event results in a value being returned. This

information is used to determine which physical variables have multiple sources for their values.

Analysis of Inconsistency Event Matching

For physical variables that do not have multiple sources for their values, it does not make sense to do *Inconsistency* event matching, as single values cannot be inconsistent. To detect whether *Inconsistency* event matching is performed for such physical variables, one could suggest performing straightforward analysis of the matching parts in the filter set. This means checking whether there are matching parts that have the selection statements `event.variableName=='v' & event.eventType=='Inconsistency'` for all variables v that do not have multiple sources for their values. But this basic analysis is not sufficient, as composition filters have the capability to change the values of the event properties. Some filter might change the `variableName` property of the event, which corrupts the results of the basic analysis.

Filter reasoning can be applied to perform this type of analysis correctly, as the simulation of the filter set takes changes of the values of properties into account. To perform this analysis, the following subset of all representative events is taken: the subset that only contains those events for which the `variableName` property is a physical variable with a single source for its value. The filter set is simulated for all events in this subset. The resulting state space can be checked for whether there exists an `event.eventType=='Inconsistency'` check that matches. If this is the case, the property that no inconsistency matching is performed for variables that do not have multiple sources for their values is violated.

Further Diagnosis of Failures in the Physical Model

Section 3.3 discussed the runtime verification of physical models. Diagnosis of encountered failures was discussed in Section 3.3.2. This diagnosis can lead to the suspicion of an incorrect *request value* (i.e., a value that is provided by the composition filters after a *CheckUpdate* event). In this case, the filter set can be analyzed to determine the origin of the value, e.g., a call to a sensor component. Filter reasoning can be applied to simulate the behavior of the filter set for the given event and detect the single origin or the multiple possible origins of the value.

Efficient Execution of Composition Filters

The current implementation of our approach makes use of the Compose* interpreter to execute defined composition filters when an event occurs in the physical model instance. Such an interpreter based approach provides the flexibility to experiment with the implementation, but it does not optimize for efficiency. Instead, it introduces a runtime performance overhead. To apply the approach in industry, more efficient execution is necessary.

More efficient execution of composition filters can be obtained by compiling the filter set to GPL code [33]. The compilation can be performed in three main steps:

- Locate the artifacts (e.g., classes, physical model instances) on which composition filters are superimposed.

- For each of these artifacts, find the locations in their code at which a message is sent or an event occurs (in aspect-oriented programming terminology, these are called *join-point shadows* [60]).
- For each of these locations, apply filter reasoning on the filter set with the message or event corresponding to the code location. The result of this filter reasoning is the specific behavior of the filter set for the given message or event. Code can be generated that executes this specific behavior and this code can be woven at the specific location in the artifact.

In our approach, the composition filters are superimposed on physical model instances. Code generation for the composition filters should be performed together with code generation for the physical model instance, as the generated GPL code with filter behavior should be *woven* into the generated GPL code of the physical model instance.

3.5 Applying Inconsistency Monitoring for Calibration and Broken Sensor or Component Detection

Inconsistencies in multiple values of a physical variable do not always indicate a failure. This section shows an example in which inconsistency monitoring is used to calibrate the physical state in a physical model instance. However, precautions are taken to cope with a broken sensor, a scenario that can happen in practice.

3.5.1 Calibration

In the Drum Shuttling case study introduced in Section 1.3.2, the z-position of the drum (*zPos*) is derived from the step position of the stepper motor (*stepPos*). This step position is updated if a step is actuated. However, the stepper motor occasionally fails to perform a step when it is actuated, creating a deviation between the physical model and physical reality. Small deviations do not cause a problem. However, when the physical model is not corrected once in a while, the errors may accumulate into larger deviations. To cope with this problem, there is a calibration sensor in the system, called *home sensor*, that is triggered when the drum reaches a specific z-position. Example 3.5 shows how the home sensor is used to calibrate the physical model instance.

Example 3.5 Calibration of a Physical Model Instance

Software Design

Example 2.12 introduced the software design of the Drum Shuttling case study. To incorporate the home sensor, we extended the design by adding a second instance of the `stepperMotor,cam` physical model. This second instance is labeled `Shuttling2`, as

shown in Figure 3.2. The SIDOPS+ specifications of `stepperMotor` and `cam` were given in Listings 2.16 and 2.17 in Example 2.12.

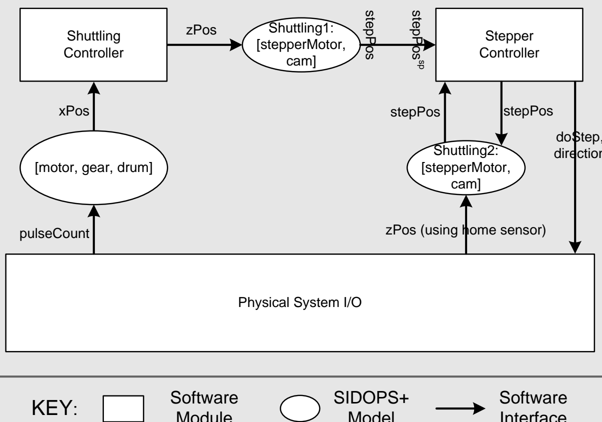


Figure 3.2: Drum Shuttling software structure and data flow

Composition definition

In the `Shuttling2` physical model instance, the value of the physical variable *stepPos* is set by the `StepperController` module after each performed step, using the base interface of the physical model instance.

Listing 3.9 shows the composition filters specification to calibrate `Shuttling2`.

```
1  filtermodule zPosRequestHandler{
2    externals
3      homeSensor: Sensor = IO.getShuttlingHomeSensor();
4    conditions
5      isActive: homeSensor.isActive();
6    outputfilters
7      homingResult: Result = (event.variableName=='zPos' &
           event.eventType=='CheckUpdate' & isActive)
           {event.resultValue=C_homePosition;
           event.resultIdentifier='homeSensor';};
8  }
9
10 filtermodule zPosMonitor{
11   outputfilters
12     inconHandler: Result = (event.variableName=='zPos' &
           event.eventType=='Inconsistency') {
           event.resultValue=event.values['homeSensor'];
           event.enforceResult='true' };
13 }
14
15 filtermodule zPosLogging{
16   outputfilters
17     zPosLog: Logging = (event.variableName=='zPos' &
           event.eventType=='Inconsistency' & event.margin > 2);
```

```
18 }
19
20 superimposition{
21   selectors
22     models = { M | isModelInstance (M, [Shuttling2]) };
23   filtermodules
24     models <- zPosRequestHandler, zPosLogging, zPosMonitor;
25 }
```

Listing 3.9: Composition filters specification to handle home sensor for calibration

There are three filter modules. The `zPosRequestHandler` filter module handles `request` events for *zPos*. But it only accepts if the condition `isActive` is `true`. This condition is defined to check whether the home sensor is active. If the filter matches, the result of the `request` event is set to home position (which is defined as the constant `C_homePosition`).

The `zPosMonitor` filter module implements the actual calibration. It monitors for inconsistency events in *zPos* and selects the value that is provided by the `homingResult` filter (i.e., the home position). It also enforces this result upon the entire physical state, by setting the `enforceResult` property to `true` (see Section 2.5 for more information about the `enforceResult` property).

The `zPosLogging` filter module performs logging of the inconsistency if the deviation is larger than two, for diagnosing the problem later on.

Lines 20 until 25 show how the two filter modules are superimposed on the module `ShuttlingModel`.

3.5.2 Broken Sensor or Stepper Motor Detection

The previous example showed how the home sensor is used to calibrate the physical model instance. This works fine when both the home sensor and stepper motor work properly. But if one of them is broken, this leads to incorrect results, which may even damage the physical system. For example, when the sensor is broken and always gives a signal, the physical model instance will be continuously calibrated to the home position. This continuous calibration means that *stepPos* remains the same. The value of $stepPos^{sp}$, as indirectly requested by the `ShuttlingController`, will continue to move further away from the home position. So, the difference between $stepPos^{sp}$ and *stepPos* in the model increases, and this difference is larger than in reality, as in reality *stepPos* does change and as such is closer to $stepPos^{sp}$. That the calculated difference between *stepPos* and $stepPos^{sp}$ is larger than in reality means that the `StepperController` will perform more steps to bring *stepPos* to $stepPos^{sp}$ than actually needed. Eventually, the stepper motor will be controlled in such a way that the *zPos* of the drum will go outside its physical boundaries, which possibly results in damage of the physical system. Using proper runtime verification, such problems can be prevented. In this section we show how the detection of a malfunctioning sensor or malfunctioning stepper motor can be implemented using the techniques presented in this chapter. A malfunctioning sensor can give two results: either the sensor never gives a signal or the sensor always gives a signal. Example 3.6 shows how a sensor that

never gives a signal can be detected. Example 3.7 shows how a sensor that always gives a signal can be detected.

Example 3.6 Detection of Never Active Sensor

Detection of a sensor that is never active is performed as follows. When the drum is supposed to be at the home position ($zPos == C_{homePosition}$), and the sensor is not active, then detection starts. If the sensor is working properly, an active signal will arive within a limited number of steps. But if the sensor is broken, an active signal will never arive. Therefore, if the $zPos$ has changed outside certain boundaries around $C_{homePosition}$, and still there is no signal, the sensor is assumed to be broken. These boundaries are represented by the constants $C_{checkLowBound}$ and $C_{checkUpBound}$. If $zPos$ is still within the boundaries and a signal of the sensor does arrive, checking for an inactive sensor stops. Note that when an always inactive sensor is detected, this means that either the sensor is broken and gives no signal or that the stepper motor is broken and the drum never reaches the home position. Listing 3.10 shows the specification of a filter module that detects inactive sensors.

```
1  filtermodule InactiveSensorFM{
2    externals
3      homeSensor: Sensor = IO.getShuttlingHomeSensor();
4    internals
5      detector: Detector;
6    conditions
7      isActive: homeSensor.isActive();
8      checkInactive: detector.checkInactive();
9    outputfilters
10     startCheck: Before = (event.variableName=='zPos' &
           event.eventType=='Change' & event.value==C_homePosition &
           !isActive) {filter.target=detector;
           filter.selector='startCheck';};
11     errorDetect: Before = (event.variableName=='zPos' &
           event.eventType=='Change' & (event.value<C_checkLowBound |
           event.value >C_checkUpBound) & checkInactive & !isActive)
           {filter.target=detector; filter.selector='sensorNoSignal';};
12     stopCheck: Before = (event.variableName=='zPos' &
           event.eventType=='Change' & isActive)
           {filter.target=detector; filter.selector='stopCheck';}
13 }
```

Listing 3.10: Composition filters specification to detect inactive sensor

The filter module uses an `internal` object of class `Detector`, to maintain state about the detection. There are three filters. The filter `startCheck` starts the detection of an inactive sensor. It matches when a change of $zPos$ to $C_{homePosition}$ occurs and the sensor is not active. When the filter matches, it performs a call to `startCheck` in `Detector` to set the condition `checkInactive` to *true*. The filter `errorDetect` detects that $zPos$ is outside the boundaries and we are still waiting for a signal. This filter executes error code in `Detector`. The filter `stopCheck` detects that the sensor is active and stops the checking by calling `stopCheck` in `Detector`. This call sets the condition `checkInactive` to $false$.

Example 3.7 Detection of Always Active Sensor

Detection of a sensor that is always active is performed as follows. The `StepperController` actuates the stepper motor to make steps. If everything works properly, then if the drum is at the home position and a step is made, the active sensor has to become inactive. The only exception to this property is when the stepper motor accidentally misses a step; in this case the sensor is still active as the drum is still in home position. But if multiple steps are made, for example more than 4, and the sensor is still active, this is assumed not to be caused by the expected behavior of the stepper motor to miss a step once in a while, but by a broken sensor that always gives a signal. Note that a sensor that is always active may also indicate a broken stepper motor, while the drum is exactly at the home position. Listing 3.11 shows the specification of a filter module that detects always active sensors.

```
1  filtermodule ActiveSensorFM{
2    externals
3      homeSensor: Sensor = IO.getShuttlingHomeSensor();
4    internals
5      detector: Detector;
6    conditions
7      isActive: homeSensor.isActive();
8      checkActive: detector.checkActive();
9    outputfilters
10     doCheck: Before = (event.variableName=='stepPos' &
           event.eventType=='Update' & isActive)
           {filter.target=detector; filter.selector='activeDetect';};
11     resetCheck: Before = (event.variableName=='stepPos' &
           event.eventType=='Update' & checkActive & !isActive)
           {filter.target=detector; filter.selector='inactiveDetect';}
12 }
```

Listing 3.11: Composition filters specification to detect inactive sensor

There are two filters. The filter `doCheck` matches when *stepPos* is updated and the sensor is active. The filter action is a call to method `activeDetect` in class `Detector`. This method increments a counter that indicates how many times after each other a step is made while the sensor remains active. The method also implements the check to detect that the counter becomes to large and the method implements the handling of the detected failure. The condition `checkActive` is *true* when the counter is larger than 0, otherwise the condition is $false$. The filter `resetCheck` matches when *stepPos* is updated and the sensor is not active. A call to the method `inactiveDetect` is performed to reset the counter in `Detector` to 0.

3.5.3 Benefits of the Composition Filters Model

The examples in this section clearly show the benefits of the Composition Filters model in our approach; the additional functionality of calibration of the physical model instance and detection of a broken sensor or component can be added to the software modularly, without having to modify or extend the SIDOPS+ specification of the physical model or existing sotware modules.

3.6 Discussion

This section discusses some additional subjects related to our approach and the implementation of our approach.

3.6.1 Efficiency of the Monitoring Approach

Applying runtime verification means that additional behavior is executed, leading to a certain performance overhead. We used an interpreter based implementation for the evaluation of the SIDOPS+ specifications and the execution of the composition filters for monitoring. Such an interpreter based approach introduces considerable runtime overhead. However, the aim of the interpreter based implementation is to experiment with and demonstrate our approach, not to provide an efficient runtime environment. Efficient compilation algorithms for aspect-oriented languages, such as the Composition Filters model, are known in literature, e.g., in [21, 33].

3.6.2 Moment of Checking

A fault or inconsistency in a physical model can lead to a failure when the physical model is evaluated and the results of this evaluation are used by GPL modules. Since we want to monitor whether there are inconsistencies in the physical model that can lead to failures of the system, the best time to do monitoring is after the physical model has been evaluated but before the result is used. In this way, the results of the evaluation can be used by the monitoring code, reducing the overhead of performing additional calculations. Furthermore, action can be taken before erroneous results of the evaluation are used by GPL modules and lead to failures in the system. This moment of checking is enforced by the evaluation algorithm, as explained in Sections 2.4.5 and 2.5.5: first the inconsistencies are resolved before *Change* events are communicated.

One could argue that monitoring can also be performed at other time instances, for example to diagnose problems in an earlier stage. But this may lead to complications, as the physical relationships in the physical model may be designed to be consistent only when the physical model instance is evaluated. Furthermore, performing monitoring at other time instances reduces performance of the system, as additional calculation and solving of the physical relationships in the physical model needs to be performed, to derive values when the physical model instance is not evaluated.

3.6.3 Recovery Actions

When an inconsistency has been detected, a recovery action can be taken. Depending on how the engineer perceives the severity of the inconsistency, there are several options, which include:

- Stop the operation of the system, for safety-critical operations.
- Select one of the calculated values, e.g., randomly, based on a voting scheme or based on a preference.

- Log the inconsistency for diagnosis.

In some cases, the system can also be reconfigured at runtime if it is necessary to recover from an error. For example, Babty et al. [15] facilitate dynamic reconfiguration of systems for error recovery.

3.7 Related Work

Runtime Verification

Literature shows, e.g., in the taxonomy of Delgado et al. [37] and in the work of Barringer et al. [16], that the common approach for runtime verification is first to create a data and/or event model in which the software can be described. Next, several properties that are specified in a certain logic, such as temporal logic or regular expressions, are verified for the created model. Examples of such runtime verification approaches are the MOP framework [25] and the tracematches extension to AspectJ [13].

The MOP framework is an generic framework for runtime verification, providing a mechanism to add different types of logics. The MOP framework is able to generate monitoring code based on the specification of properties that need to be checked. The Java implementation of the MOP framework generates monitoring code in AspectJ [25]. The tracematches extension to AspectJ enables the monitoring of regular patterns of events in a program and the execution of certain code when such a pattern has been detected [13].

Such approaches cannot be applied in our case, as we focus on the impact of the software behavior on the behavior of the physical system; failures only become apparent in the behavior of the physical system. Therefore, our approach uses redundant models of physical relationships to verify their conformance with physical reality. Like other runtime verification approaches, our approach applies aspect-oriented techniques to specify monitors.

Fault Diagnosis

Fault diagnosis aims at determining the health state of the system or components in the system, by analyzing the output of the system given a certain input. There are two approaches to diagnose the location of faults in components. Model-based diagnosis, as introduced by Reiter [93] and De Kleer [32], uses a model of the system to diagnose the failing component based on the system's input and output. Spectrum-based fault localization is a statistical approach that diagnoses failing components by correlating failures in the output with execution traces [115]. Van Gemund et al. combined both approaches to be applied on the combination of embedded system and the corresponding embedded software [9, 47].

Our approach supports the model-based diagnosis approach by Reiter and De Kleer in the context of physical models; we can define a physical model primarily aimed at determining values of physical variables that are checked with sensor values, to detect inconsistencies in the sensor values. A difference with the model-based diagnosis approach is that our approach does not only apply models to verify sensor outcomes, but also uses sensor outcomes to verify the physical models and our

approach uses the physical models to implement behavior in software. In the last case, runtime verification of inconsistencies is an additional benefit. Furthermore, our approach is not only able to check outcomes of the model with sensor values, but can also check whether redundant physical relationships in the physical model are consistent.

Composition Analysis

Physical models are composed with GPL modules using the aspect-oriented Composition Filters model. This chapter explained how a set of composition filters can be analyzed. Krishnamurthi et al. describe in [77] how aspects can be verified modularly, i.e. independent from the base program. They try to verify that introduced aspects do not invalidate certain enforced properties of the base program. To do this, they use a state machine representation of the aspects. Related to this is work by Goldman and Katz, who introduce in [54] a technique to verify modularly that an aspect satisfies certain properties if the base system satisfies certain assumptions of the aspect. The state space generated with filter reasoning can be used as the state transition diagram used in these verification techniques. Furthermore, the consistency analysis of composition filters can also be performed modularly, without knowledge of the base system (in this case, the physical model instance and GPL modules).

Our filter reasoning approach is able to detect certain consistency conflicts, for example caused by an incorrect ordering of the composition filters. This is one type of conflict that can occur in aspect-oriented programming languages. Different types of conflicts have been investigated in literature. Störzer and Krinke show in [99] how binding interferences in AspectJ, caused by the introduction and hierarchy modification features of AspectJ, can be analyzed. Kniesel describes how to detect and resolve weaving interactions in [74]. Havinga et al. introduced techniques to detect composition conflicts that are caused by introductions [57].

Dürr investigated how to detect behavioral conflicts among aspects [40]. Behavioral conflicts are conflicts in which the behavior of an aspect interferes with the behavior of other aspects, resulting in unexpected behavior. Dürr's approach for behavioral reasoning makes use of a resource-operation model. He applied this approach to the Composition Filters model, using filter reasoning as the basis for the analysis.

3.8 Conclusion

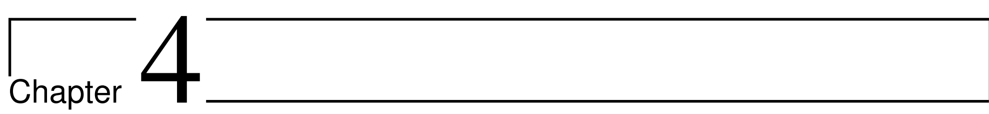
Models of physical characteristics, i.e., physical models, become part of control software in embedded systems. Such models are used for estimating physical relationships among system components, and influencing the control behavior accordingly. As such, faults that are undetected in physical models can lead to failures in control behavior. Therefore, it is important to monitor and verify the accuracy of these physical models at runtime. Furthermore, static analysis of the composition of physical models with GPL modules is necessary, e.g., to verify the consistency of the composition and to detect redundancy in the physical model.

The focus of traditional runtime verification techniques is on verifying whether software behavior corresponds to a separately specified model of correct behavior.

Our aim is to verify physical models used in software for their correspondence with physical reality. As such, traditional runtime verification techniques cannot be applied to verify physical models.

This chapter introduced a novel technique to verify physical models at runtime using redundancy in the physical model. This technique can, for example, be used to detect inconsistencies in the physical model and to monitor for wear and tear in the physical system. Using a derivation graph of the physical model, detected inconsistencies can be diagnosed. This diagnosis leads, for example, to the detection of incorrect physical relationships in the physical model or of malfunctioning components in the physical system.

This chapter also introduced a technique to statically analyze the composition filters that compose physical models with GPL modules. This static analysis is called filter reasoning. Filter reasoning applies simulation of the filter set for specific messages/events to create a state space of all possible behaviors of the filter set. This resulting state space has several applications, e.g., to detect inconsistencies in the filter set, to detect redundancy in the physical model, to further diagnose the cause of detected failures or inconsistencies in the physical model and to compile the filter set to code.



The MO2 Method for Runtime Optimization of Multiple System Qualities in Embedded Control Software[1]

4.1 Introduction

A current trend in embedded systems is towards adaptivity under varying circumstances (e.g., environmental conditions, user needs and input). For example, in high-end printing systems adaptive optimization of multiple *system qualities*, such as productivity and energy consumption, is performed. Trade-off decisions between conflicting objectives are made based on user needs. Adaptive optimization results in more competitive systems that have better specifications and that better satisfy customer needs while hardware costs are kept low.

To optimize the system behavior with respect to system qualities, the right values for certain controllable variables (the *decision variables*) have to be chosen. Examples of decision variables are the speed of the system and the temperature setpoint of a heating device. The values of the decision variables are often subject to constraints. For example, the speed of the system is limited to a maximum speed and the amount of power the system can consume is limited to the amount of power available. The problem of multiple system qualities (or system objectives) to be optimized given a set of decision variables that can be influenced, but that are subject to constraints is known as multi-objective optimization (MOO) [69].

In modern embedded systems, the control logic is usually implemented in software. In current design practice, the control logic implemented in *embedded control software* is decomposed into many different controllers, each controlling a part of the system. Such a decomposition is made as it makes the control logic easier to comprehend and maintain, as opposed to, for example, a single (black box) controller that controls all variables. As the software decomposition usually follows the control decomposition, the different controllers are implemented in several software modules in the embedded control software. Certain system qualities, such as power consumption and produc-

[1] An early version of this chapter has been published in the 8th Working IEEE/IFIP Conference on Software Architecture, WICSA 2009 [35].

tivity, are not controlled by a specific controller, but emerge from the behavior of the system as a whole. So, if we want to influence these system qualities, we have to manipulate and coordinate many controllers, scattered through the embedded control software. This manipulation and coordination of controllers introduces additional structural complexity within the embedded control software.

There is a lack of systematic methods to design and implement multi-objective optimization of system qualities in embedded control software. We have observed in practice that attempts to realize multi-objective optimization leads to:

- Solutions that are tailored to the specific characteristics of the embedded system, and therefore inflexible in case the physical system changes or evolves.
- Solutions that are tightly integrated into and coupled with the control software modules, making the control software difficult to comprehend and hard to maintain.
- Solutions that are sufficient, but not optimal, as better optimizing solutions would be too complex to implement.

As such, the lack of systematic methods to design and implement multi-objective optimization in embedded control software leads to higher development and maintenance costs [14] and possibly to functionality that is not optimal. Alternatively, the lack of systematic methods might lead to the decision not to implement multi-objective optimization, as the benefits of a more optimized system do not outweigh the reduced software quality and increased development and maintenance costs.

The contribution of this chapter is a systematic method, called the *MO2 method*, to design and include multi-objective optimization and dynamic trade-off making in embedded control software. The MO2 method includes an architectural style to specify and document a multi-objective optimization solution within the architecture of the embedded control software. The architectural style is supported by techniques, implemented in a toolchain, to validate the consistency of the solution, include general optimization algorithms and generate code that implements the optimization algorithm and coordination of the control modules.

This chapter is organized as follows. First, background on multi-objective optimization is provided. Next, the problem is explained. This is followed by an overview of the MO2 method and the presentation of its elements: the MO2 architectural style and the MO2 toolchain. Then, two advanced applications of the MO2 architectural style are presented, demonstrating that the MO2 method supports hierarchical application of the MO2 style and optimization over time (using model-predictive control). The chapter ends with a discussion, the presentation of related work and a conclusion.

4.2 Background: Multi-Objective Optimization

A multi-objective optimization problem is a mathematical problem in which the goal is to optimize multiple objectives. The objectives can be influenced by a set of decision variables. The values of these decision variables are subject to constraints. This

type of problem was first introduced in works on decision making in economy by Edgeworth [42] in the late nineteenth century. Pareto extended the work with the concept of Pareto optimality [91]. This section introduces the basic terminology of multi-objective optimization. First, some terminology is introduced that is related to describing a multi-objective optimization problem. Then, the concepts related to optimization with multiple objectives are explained. At the end of each subsection, the concepts are illustrated with an example. For more information on this subject, we refer to [28, 69].

4.2.1 Problem Description

A multi-objective optimization problem is defined as follows [28]:

Definition 4.2.1 (Multi-Objective Optimization Problem) *A multi-objective optimization problem can be represented by a tuple* $\langle \boldsymbol{o}(\boldsymbol{x}), \boldsymbol{g}(\boldsymbol{x}), \boldsymbol{h}(\boldsymbol{x}) \rangle$*, where*

- $\boldsymbol{x} \in \mathbb{R}^n$ *represents the value of the* n *decision variables.*
- $\boldsymbol{o}(\boldsymbol{x}) \in \mathbb{R}^n \rightarrow \mathbb{R}^k$ *is a vector of* k *functions in the decision variables* $\boldsymbol{x}$ *that provide a valuation for the* k *objectives. These functions are referred to as* objective functions*.*
- $\boldsymbol{g}(\boldsymbol{x}) \in \mathbb{R}^n \rightarrow \mathbb{R}^l$ *is a vector of* l *functions that describe inequality constraints. An inequality constraint means that* $\boldsymbol{x}$ *should be choosen in such a way that the outcome of each of the* l *functions in* $\boldsymbol{g}$ *is smaller than* 0*.*
- $\boldsymbol{h}(\boldsymbol{x}) \in \mathbb{R}^n \rightarrow \mathbb{R}^m$ *is a vector of* m *functions that describe equality constraints. An equality constraint means that* $\boldsymbol{x}$ *should be choosen in such a way that the outcome of each of the* l *functions in* $\boldsymbol{g}$ *is equal to* 0*.*

The solution to this problem are the $\boldsymbol{x} \in \mathbb{R}^n$ *that minimize* $\boldsymbol{o}(\boldsymbol{x})$ *under the constraints* $\forall i \in [1 \ldots l] g_i(\boldsymbol{x}) \leq 0$ *and* $\forall j \in [1 \ldots m] h_j(\boldsymbol{x}) = 0$*.*

Note that in this definition the minimization of a vector of objective functions is not clearly defined, as there is no clear *smaller than* relation defined on vectors. One could assume the following definition:

$$\boldsymbol{o}(\boldsymbol{x_1}) \leq \boldsymbol{o}(\boldsymbol{x_2}) \leftrightarrow \forall i \in [1 \ldots k] o_i(\boldsymbol{x_1}) \leq o_i(\boldsymbol{x_2})$$

But this only gives a *partial ordering*, which means that there may not be a single outcome of the objective functions that is smaller than all other outcomes. Pareto solved this problem by defining the concept of Pareto optimality [91]. This concept will be explained later in this section. Example 4.1 on Page 120 presents an example multi-objective optimization problem.

The constraints limit the possible values for the decision variables to a subset of $\mathbb{R}^n$. This subset is called the *feasible decision space*. The solution to the optimization problem is selected from the feasible decision space.

Definition 4.2.2 (Feasible decision space) *The subset of $\mathbb{R}^n$ containing only those values $\boldsymbol{x}$ that satisfy the constraints. This subset is also referred to as Dspace, and is formally defined as:*

$$Dspace = \{\boldsymbol{x} \in \mathbb{R}^n | \forall i \in [1 \ldots l] g_i(\boldsymbol{x}) \leq 0 \wedge \forall j \in [1 \ldots m] h_j(\boldsymbol{x}) = 0\}$$

Figure 4.1 in Example 4.1 shows an example of a feasible decision space.

Each value $\boldsymbol{x}$ in the feasible decision space can be input to the objective functions $\boldsymbol{o}(\boldsymbol{x})$. The set of all objective values that result from applying all values in the feasible decision space to the objective functions is called *objective space* or *criterion space.*

Definition 4.2.3 (Objective Space) *The set of values that are the result of applying the objective functions $\boldsymbol{o}(\boldsymbol{x})$ to all elements in Dspace:*

$$Ospace = \{\boldsymbol{o}(\boldsymbol{x}) | \boldsymbol{x} \in Dspace\}$$

Figure 4.2 shows an example objective space.

Example 4.1 Multi-Objective Optimization Example

The following is an example specification of a multi-objective optimization problem, containing two decision variables, three constraints and two objective functions.

- **Decision Variables:** x_1, x_2
- **Constraints:**
  - $x_2 - \frac{1}{10}x_1^2 <= 0$
  - $1 - x_2 <= 0$
  - $x_1 - 8 <= 0$
- **Objective functions:**
  - $o_1(\boldsymbol{x}) = 70 - x_1^2 + x_2^2$
  - $o_2(\boldsymbol{x}) = 5.5 - \frac{1}{10}x_1 x_2$

The three constraints limit the decision space to a *feasible decision space.* Figure 4.1 shows the three constraints in the decision space of the two decision variables. The marked area is the feasible decision space. In this example, the value of each decision variable can be selected with a precision of 0.1 (In the domain of embedded systems it is common that values of controllable variables are discretized).

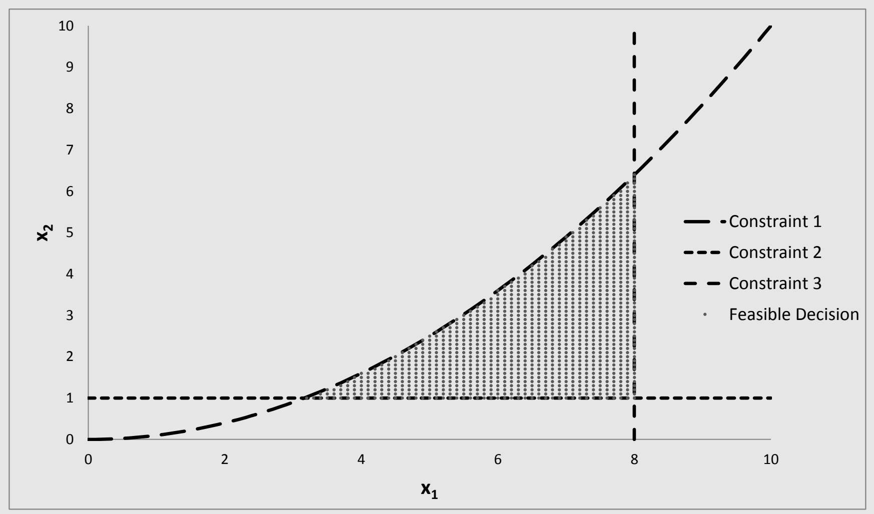


Figure 4.1: The feasible decision space

The feasible decision space contains all values for $\boldsymbol{x}$ that may be selected. Each point in the feasible decision space has a corresponding outcome of the objective functions. Figure 4.2 plots for each point $\boldsymbol{x}^a$ in the feasible decision space the corresponding outcome of the objective functions ($\boldsymbol{o}(\boldsymbol{x}^a)$), resulting in the *objective space*.

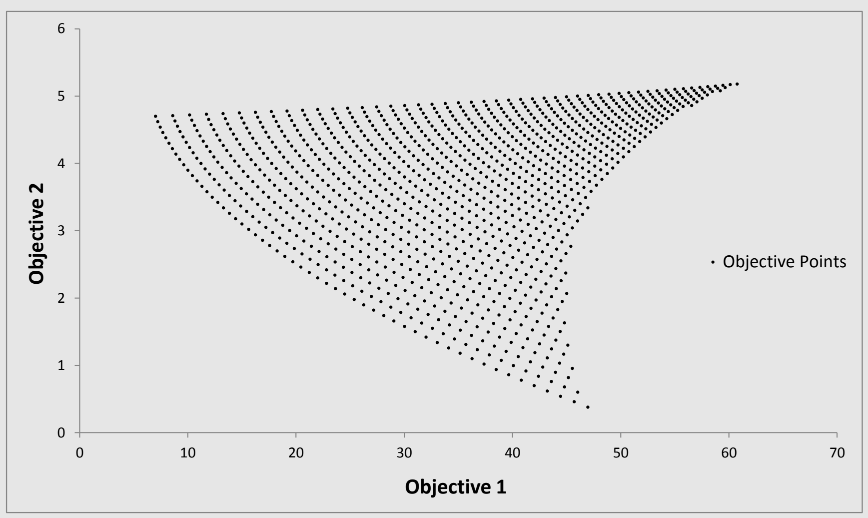


Figure 4.2: The objective space

4.2.2 Optimization

Optimizing a single objective is straightforward, as there is a one-dimensional objective space in which there is one point that has a smaller objective value than all other points in the objective space. But when there are multiple objectives, there is usually not a single point in the objective space that has the minimal value for all objectives. In this case, there is another definition of optimality, called *Pareto optimality*, which was introduced by Pareto [91]. This subsection describes the concepts of *dominated points*, *Pareto optimal points* and *Pareto frontier*.

One point in the objective space is said to dominate another point in the objective space if it improves upon at least one of the objectives, without compromising on the other objectives. A point that dominates another point is considered to be better.

Definition 4.2.4 (Dominated point) *A point $\boldsymbol{o}^a \in Ospace$ is dominated by a point $\boldsymbol{o}^b \in Ospace$ iff $\boldsymbol{o}^b$ has a better value for one of the objectives, while not compromising the other objectives (i.e., the values for the other objectives are not worse). Formally:*

$$Dominates(\boldsymbol{o}^b, \boldsymbol{o}^a) \leftrightarrow \exists i \in [1 \dots k] o_i^b < o_i^a \wedge \forall j \in [1 \dots k] o_j^b \leq o_j^a$$

The Dominates relation provides a strict partial ordering *of Ospace.*

In a multi-objective optimization problem, there might not be a single point that dominates all other points. Instead, there can be multiple points in $Ospace$ that are not dominated by any other point in $Ospace$. These points are called *Pareto optimal points.*

Definition 4.2.5 (Pareto optimal point) *A point in Ospace is Pareto optimal iff there is no other point in Ospace that dominates this point:*

$$ParetoOptimal(\boldsymbol{o}^a) \leftrightarrow \neg\exists \boldsymbol{o}^b \in Ospace Dominates(\boldsymbol{o}^b, \boldsymbol{o}^a)$$

Definition 4.2.6 (Pareto frontier) *The set of all Pareto optimal points:*

$$ParetoFrontier = \{\boldsymbol{o}^a \in Ospace | ParetoOptimal(\boldsymbol{o}^a)\}$$

Figure 4.3 in Example 4.2 shows an example of a Pareto frontier.

For practical applications, such as in the control of embedded systems, a single result of the multi-objective optimization problem is necessary. If no additional information about the relative importance of the objectives is available, all points on the Pareto frontier are equally important and the selection of a single point from the Pareto frontier would be arbitrary. However, when information about the relative importance of the objectives is available (e.g., based on customer needs), a single optimal value can be selected. A preference relationship on the objectives is usually specified as a trade-off function in the objectives, providing a scalar value for each point in the objective space, and as such providing a total ordering relationship on the Pareto frontier.

Definition 4.2.7 (Trade-off function) *A function that maps values in the objective space to a scalar value ($tradeoff(\boldsymbol{o}) : \mathbb{R}^k \rightarrow \mathbb{R}$) for the purpose of having a preference relationship between the objectives to select a single value from a set of Pareto optimal values. The solution of the optimization problem is:*

$$\{\boldsymbol{x_o} | \boldsymbol{x_o} \in Dspace \wedge \forall \boldsymbol{x} \in Dspace tradeoff(\boldsymbol{o}(\boldsymbol{x_o})) \leq tradeoff(\boldsymbol{o}(\boldsymbol{x}))\}$$

Note that this set may contain multiple values. But as all these values have the same outcome of the trade-off function, there is no preference between them.

Example 4.2 Multi-Objective Optimization Example

Not all outcomes of the objective functions are optimal. The *Pareto frontier* in Figure 4.3 shows the *Pareto optimal* points in the objective space. Each of these optimal points in the objective space correspond to a certain value of the decision variables.

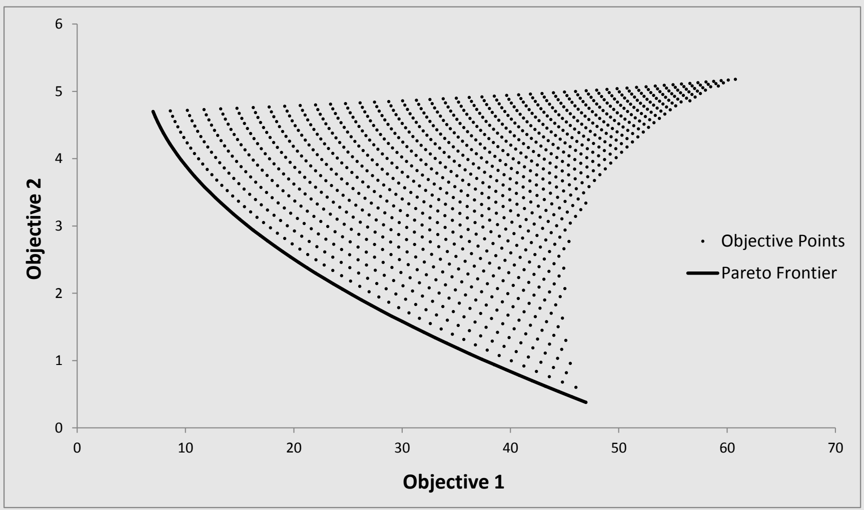


Figure 4.3: The objective space and Pareto frontier

The Pareto frontier can also be mapped in the feasible decision space, showing the decisions that lead to Pareto optimal points in the objective space. Figure 4.4 shows the Pareto frontier in the feasible decision space. Note that for all optimal values, the value of x_1 is equal to 8. This means that only x_2 is varied to make a trade-off between the objectives. The fact that one of the decision variables is constant for all optimal values is a coincidence in this example, but not always the case in general.

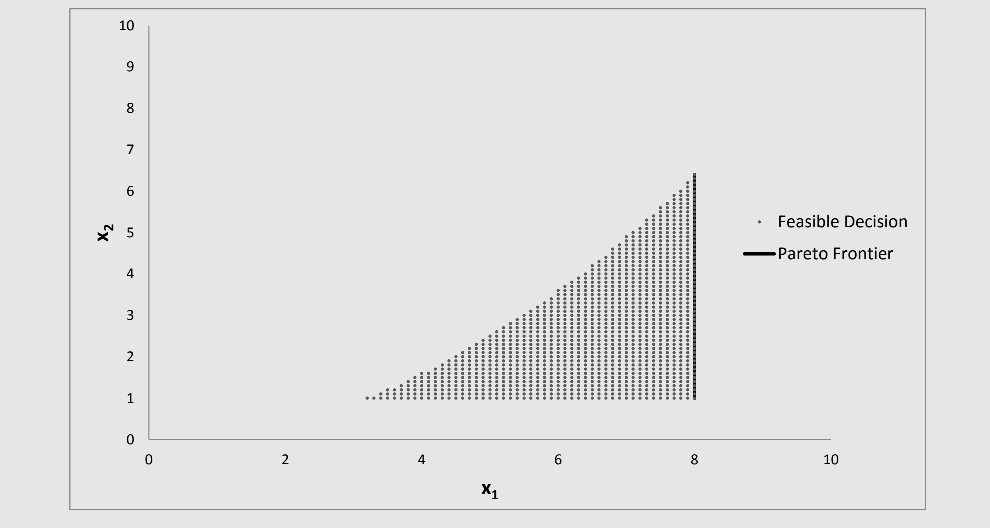


Figure 4.4: The Pareto frontier plotted in the feasible decision space

4.3 Context & Problem

In traditional embedded systems development, trade-offs between (conflicting) system qualities (e.g., productivity, energy consumption) are made at design time. This results in embedded systems in which the trade-off between these qualities is fixed. For example, the embedded system is either a high-productive system with high energy consumption, or an energy-saving system with low productivity. Nowadays, market forces demand more flexible and adaptable machines, in which the trade-offs between system qualities can be made at runtime. For example, a customer of a printing system might sometimes need a high-productive machine (e.g., for time-critical print jobs), while in general this customer prefers an energy-saving system. The embedded system has to dynamically optimize the system qualities under changing circumstances, such as changes in environmental conditions and changes in user input. Furthermore, the embedded system not only has to optimize the system qualities, but also has to make dynamic trade-offs between them, to accommodate changing user preferences.

In this section we are going to analyze what the impact of such functionality is on the design of embedded control software. First, the characteristics of multi-objective optimization in combination with the architecture of embedded control software are analyzed. Next, we explore the challenges that arise when engineers attempt to implement multi-objective optimization in embedded control software without systematic methods to support them. Finally, the requirements for a systematic method to design multi-objective optimization in embedded control software are proposed.

4.3.1 Analysis of Optimization in the Warm Process Case Study

This subsection analyzes the characteristics of multi-objective optimization and of control architectures that are important for the design of a multi-objective optimization solution in embedded control software.

Example 4.3 Multi-Objective Optimization in the Warm Process

In the Warm Process case study, engineers aim to introduce the possibility for the user to make trade-offs at runtime between the two conflicting objectives *power consumption* and *productivity* of the printing system. They identified the decision variables that can be used to influence the objectives, the different constraints in the system and the objective functions:

- **Decision Variables:** Speed of the system (v), temperature setpoint of the paper heater (T_{ph}^{sp}).
- **Constraints:**
  - $60 \leq v$ (Minimal speed is 60).
  - $v \leq 120$ (Maximal speed is 120).
  - $40 \leq T_{ph}^{sp}$ (Minimal setpoint for the paper heater is 40).
  - $T_{ph}^{sp} \leq 90$ (Maximal setpoint for the paper heater is 90).
  - $P_{ph} \leq 1200$ (Maximal power to the paper heater is 1200W).
  - $P_{rad} \leq 800$ (Maximal power to the radiator is 800W).
  - $P_{total} \leq P_{avail}$ (Total power consumption should not exceed the amount of power that is available to the system).
- **Objectives (Objective functions):**
  - Total power consumption: $P_{total} = P_{ph} + P_{rad}$.
  - Productivity: $Prod = 1/v$ (productivity is defined as an inverse of speed, as in multi-objective optimization the goal is to minimize the objective functions).

In the example multi-objective optimization problem for the Warm Process case study one might notice that the constraints and objective functions are not directly expressed in terms of the decision variables. The constraints are specified in terms of the components they relate to. For example, the installed radiator in the system has a maximum power of 800W. For an engineer, it is natural to describe this constraint as a constraint on P_{rad} ($P_{rad} \leq 800$), instead of a constraint on the decision variables v and T_{ph}^{sp}. The control logic in the system relates the variable P_{rad} to the decision variables.

Example 4.4 Warm Process Control Software Architecture

Figure 4.5 shows the architecture of the control software for the Warm Process case study.

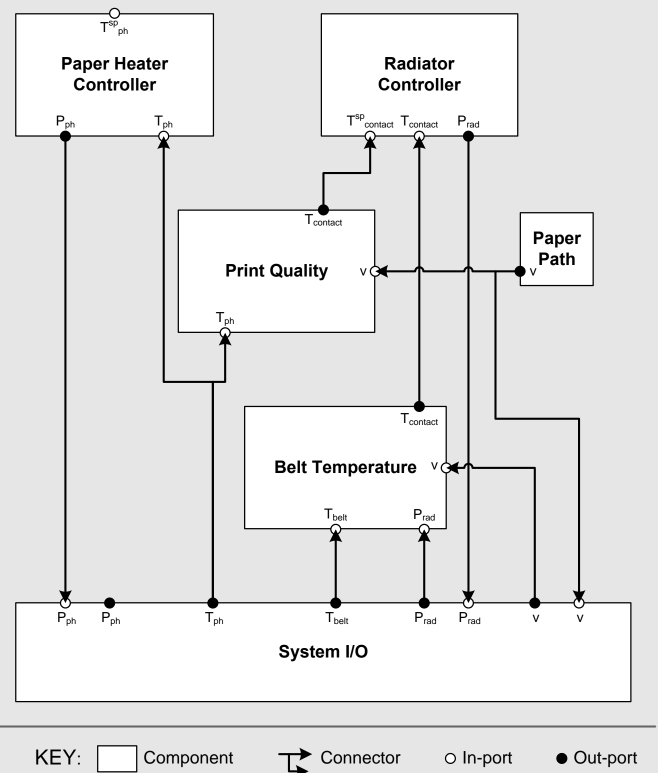


Figure 4.5: Warm process software architecture

The software architecture contains six different components. Components have interfaces, which are represented as *in-ports* and *out-ports*. In-ports can be used to provide a value to the component. Out-ports can be used to retrieve a value from the component.

The `System I/O` component provides an interface to the sensors and actuators in the system. The component has an out-port for each sensor (T_{ph} and T_{belt}). The component has both an in-port and an out-port for each actuator (P_{ph}, P_{rad} and v). The out-port for an actuator provides the current level of the actuator (i.e., the last value that has been provided to the in-port for the actuator).

`PaperHeaterController` and `RadiatorController` are components that contain the control logic for respectively the paper heater and the radiator. The components `PrintQuality` and `BeltTemperature` implement respectively the equation that determines print quality (Equation 1.1) and the equation that determines belt temperature (Equation 1.2). These two components can be implemented using physical model

instances, as was shown in Example 2.11 in Chapter 2. `PaperPath` is the component that controls the behavior of the paper path. It determines the speed of the system and provides this to the Warm Process control components.

If we relate the variables used in the specification of the multi-objective optimization problem to the different ports in the software architecture, we can conclude that these variables are related to different architectural elements (e.g. ports), which are spread through the software architecture. From this it follows that the optimization functionality has a system-wide impact in the architecture of the control software[2].

4.3.2 Lack of Systematic Methods

There is a lack of systematic methods to design and implement multi-objective optimization in embedded control software. Attempts to realize such functionality can lead to the following problems:

- **Ad-hoc solutions**. Because there is no systematic method to design multi-objective optimization, a software engineer may fail to make the proper abstractions and corresponding mapping to the domain of multi-objective optimization. This leads to solutions that are tailored to the specific system and therefore inflexible when the system changes or evolves, to solutions that are difficult to understand, because they do not apply standard multi-objective optimization abstractions and to solutions that may not optimally control the system, as an ineffective optimization algorithm has been chosen.
- **Tight coupling**. The different elements of an multi-objective optimization solution (decision variables, constraints and objective functions) are related to different architectural elements (ports, components) in the control software architecture. This creates a scattered relationship between the multi-objective optimization solution and the control software architecture. As such, it may be difficult to implement a proper decomposition of the chosen multi-objective optimization solution in embedded control software. This results in a multi-objective optimization solution that is tightly coupled and integrated with, and spread out over multiple control software components, making the control software less comprehensible and more difficult to maintain and reuse.

The above described problems lead to a reduction in software quality: ad-hoc and tightly coupled solutions are more difficult to comprehend, their inflexible nature hinders their evolvability and reusability. The lack of a common methodology and corresponding terminology makes it harder to document and communicate the design decisions. The result is higher development and maintenance costs [14]. Alternatively, as we have witnessed in practice, it can lead to the decision to remove the runtime optimization requirement, as the benefits of a more optimal system do not outweigh the reduced software quality and increased development and maintenance costs.

[2]In this example we apply multi-objective optimization to one function of the printing system, the warm process. Therefore, the multi-objective optimization solution only has an impact on the warm process control components. The classification as a system-wide impact is in this case a relative classification within the context of the warm process function.

4.3.3 Requirements for a Systematic Specification Method

Based on the previous analysis of the characteristics of multi-objective optimization in combination with the software architecture, we deduced the following requirements for a systematic specification method to design and implement multi-objective optimization in embedded control software.

1. The ability to solve the design of multi-objective optimization at the architectural level, instead of only the implementation level, because of the system-wide impact of a multi-objective optimization solution. Additionally, an architectural description provides documentation of the MOO solution.

2. The ability to connect each different element of the MOO problem (decision variables, constraints and objective functions) to an architectural element (e.g., a component, an interface) with which it has a conceptual relation. For example, a constraint on an actuator value should be connected to the element in the software architecture that provides that value to the actuator. This localizes the domain knowledge in the software architecture.

3. The ability to specify constraints and objective functions on other (physical) variables than the decision variables. As Example 4.3 shows, this enables the creation of more concise and better comprehensible constraints and objective functions. However, to be able to correctly process the specified multi-objective optimization solution, there should be a mathematical relationship between these other variables and the decision variables.

4. The ability to specify the mathematical relationship between the decision variables and the other variables used in constraints and objective functions.

5. The ability to extend an existing software architecture with a MOO solution without extensive restructuring. This provides the possibility to include multi-objective optimization solutions in existing embedded control software.

6. The ability to mathematically verify the consistency of the specified MOO solution in the software architecture. A specified MOO solution is inconsistent if there is a constraint or objective function that is not mathematically related to the decision variables.

7. The ability to generate code from an architectural description of the MOO solution.

4.4 MO2 Method Overview

To fulfill the requirements presented in the previous section, we propose the MO2 method to design and implement control software with multi-objective optimization. Figure 4.6 shows an overview of the MO2 method.

The MO2 method consists of two main parts: The *MO2 architectural style* and the *MO2 toolchain*.

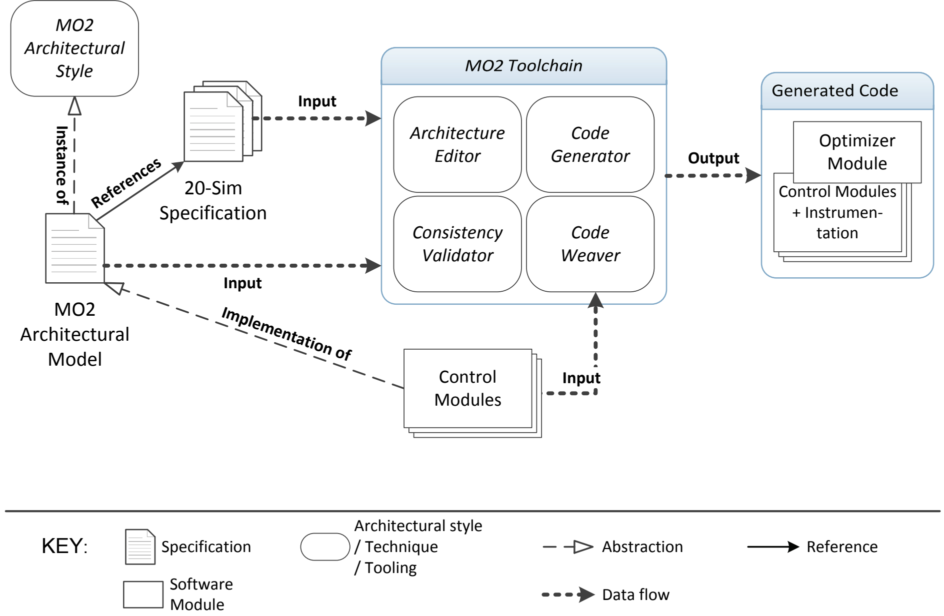


Figure 4.6: Overview of the Method

The *MO2 architectural style* (abbreviated as *MO2 style*) provides the ability to specify a Component-and-Connector model [27] for the architecture of the control software. This includes the possibility to specify the elements of the MOO solution (decision variables, constraints and objective functions) in a structured way within the control architecture. As such, the MO2 style supports the design and documentation of embedded control software that includes multi-objective optimization functionality. We refer to an architectural model that is created according to the MO2 architectural style as *MO2 architectural model* or *MO2 model*.

A MO2 model specifies the architecture of the control software, which consists of software components that implement control logic (i.e., *control components*). The interfaces of these software components consist of input and output variables. The control components are composed into a control architecture by connecting output variables to input variables. Furthermore, a MO2 model specifies which variables are decision variables, which variables have constraints (*constrained variables*) and which variables represent the outcome of objective functions (*objective variables*). In this way, a MO2 model includes the different elements (decision variables, constraints, objective function) of a specific MOO solution. A MO2 model serves two purposes:

1. To serve as design documentation of embedded control software containing MOO.

2. To serve as an input model to the MO2 toolchain. The MO2 toolchain uses a MO2 model as input to generate an optimization module specific for the given software architecture and defined MOO solution.

As explained in Section 4.3, the variables used to specify constraints and objective functions can be different from the decision variables. However, the computational logic that is implemented in the software modules creates a mathematical relationship between the decision variables and the variables used in constraints and objective functions. To be able to analyze a MO2 model and generate an optimizer module, the mathematical relationship between the decision variables and the other variables should be specified. Therefore, the MO2 method provides the possibility to refer to models that specify the computational logic (e.g., control logic, implemented physical characteristics) of components in the architecture(parts of) the mathematical relationships. These models can be specified in any language that is able to mathematically specify computational logic. In this thesis, and in our implementation of the MO2 method, we apply the SIDOPS+ language (20-Sim models), because of the suitability of this language to model control logic and physical characteristics.

The MO2 method includes a toolchain. The input to the toolchain is a MO2 architectural model and the referenced 20-Sim specifications. The toolchain contains a graphical editor, which is an extension of the ArchStudio 4 toolset [30], to create and edit MO2 architectural models. The *MO2 consistency validator* checks the consistency of the MOO solution. For example, it checks whether each variable that is used in constraints and objective functions has a mathematical relationship with the decision variables. Such a relationship should have been specified using 20-sim models. If the MO2 architectural model is consistent, it can be provided to the *MO2 code generator*, to generate an optimizer module specific for the given architecture and given MOO solution. The software modules that implement the basic control architecture are provided to the *MO2 code weaver*. The code weaver creates the interaction between these software modules and the generated optimizer module by weaving instrumentation in the software modules. The result is embedded control software that includes multi-objective optimization functionality.

The next sections explain the MO2 architectural style and MO2 toolchain in more detail.

4.5 MO2 Architectural Style

The *MO2 architectural style* is a specialization of the Component-and-Connector (C&C) viewtype as described in [27]. The following gives a brief description of the MO2 style, using the same description structure as used in [27] to describe architectural styles. The different characteristics of the style are described in further detail in following subsections.

4.5.1 Style Description

- *Elements*: Component;
- *Interfaces*: In-port, out-port;
- *Relations*: Connector;
- *Properties of elements*: Components have the following properties:

  - `20SimReference`: An optional reference to a 20-Sim model. This 20-Sim model describes the computational logic between the in-ports and out-ports of the component. If this computational logic is necessary to analyse the specified multi-objective optimization solution, a reference to a 20-Sim model should be provided. Otherwise, it can be omitted.
  - `constraints`: A list of constraints on the variables corresponding to the ports of the component.
  - `isOblivious`: A flag that indicates whether the component is *oblivious*. This means that there is no software module that implements the component: the component is only used by the engineer to specify the MO2 solution and provide this information to the tooling[3]. If the `isOblivious` flag is set, the component should always have a reference to a 20-Sim model.
  - `subModelReference`: An optional reference to another MO2 model. This represents encapsulation of MO2 models, to provide hierarchical application of the MO2 style.

- *Properties of interfaces*: In-ports and out-ports have the following properties:
  - `variableName`: String containing the name of the corresponding (physical) variable.
  - `constraints`: A list of constraints on the value of this port.
  - `isDecisionVariable`: A flag that indicates whether the variable corresponding to this port is a decision variable (i.e., the optimization algorithm may determine the value of the port).
  - `isObjective`: A flag that indicates that the variable corresponding to this port represents the outcome of one of the objective functions in the multi-objective optimization problem.

- *Properties of relations*: Same as the C&C viewtype [27].

- *Topology*: Connectors connect ports. The *start-point* of a connector is always an out-port. The *end-point* of a connector is always an in-port. The semantics of a connector is that the value on the end-point of the connector (in-port) is set to the value of the start-point of the connector (out-port). An out-port can be the start-point of multiple connectors. An in-port cannot be the end-point of more than one connector.

4.5.2 Style Notation

The MO2 style also has a notation to create graphical representations of *MO2 models*. These graphical representations are called *MO2 views*. Table 4.1 shows the different elements of the notation. Example 4.5 gives an example MO2 view.

[3]Hence the term *oblivious* component, to indicate that the actual implementation of the software is not aware of this component.

Notation	Description
«Stereotype» Name	Component with a *stereotype* and a *name*. Three stereotypes are available: `Analyzable`, `Oblivious` and `SubModel`. If a component has a reference to a 20-Sim model, then it has the stereotype `Analyzable`. If a component has the flag `isOblivious` set, then it has the stereotype `Oblivious`. Oblivious components have by definition a reference to a 20-Sim model, so the stereotype `Analyzable` is omitted. If the component references another MO2 model (i.e., hierarchical composition), the stereotype is `SubModel`.
○	In-port
●	Out-port
□ ■	In-port/out-port with the `isDecisionVariable` flag set.
▽ ▼	In-port/out-port with the `isObjective` flag set.
<Stereotype> Name id ●	Usage of a port: The ports that belong to a component are attached to the edge of the component. The port may be labeled with its `variableName`.
⟶	Connector
①	Informal label indicating that the component or port has constraints.

Table 4.1: Notation of the MO2 style

4.5.3 Basics of Components, Ports and Connectors

Component

In the definition of the Component-and-Connector viewtype in [27], *components* are "principal processing units and data stores" [27]. In the MO2 architectural style, the components implement the control logic, consisting of, among others, control algorithms and models of physical characteristics.

Port

Components in the MO2 style have *ports* to interact with other components. A port communicates the value of a specific (physical) variable. There are two types of ports: *in-ports* and *out-ports*. An in-port is used by a component to receive the value of a variable from other components. An out-port is used by a component to communicate the value of a variable to other components.

Example 4.5 MO2 Architectural Model

Figure 4.7 shows the MO2 view of a MO2 model for the case study introduced in Example 4.3. The labels 1 to 5 indicate that constraints have been added to the corresponding port/component. This example MO2 view will be used in the next section to illustrate the different characteristics of the style.

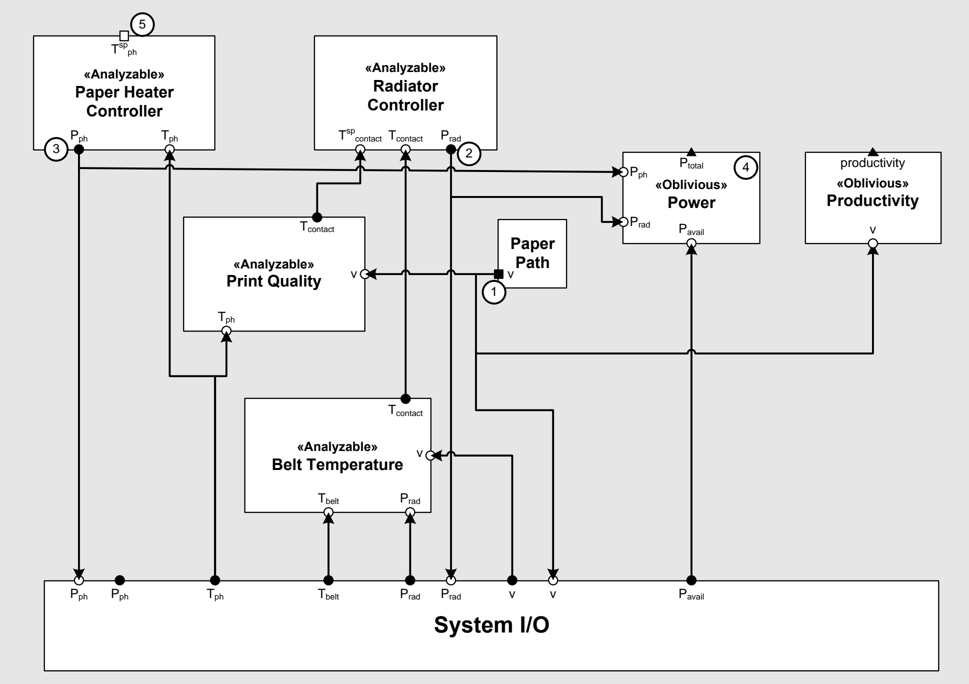


Figure 4.7: MO2 model of the case study

Connector

In the MO2 style, connectors communicate the values of variables between components, by connecting out-ports to in-ports. Out-ports can be the start-point of multiple connectors. In this way, the value on the out-port can be transmitted to multiple other ports. In-ports can be the end-point of only one connector. This allows only one source of values for an in-port.

4.5.4 Specifying the MO2 Solution

This subsection explains the characteristics of the MO2 style that can be used to specify a multi-objective optimization solution in a MO2 model.

Ports providing Decision Variables and Objective Functions

Ports have the flags `isDecisionVariable` and `isObjective`. By setting the `isDecisionVariable` flag, a designer indicates that the port represents a decision variable. This means that the optimization algorithm may choose the value on the port. If the port also gets a value from a connector (in case of an in-port) or a component (in case of an out-port), the value provided by the optimization algorithm overrides the value provided by the connector or component. In Section 4.8.1 a discussion is given about why the style allows setting the `isDecisionVariable` flag on existing ports and overriding the value provided by the component to the port.

Figure 4.7 shows two ports with the `isDecisionVariable` flag set: The in-port T_{ph}^{sp} of the `Paper Heater Controller` component and the out-port v of the `Paper Path` component. This makes the speed and the setpoint of the paper heater the two decision variables of the specified MOO solution in the MO2 model.

By setting the `isObjective` flag, a designer indicates that the value on the port represents the outcome of an objective function. Figure 4.7 shows two ports with the `isObjective` flag set: The out-port P_{total} of the `Power` component and the out-port *productivity* of the `Productivity` component. This means that there are two objectives in the system: total power consumption and productivity. Note that this MO2 model does not specify a trade-off function between the two objectives.

Constraints

Components and ports have the property `constraints`, which contains a list of constraints. Component constraints provide a constraint among the values on the component's ports (e.g., $P_{total} \leq P_{avail}$). Port constraints only constrain the value on that specific port. Note that port constraints can also be specified as component constraints. However, adding a constraint to a port, instead of adding an equivalent constraint to the corresponding component, documents more clearly that the constraint is specificly for the port.

In the MO2 view shown in Figure 4.7, there are four ports (labeled 1, 2, 3 and 5) that have constraints and one component (labeled 4) that has constraints. The constraints on these four ports and single component are:

1. Port: v
   (a) $value \geq 60$
   (b) $value \leq 120$
2. Port: P_{rad}
   (a) $value \geq 0$
   (b) $value \leq 800$
3. Port: P_{ph}
   (a) $value \geq 0$
   (b) $value \leq 1200$

4. Component: `Power`
   (a) $P_{total} \geq 0$
   (b) $P_{total} \leq P_{avail}$
5. Port: T_{ph}^{sp}
   (a) $value \geq 50$
   (b) $value \leq 90$

The two constraints on the out-port that is labeled 1 provide boundaries on the speed. The two constraints on the out-port labeled 2 provide boundaries on the power given to the radiator. Note that the optimizer is able to influence the value on this out-port by adjusting the speed; the control logic in components `Print Quality` and `Radiator Controller` relate speed to the value on the out-port labeled 2, i.e., the power given to the radiator (P_{rad}). The two constraints on the out-port labeled 3 constrain the power given to the paper heater. It can be influenced by the decision variable T_{ph}^{sp}. The two constraints on the `Power` component limit the power consumption of the system to the amount of power available. The two constraints on the decision variable T_{ph}^{sp} give boundaries for this decision variable.

20-Sim Reference / Analyzable Component

In a MO2 model, constraints and objective functions can be specified using variables (i.e., ports) other than the decision variables. However, there should be computational logic implemented in the components that provides mathematical relationships between the decision variables and the other variables used in constraints and objective functions. Otherwise, the constraints and objective functions cannot be influenced by the decision variables. To be able to analyze the MOO solution from the MO2 architectural model, the model should include these mathematical relationships. Therefore, the components in the MO2 architectural model have a property `20SimReference`. This property can be used to make a reference to a 20-Sim model that specifies the computational logic (e.g., a model of physical characteristics or continuous control logic) of the component. It is not necessary to include a 20-Sim model for each component. Only for components that relate constraints and objective functions to the decision variables the control logic should be included. As such, the mathematical relationships between the decision variables and other variables used in constraints and objective functions can be analyzed.

In the MO2 notation, components that have a reference to a 20-Sim model get the stereotype `Analyzable`. `Oblivious` have, by definition, a reference to a 20-Sim model. Therefore, in the notation the stereotype `Analyzable` is omitted for `Oblivious` components. Figure 4.7 shows four components with stereotype `Analyzable`. Their referenced 20-Sim models contain the control logic explained in Section 1.3. The figure also shows two components with stereotype `Oblivious`. Oblivious components are explained next.

Oblivious Components

Certain components in a MO2 model are present only as 'helper' components to specify additional constraints and objective functions. They do not have any other purpose at runtime, and as such there is no implementation of these components in software modules. This means that they are oblivious in the implementation and at runtime. To indicate in a MO2 model that a component is oblivious, the component's `isOblivious` flag should be set.

Figure 4.7 shows two oblivious components: `Power` and `Productivity`. The purpose of these components is to model the objective functions of the MOO problem and to model the constraint that the power consumption of the system is limited to the amount of power available. The `Power` component references a 20-Sim model containing the following equation: $P_{total} = P_{ph} + P_{rad}$. The `Productivity` component references a 20-Sim model containing the equation: $productivity = 1/v$ [4].

SubModel Components (MO2 Model Reference)

In hierarchical multi-objective optimization problems, the solutions of smaller MOO problems are combined to create the solution of a larger MOO problem. SubModel components provide the ability to hierarchically compose MO2 models. MO2 models, describing smaller multi-objective optimization solutions in sub-systems are composed into larger MO2 models, describing larger multi-objective optimization solutions in composed systems.

Components in a MO2 model have a property `subModelReference` with which a reference to another MO2 model can be made. This means that the component encapsulates the referred MO2 model, providing hierarchical composition of multi-objective optimization solutions. Lets define the following terms:

MO2 submodel
: A referenced MO2 model.

SubModel component
: A component in a MO2 model that represents/encapsulates another MO2 model.

Parent MO2 model
: The relative reference from a MO2 submodel to the MO2 model that contains a SubModel component referring to the given MO2 submodel.

Ports of SubModel components The ports of a SubModel component are not specified by the designer, but are defined by the MO2 submodel as follows:

- **In-ports:** The in-ports of a SubModel component correspond to the decision variables of the referred MO2 submodel. Note that such in-ports might not themselves have the flag `isDecisionVariable` set, as the decision variables of the MO2 submodel may be derived from other variables in the parent MO2 model.

[4] Strictly following the mathematical definition of a multi-objective optimization problem, the goal of multi-objective optimization is to minimize the objective functions. Therefore, productivity is expressed as the inverse of speed: higher speed leads to a lower outcome of the productivity objective function.

- **Out-ports:** The out-ports of a SubModel component reflect the objective functions of the corresponding submodel. In this way, the objective functions of the MO2 submodel can be hierarchically composed, for example into trade-off functions or larger-scale objective functions. Note that such out-ports might not themselves have the flag `isObjective` set, as the objective in the MO2 submodel might not be an objective in the parent MO2 model.

Figure 4.8 shows an example of a SubModel component. This SubModel component references the Warm Process MO2 model, as was presented in Figure 4.7, using the component's property `subModelReference`. The two in-ports of the SubModel component correspond to the two decision variables v and T_{ph}^{sp} in the Warm Process MO2 model. The two out-ports of the SubModel component correspond to the two objective functions *power* and *productivity* in the Warm Process MO2 model. Note that the two objective functions of the MO2 submodel are not objective functions in the parent MO2 model (as can be recognized from that fact that the ports are normal out-ports). The decision variable v is also not a decision variable itself in the parent MO2 model (in this case it should be dependent on a decision variable in the parent MO2 model). The decision variable T_{ph}^{sp} is also marked as a decision variable in the parent MO2 model.

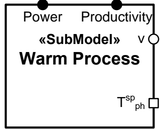


Figure 4.8: SubModel component of the Warm Process MO2 model

An example of a hierarchical MO2 model, using the Warm Process MO2 model as a MO2 submodel, is presented later in the chapter, in Section 4.7.2.

4.6 MO2 Toolchain

Besides an architectural style, the MO2 method also includes a toolchain to edit and validate MO2 models and to generate code from MO2 models.

Figure 4.9 shows an overview of the MO2 toolchain. The figure shows several artifacts and a number of processes that process certain artifacts to create other artifacts. One of the artifacts is the MO2 model that is the input to the toolchain. The MO2 model can be edited by the *MO2 Model Editor*. This editor is an extension of Archstudio [30]. It provides a graphical modeling environment to create and edit MO2 models. The MO2 models are stored in an extended version of an *xADL* file. xADL is the XML file format of Archstudio. The *MO2 Model Processor* takes as input an xADL file that specifies a MO2 model, and the SIDOPS+ specifications (containing 20-Sim models) referenced from components in the MO2 model. The MO2 Model Processor parses the provided xADL file and SIDOPS+ specifications, and generates a *mathematical representation* of the MO2 model, including the semantics from the referenced 20-Sim specifications. This mathematical representation is used by the

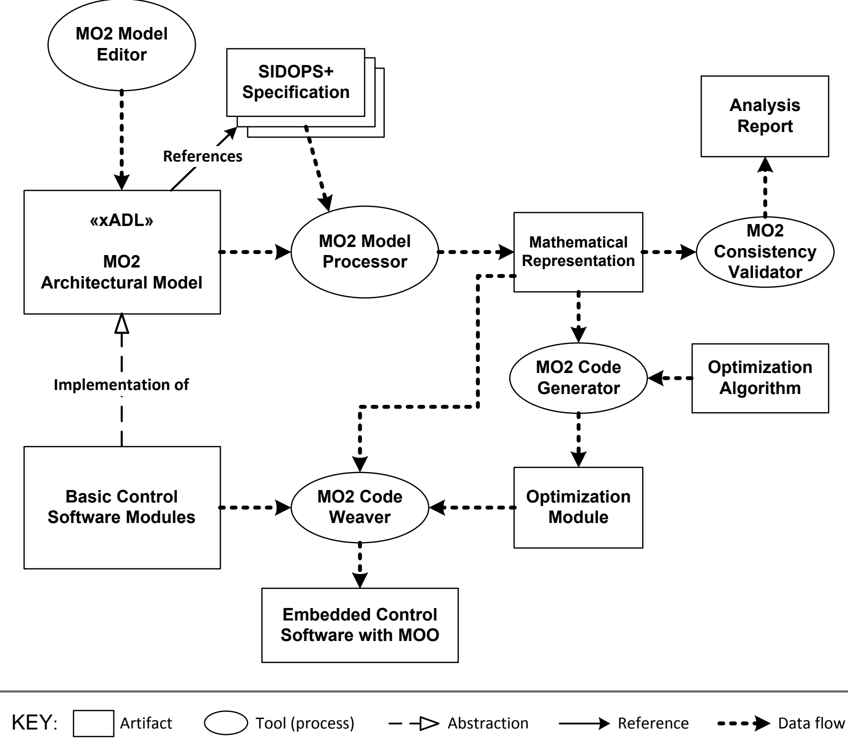


Figure 4.9: Overview of the toolchain

Consistency Validator to check the consistency of the MOO solution. If the MOO solution is consistent, the *Code Generator* generates an *optimization software module*, based on a generic optimization algorithm and the specific MOO problem provided by the mathematical representation. This optimization module needs to interact with the software modules that implement the basic control logic, to obtain values of certain (physical) variables and to influence the decision variables. The *Code Weaver* weaves this interaction into the control software modules. This results in embedded control software that includes multi-objective optimization functionality.

The different tools/processes in the MO2 toolchain are discussed in detail in the following subsections.

4.6.1 MO2 Model Editor

The Archstudio toolsuite [30] offers a graphical editor for Component-and-Connector models. We extended the Archstudio toolsuite to obtain a graphical editor for MO2 models. Figure 4.10 is a screenshot of the extended Archstudio tooling, showing a *structural diagram* that represents the MO2 model in Figure 4.7.

Archstudio stores the architectural models in the architecture description language called xADL [31]. xADL is an XML-based architecture description language that is

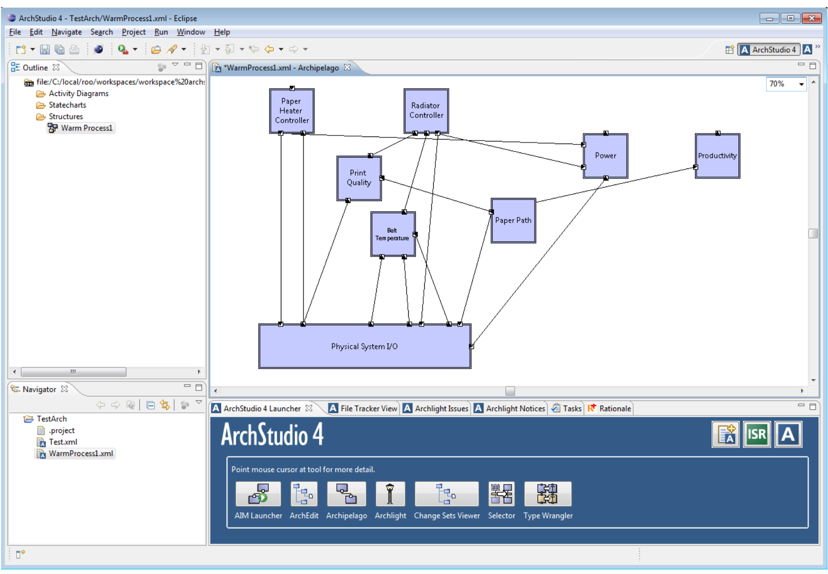


Figure 4.10: Screenshot of the MO2 Model Editor

defined using XML schema. The xADL schema can be extended to be able to express new elements and properties in an architecture description. We extended xADL to be able to add the different elements of the MOO solution (decision variables, constraints and objective functions) to the components and ports in the structural diagrams of xADL. Appendix C shows the XML Schema definition for the xADL extension.

4.6.2 MO2 Model Processor / Mathematical Representation

The *MO2 Model Processor* takes an xADL file as input, parses the SIDOPS+ files referred to by components in the xADL file and creates a mathematical representation of the MO2 model, including the semantics from the SIDOPS+ files. This section describes the mathematical representation of MO2 models and explains how this representation is created.

The basic model for the mathematical representation is a derivation graph (see Section 2.4.5 and Appendix A.2) that specifies how all variables in the architecture relate to each other. This derivation graph is created using the following steps:

1. A derivation graph of each component in the MO2 model is created.

2. The derivation graphs of the components are combined according to the connections between the components in the MO2 model.

These two steps are explained next. For the first step, a distinction is made between how a derivation graph is created for `SubModel` components and how such a graph is created for other components (i.e., `Analyzable` components, `Oblivious` components, regular components), as both procedures differ. First, it is explained how a

derivation graph is created for other components than `SubModel` components. Next, it is explained how the derivation graphs of components are combined into the derivation graph of the MO2 model. Finally, it is explained how the derivation graph of a `SubModel` component is created.

Step 1. Creating a Component's Derivation Graph

The derivation graph of a component is created using the following procedure:

Step 1a. Create the dependency graph If the component references a SIDOPS+ specification, this specification is parsed and the corresponding dependency graph is created (see Appendix A.1 for an explanation of dependency graphs). The dependency graph contains the relations between the different variables, as specified by the referred SIDOPS+ specification.

Example 4.6 Dependency Graph of a Component

Figure 4.11 shows an example component, two equations from the component's referred SIDOPS+ specification and a constraint of the component. Figure 4.12 shows the dependency graph that is created from the SIDOPS+ specification.

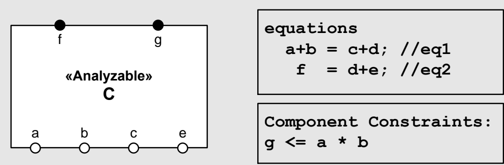


Figure 4.11: Example component, SIDOPS+ specification and component constraint

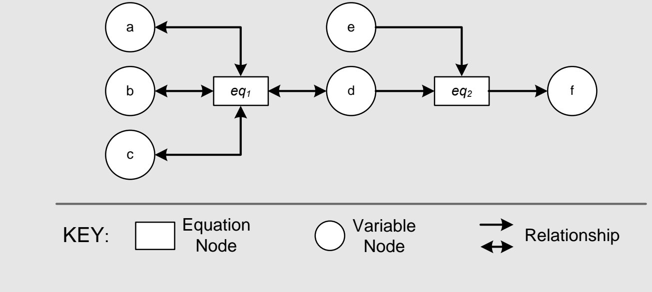


Figure 4.12: Corresponding dependency graph

Step 1b. Match ports to variable nodes Each port of a component corresponds to a variable. The variable nodes in a dependency graph also correspond to a variable. Using name matching of the corresponding variables, ports of the component are matched to variable nodes in the dependency graph. For each port that has no

matching variable node in the dependency graph, a new variable node is added to the dependency graph[5].

The result of this step is the relationship $portMatching : ports \rightarrow vNodes$. This relationship is used by other processes in the MO2 toolchain.

Example 4.7 Matching Ports to Variable Nodes

Figure 4.13 shows the matching of the component's ports to the variable nodes in the dependency graph. The out-port for variable g did not have a corresponding variable node. Therefore, a new variable node is added to the dependency graph.

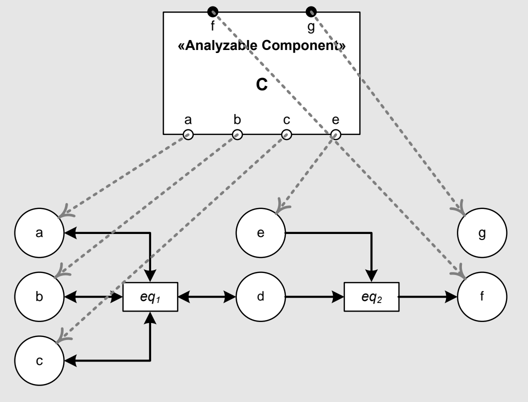


Figure 4.13: Matching of ports to variable nodes

Step 1c. Handling Component's Constraints A component can have constraints attached. These constraints should be reflected in the component's dependency graph. To include a component's constraint CC_i in the component's dependency graph, the following approach is taken:

1. The constraint CC_i is rewritten in one of the following forms:

   (a) $expressionCC_i < 0$, for 'greater than' and 'less than' constraints. For example, $g < a * b$ becomes $g - a * b < 0$.

   (b) $expressionCC_i <= 0$, for 'greater than or equal to' and 'less than and equal to' constraints. For example, $c \geq a + b$ becomes $a + b - c \leq 0$.

[5] This means that the variable corresponding to the port is not used in the SIDOPS+ specification. Note that this does not mean that the component's implementation does not use (in case of in-ports) or compute (in case of out-ports) the values of that port. The SIDOPS+ model of a component only needs to include the part of the semantics that relates constraints and objective functions to the decision variables. Therefore, the model may leave out unimportant relationships, in this way excluding certain variables.

   (c) $expressionCC_i = 0$, for equality constraints. For example, $b + c = d$ becomes $b + c - d = 0$.

2. A structure reflecting the mathematical equation $cc_i = expressionCC_i$ is added to the dependency graph. In this equation, cc_i is a new variable. $expressionCC_i$ is the expression formed in the previous step. The structure that is added to the dependency graph consists of an equation node, which reflects $expressionCC_i$ and a variable node, which reflects cc_i.

3. Depending on the type of the rewritten constraint in the first item of this approach, the cc_i variable node is annotated with the constraint $value < 0$, $value <= 0$ or $value = 0$. The mapping from the constraint to the variable node is added to the relationship $constraintMatching : constraints \rightarrow vN_{arch}$. This relationship is used by other processes in the MO2 toolchain.

Example 4.8 Component's Constraints in the Dependency Graph

Figure 4.14 shows the component's dependency graph extended with an equation node CC_1 and a variable node cc_1 for the constraint. The constraint $g \leq a * b$ is rewritten as $g - a * b \leq 0$. The equation node CC_1 reflects the equation $cc_1 = g - a * b$. The cc_i variable node gets annotated with the constraint $value \leq 0$.

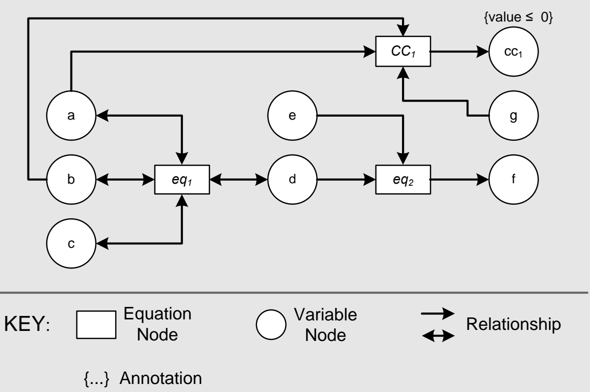


Figure 4.14: Dependency graph extended with the component's constraint

Constraints Attached to Ports

Constraints can also be attached to ports. After having discussed how component constraints are represented in the component's dependency graph, the question may be raised how port constraints are represented in the derivation graph of the corresponding component.

Each port of a component has a matching variable node in the dependency graph. Constraints on ports only constrain the variable corresponding to that port, no other variables. Therefore, a constraint on a port can directly be translated as a constraint on the variable node corresponding to the port. This is equivalent to, for example, the constraint added to the variable node cc_1 in Figure 4.14. As such, no additional constructs have to be added to the dependency graph.

Step 1d. Transform the dependency graph to the derivation graph The dependency graph only specifies the dependencies between variables as specified by the 20-Sim model. It does not represent how the values of certain variables are derived from the values of other variables. This is needed to analyze the relationship between the different variables in the MO2 model. Therefore, a derivation graph is created, applying the algorithm in Section 2.4.5. The variable nodes that provide the input for the derivation are those variable nodes that match with the component's in-ports.

Example 4.9 Component's Derivation Graph

Figure 4.15 shows the component's derivation graph, which is created from the dependency graph in Figure 4.14. The graph shows that the in-ports labeled a, b, c and e are independent variables for equations $eq1$ and $eq2$. Furthermore, the dependent variable d of $eq1$ is an independent variable for $eq2$. The dependent variable of $eq2$ is f, which is an out-port of the component.

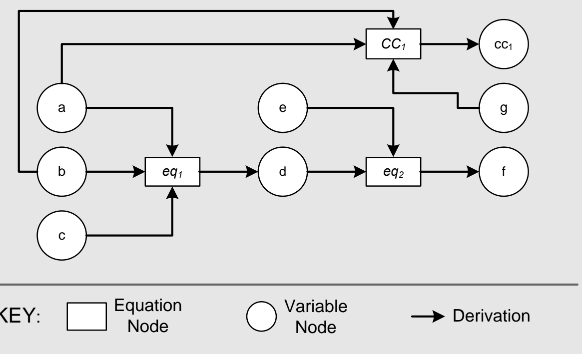


Figure 4.15: The component's derivation graph

In this explanation, we omitted how a derivation graph is created for `SubModel` components. To understand this works, it is necessary to know how the derivation graph of an entire MO2 model is created. Therefore, first step 2 is given, explaining how the derivation graph of an entire MO2 model is created. This is followed by an addition to step 1 that explains how the derivation graph of a `SubModel` component is created.

Step 2. Constructing the MO2 Model's Derivation Graph

The derivation graphs of each component in a MO2 model are combined to create the derivation graph for the entire MO2 model. The following explains how such a combination is performed.

Suppose G is the set of derivation graphs of the components in a MO2 model. The derivation graph $g_{arch} = (vN_{arch}, eN_{arch}, E_{arch})$ for the entire MO2 model is then created by combining the derivation graphs in G, as follows:

1. The set of variable nodes in the derivation graph of the MO2 model is the union of the sets of variable nodes in the derivation graph of each component: $vN_{arch} = \bigcup_{g_i=(vN_i,eN_i,E_i)\in G} vN_i$.
2. The set of equation nodes in the derivation graph of the MO2 model is the union of the sets of equation nodes in the derivation graph of each component: $eN_{arch} = \bigcup_{g_i=(vN_i,eN_i,E_i)\in G} eN_i$.
3. The set of edges is constructed in two steps:
   (a) First, create the union of the sets of edges in the derivation graph of each component: $E_{arch} = \bigcup_{g_i=(vN_i,eN_i,E_i)\in G} E_i$.
   (b) For each connector in the architecture: suppose the corresponding out-port has matching variable node $n1 \in vN_{arch}$ and the corresponding in-port has matching variable node $n2 \in vN_{arch}$, then add (n_1, n_2) to E_{arch}[6].
4. The *portMatching* mapping for the MO2 model is created by taking the union of the *portMatching* mappings for each component: $portMatching_{arch} = \bigcup_i portMatching_i$. The *portMatching* mapping is still needed to be able to relate nodes in the graph back to ports in the MO2 model.
5. The *constraintMatching* mapping for the MO2 model is created by taking the union of the *constraintMatching* mappings for each component: $constraintMatching_{arch} = \bigcup_i constraintMatching_i$.

[6]Note that the in-port and out-port attached to a connector always have a corresponding variable node.

Example 4.10 Derivation Graph of a MO2 Model

Figure 4.16 shows the derivation graph created for the example MO2 model from Figure 4.7 on Page 133. The derivation graphs created for the different components are indicated by dashed boxes.

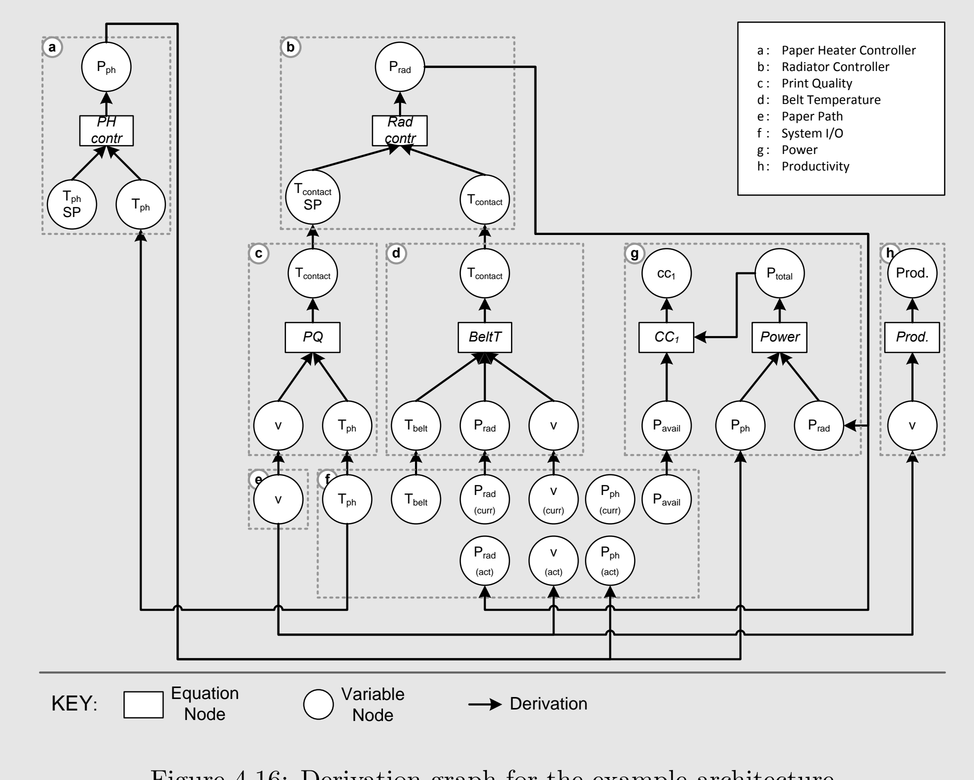


Figure 4.16: Derivation graph for the example architecture

Step 1 (Addition). Creating a `SubModel` Component's Derivation Graph

A `SubModel` component in a MO2 model (the parent MO2 model) references another MO2 model (the MO2 submodel). In the procedure of constructing the derivation graph of the parent model, the derivation graph of the `SubModel` component should be created and included in the derivation graph of the parent MO2 model. How the derivation graph of a `SubModel` component is created, depends on which *execution strategy* for hierarchical optimization is chosen. There are three different execution strategies for hierarchical optimization:

- **Local optimizer:** Each a MO2 submodel has its local optimizer.
- **Single global optimizer:** The MOO solution in the MO2 submodel is combined with the MOO solution in the parent MO2 model and only one optimizer performs the optimization based on this combined information.

- **A mix of the two other options:** Some MO2 submodels have their own local optimizer, while other MO2 submodels are first combined with the parent MO2 model to provide global optimization.

The derivation graph of the `SubModel` component is different for each of these three execution strategies. The composition of such derivation graphs into the derivation graph of the entire MO2 model (i.e., the parent MO2 model) is performed in the same way as previously described.

Option 1: Local Optimizer If the execution strategy of a MO2 submodel is using a local optimizer, this means that first optimization within the MO2 submodel (i.e., local optimization) is performed. Next, the results of the local optimization are used to perform optimization within the parent MO2 model. The results of the local optimization is a Pareto space for the MOO problem within the MO2 submodel. This Pareto space gives a mapping from values for the decision variables to outcomes of the objective functions[7].

The decision variables correspond, by definition, to the in-ports of the `SubModel` component. The objective functions correspond, by definition, to the out-ports of the `SubModel` component. Because the Pareto space that results from applying local optimization for a certain MO2 submodel provides a mapping from the decision variables to the outcome of the objective functions, this Pareto space provides the mathematical relationship that maps the in-ports of the corresponding `SubModel` component to the out-ports of this `SubModel` component.

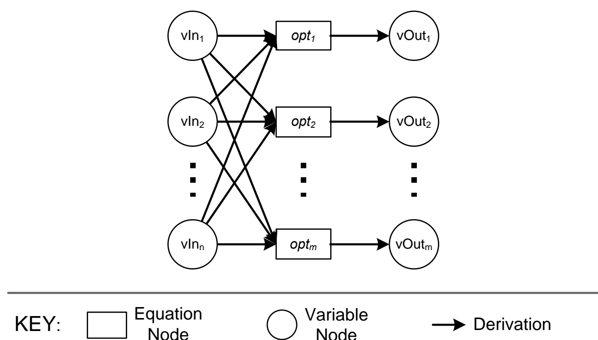


Figure 4.17: Structure of the derivation graph of a MO2 submodel

Figure 4.17 shows schematically the structure of a derivation graph for a `SubModel` component. The figure shows that for each of the m out-ports (representing the outcome of the objective functions) of the `SubModel` component there is an equation node to derive the value of the out-port. Each of these equation nodes represents the mapping in the Pareto space from a valuation of all decision variables to the outcome of the corresponding objective function. The specific mathematical relationship cannot

[7]Note that this is a partial mapping, as not all feasible valuations of the decision variables result in a Pareto optimal outcome of the objective functions.

be statically determined, as it follows from the Pareto space that results from the local optimization at runtime. The section about the MO2 code generator (Section 4.6.4) explains how this information is obtained at runtime, to perform the optimization at the level of the parent MO2 model.

Option 2: Single Global Optimizer If the execution strategy is to combine the MOO solution of the MO2 submodel with the MOO solution in the parent MO2 model before performing optimization, the derivation graph of the `Submodel` component is equal to the derivation graph that is constructed for the referenced MO2 model (the MO2 submodel).

For example, suppose that the MOO solution for the Warm Process case study, as shown in Example 4.5, is referenced from a higher-level, Printer System MO2 model. Then the derivation graph for the corresponding `Submodel` component in the Printer System MO2 model is the derivation graph that was shown in Figure 4.16.

Option 3: Mixed Local and Global Optimization It is possible to mix local optimization with single global optimization. In this case, some of the MO2 submodels have a local optimizer, while other MO2 submodels are first combined with the parent MO2 model for the Global Optimization. To create the mathematical representation in this situation, the strategy of option 1 is used for those SubModel components that have a local optimizer and the strategy of option 2 is used for those SubModel components that do not have a local optimizer.

4.6.3 MO2 Consistency Validator

The optimizer can only directly influence the valuation of the decision variables. To let the optimizer guarantee the satisfaction of a constraint or modify the outcome of an objective function in a MO2 model, there should be a mathematical relationship from one or more decision variables to the constraint or objective function. Only if such a relationship exists, the optimizer can guarantee the satisfaction of a constraint or modify the outcome of an objective function (by modifying the valuation of the decision variables). If there are constraints or objective functions in a MO2 model that do not have a mathematical relationship with the decision variables, such a MO2 model is inconsistent. Such an inconsistency in a MO2 model means that there are constraints and objective functions in the MO2 model that the optimizer cannot influence by changing the values of the decision variables. So, the optimizer is unable to guarantee that such constraint is satisfied or that an objective is actually being optimized. This section presents an algorithm to detect inconsistencies. This algorithm has been implemented in the MO2 Consistency Validator.

Besides checking for inconsistency after a MO2 model has been constructed, a partial MO2 model can be analyzed to detect for which components a reference to a 20-Sim model has to be provided to guarantee consistency. This might assist the designer in constructing consistent MO2 models. An algorithm to perform this analysis is also given in this section.

Dependency Analysis

The derivation graph of a MO2 model represents the mathematical relationships between the different variables/ports in the MO2 model. As such, the derivation graph can be used to detect which of these variables are influenced by the decision variables. Lets first define the following terms:

- **Dependent variable:** A variable that is influenced by one or more decision variables.
- **Dependent port:** A port in a MO2 model for which the corresponding variable is a dependent variable.
- **Dependent variable node:** A variable node in a derivation graph for which the corresponding variable is a dependent variable.
- **Dependent equation:** An equation in which at least one of the variables is a dependent variable.
- **Dependent equation node:** An equation node in a derivation graph for which the corresponding equation is a dependent equation.
- **Dependent node:** A dependent variable node or a dependent equation node.

Algorithm 4.1 detects all dependent nodes in a derivation graph, by checking for each variable node whether there is a path from a variable node that corresponds to a decision variable to the given variable node[8].

[8] The algorithms in this chapter use an object model of derivation graphs and of MO2 models. These object models are given in Appendix D.

Algorithm 4.1: Constructing the set of dependent nodes

```
input  : A derivation graph (vN_arch, eN_arch, E_arch) of a certain MO2 model,
         and the corresponding portMatching relationship that matches ports
         to corresponding variable nodes: portMatching : ports → vN_arch
output: The set of dependent nodes in the graph
1  dependentNodes := {}
   // Add decision variable nodes to dependentNodes
2  foreach (port, vNode) ∈ portMatching do
3  |  if port.isDecisionVariable then
4  |  |  dependentNodes := dependentNodes ∪ {vNode}
5  |  end
6  end
   // Walk through the graph to find all dependent Nodes (i.e., basic
      reachability analysis)
7  change := true
8  while change do
9  |  change := false
10 |  foreach (n_s, n_e) ∈ E_arch do
11 |  |  if n_s ∈ dependentNodes ∧ n_e ∉ dependentNodes then
12 |  |  |  dependentNodes := dependentNodes ∪ {n_e}
13 |  |  |  change := true
14 |  |  end
15 |  end
16 end
```

Example 4.11 Dependent Nodes in the Example Derivation Graph

Figure 4.18 shows in the example derivation graph from Figure 4.16 which variable nodes are dependent nodes. The figure clearly shows that the dependent nodes are only those nodes that are reachable in the graph from the decision variable nodes (v and T_{ph}^{sp}).

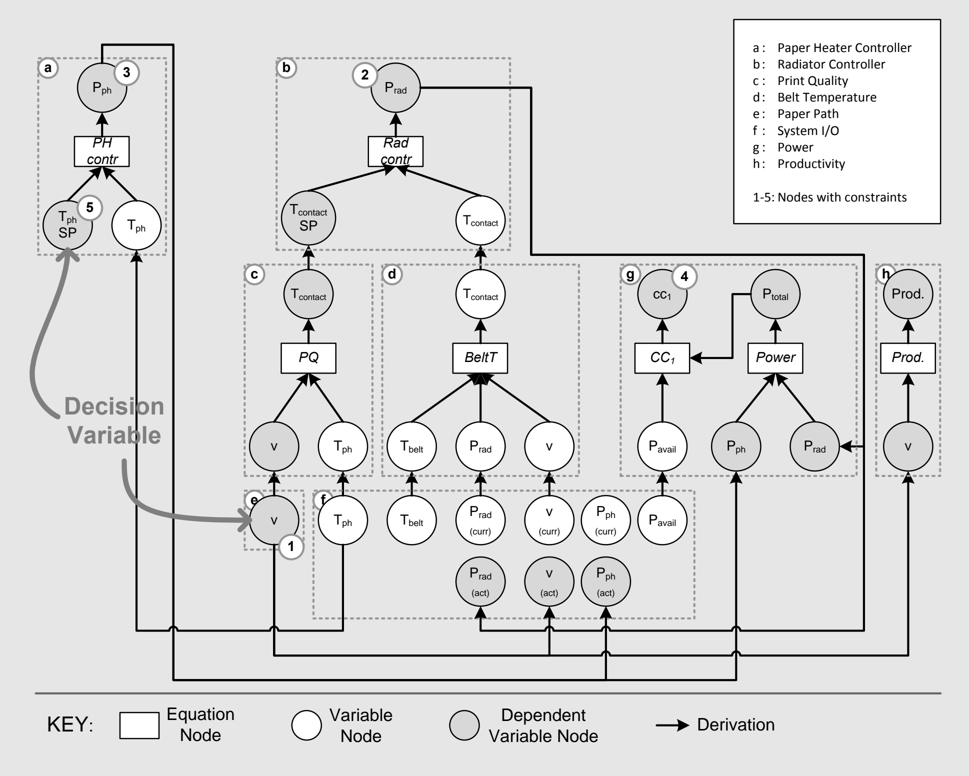


Figure 4.18: Dependent nodes in the example derivation graph

Consistency Analysis

Algorithm 4.2 performs consistency analysis of a MO2 model. It uses Algorithm 4.1 to create the set of dependent nodes. The algorithm checks for each port in the MO2 model that has constraints or is the outcome of an objective function whether the port is a dependent port. This is performed by checking whether the variable node that corresponds to the port is in the set *dependentNodes*. The algorithm also checks for the component constraints in the MO2 model, whether the corresponding variable node is in the set *dependentNodes*. If such a variable node is found not to be in the set *dependentNodes*, there is no mathematical relationship with the decision variables, so an inconsistency has been detected.

Algorithm 4.2: Consistency Analysis

input : A derivation graph $(vN_{arch}, eN_{arch}, E_{arch})$ of a certain MO2 model, the corresponding *portMatching* relationship that matches ports to corresponding variable nodes ($portMatching : ports \rightarrow vN_{arch}$) and the corresponding *constraintMatching* relationship that matches component constraints to their corresponding variable nodes ($constraintMatching : constraints \rightarrow vN_{arch}$)

```
1  Construct the set dependentNodes using Algorithm 4.1
2  foreach (port, vNode) ∈ portMatching do
3  |  if
   |  (port.constraints.size > 0 ∨ port.isObjective) ∧ vNode ∉ dependentNodes
   |  then
4  |  |  Inconsistency detected!
5  |  end
6  end
7  foreach (constr, vNode) ∈ constraintMatching do
8  |  if vNode ∉ dependentNodes then
9  |  |  Inconsistency detected!
10 |  end
11 end
```

Example 4.12 Consistency Analysis on the Example MO2 Model

Figure 4.18 shows that all variable nodes that have constraints (the variable nodes labeled 1 to 5) are dependent variable nodes. Furthermore, the variable nodes that represent the outcome of objective functions (P_{total} and $Prod.$) are dependent variable nodes. Therefore, the specified MO2 model is consistent. This means that an optimizer is able to influence all constraints and objective functions by changing the values of the decision variables. In this way, the optimizer can guarantee the satisfaction of the constraints and the optimization of the objective functions[9].

Detecting which Components need a SIDOPS+ specification

Besides checking the consistency of a MO2 model after it has been constructed, analysis can be performed during its construction to detect for which components a reference to a SIDOPS+ specification (i.e., 20-Sim model) is required to prevent inconsistencies. This might assist the designer in constructing the MO2 model when the structure of the architecture and the different elements of the multi-objective optimization solution (decision variables, constraints and objective functions) are known, but the designer does not know which components should reference a SIDOPS+ specification to make the specified multi-objective optimization solution consistent.

To perform this analysis, first a *structure graph* of the MO2 model is created. A structure graph shows the potential relationships between ports as specified by the components and connectors. A structure graph contains two types of nodes:

port nodes and *component nodes*. Port nodes correspond to ports in the MO2 model. Component nodes correspond to components in the MO2 model. Algorithm 4.3 shows how a structure graph $(pNodes, cNodes, edges)$ is created.

Algorithm 4.3: Creating a structure graph

input : A MO2 model $mo2Model$
output: A structure graph $(pNodes, cNodes, edges)$ for the given MO2 model

```
// Mapping from components to nodes:
```
1 $cNodeMap := \{(comp, \text{new Node}())|comp \in mo2Model.components\}$
```
// The set of component nodes:
```
2 $cNodes := \{cNode|(comp, cNode) \in cNodeMap\}$
```
// Mapping from ports to nodes:
```
3 $pNodeMap := \{(port, \text{new Node}())|comp \in mo2Model.components \wedge port \in comp.ports\}$
```
// The set of port nodes:
```
4 $pNodes := \{pNode|(port, pNode) \in pNodeMap\}$
```
// The set of edges:
```
5 $inPortEdges := \{(pNode, cNodeMap(port.component))|(port, pNode) \in pNodeMap \wedge port.isInPort\}$
6 $outPortEdges := \{(cNodeMap(port.component), pNode)|(port, pNode) \in pNodeMap \wedge port.isOutPort\}$
7 $connEdges := \{(pNodeMap(conn.startPort), pNodeMap(conn.endPort))|conn \in mo2Model.connectors\}$
8 $edges := inPortEdges \cup outPortEdges \cup connEdges$
```
// Return structure graph:
```
9 return $(pNodes, cNodes, edges)$

The following informally described procedure uses the structure graph to give insight in for which components a reference to a 20-Sim model should be made:

- For each port that has constraints or is the outcome of an objective function, and for each component that has constraints:
  - Find all paths in the structure graph from any decision variable node to the port/component node corresponding to the given port/component[10]. Paths with cycles can be excluded, as they are not relevant for the result. Suppose the result is the set *paths*.
  - There should be at least one $path \in paths$, for which all the components that correspond to the component nodes on this path reference a 20-Sim model. Only in this case a mathematical relationship might be present that relates the constraints or objective function to the decision variables[11].

[10] Note that the number of paths can be exponential in the size of the graph.
[11] Note that this does not guarantee the existence of a mathematical relationship: this depends on the actual 20-Sim specification.

4.6.4 MO2 Code Generator

This section presents the *MO2 code generator*. This part of the MO2 toolchain is responsible for generating a software module that contains the optimization functionality specific for the given embedded control software.

To generate an optimizer for the specific multi-objective optimization problem that has been specified in a MO2 model, the MOO problem structure (as was described in Definition 4.2.1) needs to be extracted from the MO2 model. An MOO problem structure consists of a set of decision variables, a set of constraints on these decision variables and a set of objective functions in these decision variables. This section first describes how to extract the MOO problem structure from a MO2 model and corresponding derivation graph. Then, this section explains how code is generated from the extracted MOO problem structure.

The procedures described in this section have access to the following information (for brevity, they are not supplied explicitly to the procedures, for example as arguments):

- The MO2 model: $mo2Model$.
- The derivation graph of the MO2 model: $g_{arch} = (vN_{arch}, eN_{arch}, E_{arch})$.
- The $portMatching$ relationship, which matches ports in the MO2 model to variable nodes in the derivation graph.
- The $constraintMatching$ relationship, which matches component constraints in the MO2 model to variable nodes in the derivation graph.
- The set $dependentNodes$, created by applying Algorithm 4.1.

Part 1. Extracting the MOO Problem Structure

Main Procedure Procedure `createMOOStructure` is the main procedure to create the MOO problem structure. It uses Procedures `getDecisionVariables`, `getComponentConstraints`, `getPortConstraints` and `getObjectiveFunctions` to create the decision variables, constraints and objective functions.

Procedure `createMOOStructure`

1 $decisionVariables := getDecisionVariables()$
2 $constraints := getComponentConstraints() \cup getPortConstraints()$
3 $objFunctions := getObjectiveFunctions()$
4 **return** $(decisionVariables, constraints, objFunctions)$

Procedure `getDecisionVariables` extracts the set of decision variables from a MO2 model, by iterating over the ports in the MO2 model and checking whether the port is a decision variable.

Procedure `getComponentConstraints` returns the set of *expanded* component constraints. Expansion of constraints means that a constraint specified in other variables than the decision variables is rewritten to a constraint in the decision variables, by

Procedure `getDecisionVariables`

1 $decisionVariables := \{port.variableName | comp \in mo2Model.components \wedge port \in comp.ports \wedge port.isDecisionVariable\}$
2 **return** $decisionVariables$

analyzing the mathematical relationships between the decision variables and the other variables in the MO2 model. For each component constraint, the procedure first creates the expanded expression, by applying the Procedure `expand` on the variable node in the derivation graph corresponding to the constraint. Next, the constraint is expanded by substituting $'value'$ for the expanded expression. The Procedure `expand` analyzes the derivation graph to create the expression that determines the value of a given variable node. This procedure is explained later.

Procedure `getComponentConstraints`

1 $expandedConstraints := \{\}$
2 **foreach** $component \in mo2Model.components$ **do**
3   **foreach** $constr \in component.constraints$ **do**
4         $expExpression := expand(constraintMatching(constr))$
5         $expConstraint := substitute(constr, 'value', expExpression)$
6         $expConstraints := expConstraints \cup \{expConstraint\}$
7   **end**
8 **end**
9 **return** $expConstraints$

Port constraints are expanded using Procedure `getPortConstraints`. Port constraints are expanded in a similar way as component constraints: by constructing the expanded expression from the derivation graph and by substituting $'value'$ for the expanded expression in the constraint.

Procedure `getObjectiveFunctions` creates the objective functions by constructing *expandedExpression* from the derivation graph for each port in the MO2 model that has the property `isObjective` set. The constructed *expandedExpression* is the objective function.

Expand Nodes in the Derivation Graph An expression for a constraint or objective function in a MO2 model is created using the derivation graph of the MO2 model. This is performed by traversing back through the derivation graph, starting from the variable node that corresponds to the constraint or objective function and ending at variable nodes that correspond to decision variables or that are independent. While traversing the graph, the expression is constructed. We call this process *expansion* of the variable node (or equation node). The following describes the procedures to expand variable nodes and equation nodes. Example 4.13 shows the procedure in an example.

The Procedure `expand(VariableNode varNode)` performs the expansion of variable

Procedure `getPortConstraints`

```
1  expandedConstraints := {}
2  foreach component ∈ mo2Model.components do
3  |  foreach port ∈ component.ports do
4  |  |  expExpression := expand(portMatching(port))
5  |  |  foreach constr ∈ port.constraints do
6  |  |  |  expConstraint :=
   |  |  |  substitute(constraint,′ value′, expandedExpression)
7  |  |  |  expConstraints := expConstraints ∪ {expConstraint}
8  |  |  end
9  |  end
10 end
11 return expConstraints
```

nodes. The procedure makes a distinction between four different types of variable nodes:

- **Dependent variable nodes:**
  - **Variable nodes corresponding to decision variables:** In this case, the expansion does not have to proceed further, as this is a decision variable.
  - **Other dependent variable nodes:** In this case, the variable corresponding to the node needs to be expanded to an expression. This is performed by expanding the node that is attached to this variable node through an incoming edge. Such an edge exists, because the variable node is in the set *dependentNodes*, so its value is derived from the decision variables.
- **Independent Variable Nodes** For independent variable nodes, expansion does not have to continue, as the value of these variables do not depend on the decision variables. Instead, at runtime the optimizer should obtain the value of the variable from the basic control architecture, before executing the optimization algorithm. How this is done, depends on two types of these variable nodes:
  - **Independent variable nodes with a corresponding port:** Certain variable nodes in the derivation graph correspond to ports in the MO2 model. At runtime, the optimizer should obtain the value of this port from the corresponding component. This is represented by the expression

Procedure `getObjectiveFunctions`

```
1  objFunctions := {expand(vNode)|(port, vNode) ∈
   portMatching ∧ port.isObjective}
2  return objFunctions
```

$valueOf(port)$, where *port* is the port in the MO2 model that corresponds to the variable node.

– **Independent variable nodes without a corresponding port:** It is possible that there are variable nodes that do not correspond to ports. This is the case for intermediate/additional variables in the 20-Sim specification referenced by a component. In this case, the value of the variable should be obtained from within the component (it is assumed that the variable is part of the component's state). This is represented by the expression $valueInState(variable, component)$. Section 4.6.5 explains in more detail how the *MO2 code weaver* implements this type of interaction between the optimizer component and the control software components.

Procedure `expand(` *VariableNode varNode* `)`

```
1  if varNode ∈ dependentNodes then
2  |   if ∃_port(port, varNode) ∈ portMatching ∧ port.isDecisionVariable then
3  |   |   return port.variableName
4  |   end
5  |   else
6  |   |   Find (startNode, varNode) ∈ E_arch
   |   |   // This edge exists, as the variable node is a dependent
   |   |      node and is not itself a decision variable.
7  |   |   return expand(startNode)
8  |   end
9  end
10 else
11 |   if ∃_port(port, varNode) ∈ portMatching then
12 |   |   return new Expression('valueOf', port)
13 |   end
14 |   else
15 |   |   Let (component, varNode) ∈ componentMatching
16 |   |   return new Expression('valueInState', varNode.variable, component)
17 |   end
18 end
```

The Procedure `expand(EquationNode eqNode)` performs the expansion of equation nodes. This procedure first determines the incoming variables (i.e., known variables). It solves the equation of the equation node with the incoming variables as the known variables. The result is an expression in the incoming variables. Next, it substitutes in this expression the incoming variables for their corresponding expression (determined by expanding the incoming variable nodes).

Procedure `expand(`*EquationNode eqNode*`)`

```
// Get incoming variable nodes
```
1 $inVarNodes := \{startNode | (startNode, endNode) \in E_{arch} \wedge endNode = eqNode\}$
```
// Determine incoming variables
```
2 $inVars := \{varNode.variable | varNode \in inVarNodes\}$
```
// Expand incoming variable nodes
```
3 $expVarNodes := \{(varNode, expand(varNode)) | varNode \in inVarNodes\}$
```
// Solve equation for the incoming variables using solve algorithm
   from Chapter 2
```
4 $expr := solve(eqNode.equation, inVars)$
```
// Substitute variables for the expanded subexpression
```
5 $expExpr := substitute(expr, expVarNodes)$

6 **return** $expExpr$

Example 4.13 Example expansion of P_{rad} variable node

Figure 4.19 shows the part of the derivation graph that is visited when the variable node P_{rad} is expanded (the derivation graph of the example MO2 model, including the set $dependentNodes$, was shown in Figure 4.18 on Page 150). The figure shows that the backward traversal of the graph stops at variable nodes that correspond to decision variables (in this case v) and variable nodes that are not in the set $dependentNodes$ (in this case $T_{contact}$ and T_{ph}).

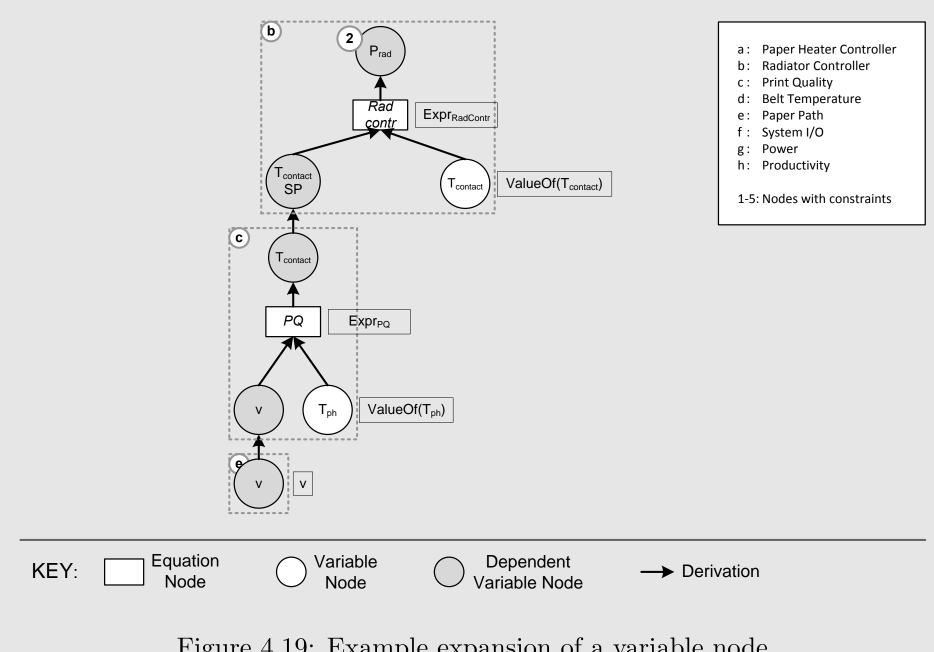


Figure 4.19: Example expansion of a variable node

The figure shows partial expressions beside certain nodes. These are the expressions that are constructed while visiting the corresponding node. The resulting expanded expression is:

$$Expr_{RadContr}(Expr_{PQ}(v, ValueOf(T_{PH})), ValueOf(T_{contact}))$$

The actual expression $Expr_{PQ}$ was given in Section 1.3.1, resulting in the following expression:

$$Expr_{RadContr}(c_1 \cdot v - c_2 \cdot ValueOf(T_{ph}) + c_3, ValueOf(T_{contact}))$$

Substitute Procedures The substitute procedures substitute given variable names in an expression for their corresponding sub-expressions. Procedure `substitute(Expression expr, Map<Variable,Expression> varMap)` substitutes in a given expression *expr* a set of variables for a set of sub-expressions, given by the mapping *varMap*.

Procedure `substitute(`*Expression expr, Map<Variable,Expression> varMap*`)`

```
1 foreach (var, subExpr) ∈ varMap do
2 |   expr := substitute(expr, var, subExpr)
3 end
4 return expr
```

The procedure `substitute(Expression expr, Variable var, Expression subExpr)` substitutes the given variable *var* in the given expression *expr* for the given expression *subExpr*.

Procedure `substitute(`*Expression expr, Variable var, Expression subExpr*`)`

1 Substitute the given Variable *var* in the given Expression *expr* for the given Expression *subExpr*

Part 2. Generating Code

After the MOO problem structure has been extracted from the MO2 model, code is generated that performs the following tasks:

1. Obtain information that can only be determined at runtime from basic control components for certain parts of the MOO problem structure. This information is, for example, the value at a certain port (`valueOf(aPort)`) or a value in the state of a component (`valueInState(aVariable, aComponent)`).

2. Provide the MOO problem structure to a generic optimization algorithm in a format that the generic optimization algorithm can process. The optimization

algorithm then determines a Pareto space or a single optimal outcome (in case there is trade-off function).

3. Process the result of the optimization algorithm:
   (a) In case of hierarchical optimization with another optimizer on a higher level: provide the Pareto space to the higher level optimizer.
   (b) Otherwise: set the decision variables in the basic control components to the result of the optimization.
4. Iterate over the above steps at a certain frequency (control loop).

Obtaining Information The first step obtains information from the basic control components to create the actual MOO problem structure at that moment in time. The actual instrumentation of the basic control components, to be able to obtain the information, is performed by the *MO2 code weaver*. The following types of information may be required:

Value of a port
As described before, expressions in a MOO problem structure may contain `valueOf(aPort)` statements (`aPort` is an independent port), which means that at runtime this statement should be substituted with the value of `aPort`. The *MO2 code weaver* provides instrumentation in the basic control component corresponding to `aPort`, to obtain the information. The *MO2 code generator* has to generate code that uses this instrumentation to obtain the value and substitute this value in the MOO problem structure.

Value in the state of a component
Expressions in a MOO problem structure may contain `valueInState(aVariable , aComponent)` statements, which means that at runtime this statement should be substituted with the value of `aVariable` in the state of `aComponent`. Similar to the value of a port, the *MO2 code weaver* generates instrumentation and the *MO2 code generator* generates code that uses this instrumentation to obtain the value and substitute this value in the MOO problem structure.

Stateful operators/functions
Certain operators/functions in expressions (which are acquired from the 20-Sim models) might maintain a certain state. An example is the function $int(x)$, which calculates the integral of x over time. This integral function maintains, over time, the current value of the integral. For such functions, the *MO2 code weaver* generates instrumentation to obtain such state information from the component that implements this function. The *MO2 code generator* generates code that uses this instrumentation to obtain the state information and applies this state information to the MOO problem structure.

Calling the Optimization Algorithm To perform the optimization, code is generated that supplies the MOO problem structure to a generic optimization algorithm in a format that it understands. Certain operators/functions may be translated to equivalent operators/functions that the optimization algorithm can process.

The current implementation only supports the Matlab function `fmincon` [4] as an optimization algorithm. Support of other optimization algorithms is considered to be future work.

Processing the Result of the Optimization If there is no higher-level optimizer, the result of the optimization process is provided to the decision variable ports in the basic control components. The *MO2 code weaver* generates instrumentation in these components to be able to set the value. The optimization code generated by the *MO2 code generator* decides on the actual value and sets the value using the generated instrumentation.

If there is a higher-level optimizer, the resulting Pareto space of the local optimization is provided to the higher-level optimizer. The Pareto space is a mapping of values of decision variables to Pareto optimal outcomes of the objective functions. The higher-level optimizer substitutes this mapping in its own MOO problem structure before providing this MOO problem structure to the higher-level optimization algorithm.

4.6.5 MO2 Code Weaver

The *MO2 code weaver* instruments the basic control components. The optimization code generated by the *MO2 code generator* uses this instrumentation to obtain information from the basic control components and to provide the optimized value for the decision variables.

Obtaining Information

Some of the options that are available for instrumentation are:

- Generating a `get` function, if not already available. This can be performed by extending the implementation of the code or using aspect-oriented extension mechanisms (such as the *inter-type declaration* mechanism of AspectJ [7] or the *interface extension* mechanism of Compose* [33, 34]). The disadvantage of this method is that the interface is extended, which may enable unsolicited access by other components.

- Using aspect-oriented mechanisms that monitor for changes in the instance variables of the component that correspond to the information to be obtained. This enables direct interaction with the code generated by the *MO2 code generator*. However, there is a more tight integration between the *MO2 code generator* and *MO2 code weaver*, as they need to exchange information to create the direct interaction.

The current implementation does not apply any of these methods. Instead, it is assumed that `get` functions are already available to obtain the required information.

Providing the Valuation of the Decision Variables

To provide the valuation of the decision variables back to the basic control components, instrumentation is generated that enables the optimizer component to set these values and that overrides values supplied to the decision variables by basic control components. Aspect-oriented mechanisms are used to intercept calls to `set` functions of the decision variables and replace the argument with the value determined by the optimizer.

4.7 Advanced Application

This section demonstrates two more advanced applications of the MO2 method. So far, the example case consisted of a single MO2 model and performed instantaneous optimization. Instantaneous optimization means that the objective functions are optimized for the current time instance, e.g., minimal power consumption at the current time instance. This section shows how the MO2 method can be applied to a case that performs optimization over time, which means that a decision is selected that provides an optimal outcome of the objective functions over a certain time period. Furthermore, this section introduces an example case that demonstrates hierarchical optimization with the MO2 method.

4.7.1 Optimization over Time

In the example application of multi-objective optimization to the Warm Process case study the optimization was done instantaneously, which means that the system's behavior is optimal at that moment in time. However, instantaneous optimization is not always effective: a decision that seems optimal on the short term may turn out to be far from optimal in the long term. This is actually the case for the Warm Process case study. In this case study, the non-optimal behavior in the long term is caused by the decision variable T_{ph}^{sp} (setpoint of the paper heater). Lowering the setpoint of the paper heater will drop the power consumption for the same productivity on the short term. So, this seems the optimal decision. But over time, the lower setpoint will result in a lower T_{ph} (as the heat capacity of the paper heater drains). To compensate, a higher $T_{contact}$ is necessary, leading to a higher power consumption of the radiator (P_{rad}). As a higher T_{ph} is more power efficient than a higher $T_{contact}$, lowering T_{ph}^{sp} thus will result in higher total power consumption over time. So, a decision that is optimal on the short term leads to suboptimal behavior on the long term.

The problem can be prevented by only having v as a decision variable. But this gives a less flexible system. Making T_{ph}^{sp} a decision variable is beneficial in case of a fluctuating power supply. Suppose that most of the time the amount of power available is sufficient. But during certain small time periods, the amount of power available is less than needed to run at the required speed. This may for example be caused by weak mains or by a certain other device (e.g., a binder that glues printed sheets together) that once in a while consumes power from the same power supply. The optimizer can now decide to lower T_{ph}^{sp} for this short period of time. In this way, the same speed can be maintained with lower power consumption, by utilizing the

heat capacity (i.e., stored energy) of the paper heater. When there is enough power available again, the temperature of the paper heater can be restored.

In cases where instantaneous optimization is ineffective, optimization should be based on the utility that is provided over a certain time period. Model-predictive control theory provides the algorithms necessary to do such optimization [79]. We show how the MO2 method can be applied on a system that applies optimization over time using model-predictive control.

Example 4.14 MO2 Model for Optimization Over Time

Figure 4.20 shows the MO2 view of the architecture. There are two big differences, compared to the view of the original case, which was shown in Figure 4.7 on Page 133.

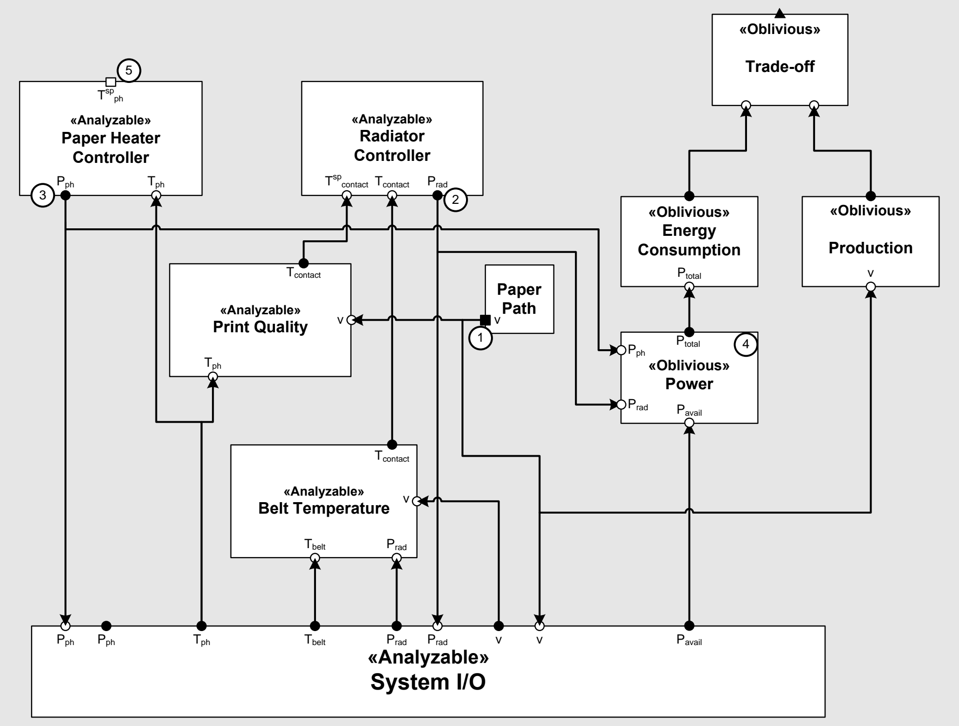


Figure 4.20: MO2 view with optimization over time

The first difference is the addition of the `Energy Consumption` component, which is placed between the `Power` component and the `Trade-Off` component. The `Energy Consumption` component calculates the energy consumption over time using the formula $\int P_{total}\,\mathrm{d}t$. The `Productivity` component has been replaced by the `Production` component, which provides production over time using the formula $1/\int v\,\mathrm{d}t$. In this way, the system makes a trade-off between consumed energy and production over a certain time period.

The second difference is that the `System I/O` component has stereotype `Analyzable` and refers to a 20-Sim model. This 20-Sim model is not a representation of the component's semantics, but a model of the physical characteristics of the system. This is typical for solutions that apply optimization over time. Such a 20-Sim model of the system is needed to predict the effect of actuation on the system and on the sensor values over time. The model-predictive optimization algorithm uses this 20-Sim model to 'simulate' the physical behavior of the system, to predict which decision is optimal over time.

4.7.2 Hierarchical Optimization

Example 4.5 in Section 4.5 showed the MO2 model for the Warm Process case study. In a digital document printing system, the Warm Process is one of many subsystems in the system. Drum Shuttling is an example of another subsystem. The embedded control software is generally divided according to such subsystems. For example, there is a software unit for the control of the Warm Process subsystem, a software unit for the control of the Drum Shuttling subsystem, etc. If there is a MO2 model for each of these software units, they can be hierarchically composed to provide a MO2 model for the larger software unit (or for the entire embedded control software) that contains the mentioned software units.

Example 4.15 Hierarchical Application of the MO2 Style

In this example case we consider a printing system that consists of three subsystems: the *Warm Process* subsystem, the *Drum Shuttling* subsystem and the *Finisher:Glue Binder* subsystem. The Warm Process subsystem and the Drum Shuttling subsystem have been introduced before (e.g., in Section 1.3). The Finisher:Glue Binder subsystem contains the control logic for a specific type of finisher, called a *glue binder*. A finisher is a device that can be attached to a printing system and provides additional processing of printed sheets of paper, such as stapling. A glue binder is a finisher that puts a cover around a stack of printed sheets and binds them together by applying glue. Figure 4.21 shows the MO2 view of the hierarchically composed MO2 model.

This figure shows the three `SubModel` components that correspond to the three subsystems. In the parent MO2 model there are two decision variables: v and T_{ph}^{sp}. Each MO2 submodel also has a decision variable v, as is reflected by the in-ports of the `SubModel` components in the parent MO2 model (remember that in-ports of `SubModel` components reflect decision variables in the referred MO2 submodel). The decision variable T_{ph}^{sp} originates from the Warm Process MO2 submodel.

Each MO2 submodel in this example has its own power consumption. The component `Total Power` calculates the total power consumption, by summing the power consumption of each subsystem. The out-port of this component has a constraint (labeled 3) that limits the total power consumption to the amount of power that is available.

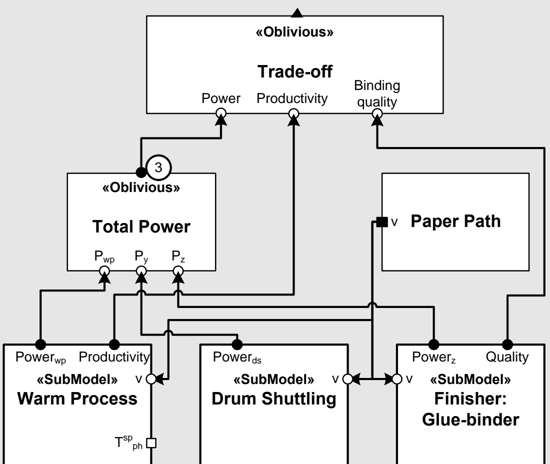


Figure 4.21: MO2 view showing hierarchical application of the style

The MO2 model utilizes global optimization. The parent MO2 model is used to compose the multi-objective optimization solutions provided by the MO2 submodels. The parent MO2 model also implements a trade-off between the objective *power* (total power consumption) and the two objectives from the MO2 submodels: *Productivity* and (Binding) *quality*.

4.8 Discussion

4.8.1 Values Provided to Ports with the isDecisionVariable Property Set

The value of an out-port is determined by the component to which the out-port is attached. The value of an in-port is provided through the connector that connects the in-port to an out-port. But if the `isDecisionVariable` property of an out-port or in-port has been set, the value provided by the component or connector is overridden by the optimizer. This raises the question why the MO2 style uses in-ports and out-ports as decision variables, instead of introducing a new element for this. Such a new element could, for example, be a *decision-variable-port* that is detached from components and that can be used as the start-point of a connector (i.e., it is a source of values). The reason why the MO2 style uses in-ports and out-ports as decision variables is two-fold. First, this approach supports introduction of MOO to existing control software. If a control structure already exists, MOO can be introduced by setting the `isDecisionVariable` property of certain in-ports and/or out-ports, instead of changing the basic control structure to accommodate new elements. Second, this approach provides robustness for the system. If the optimization component fails for one reason or another, and needs to be shut down, the analyzable component to which the decision variable port is attached can provide a value. Although this value does

not let the system operate optimally, it enables to system to continue its execution.

4.8.2 Generating Code for Analyzable Components

Because the 20-Sim model referred to by an analyzable component contains (part of) the component's semantics, this 20-Sim model could be used to generate the component's implementation. However, such a 20-Sim model might not contain all semantics of the component: it only contains the part of the semantics that is necessary to automatically construct a mathematical representation of the specified MOO solution. A component's implementation might have additional semantics, such as logic to determine/change constraints on its ports, logging code, etc. In this case, it is necessary to extend the generated implementation of the component.

4.8.3 Computational Performance of Multi-Objective Optimization

Applying multi-objective optimization algorithms results in a better performing system, but can also lead to a considerable computational performance overhead in software. This concern is particularly relevant for embedded systems, as these systems generally tend to have limited processing power available. In the end, this concern is an engineering trade-off for which the engineer has to decide between a better performing system or more efficient software that can execute on limited processing hardware.

The computational complexity of solving a multi-objective optimization problem depend on the characteristics of this problem. Certain multi-objective optimization problems can be solved in polynomial time (e.g., linear programming problems [109]), while other problems are NP-hard [53]. As such, the trade-off can be influenced by careful selection of the multi-objective optimization algorithm and adapting the multi-objective optimization problem in such a way that it can be efficiently solved. For some systems an approximation of the optimal solution would be sufficient. In this case, using an approximation algorithm, instead of an algorithm that gives an exact result can increase the efficiency of the optimization.

Many different multi-objective optimization algorithms have been designed, both algorithms that give exact results and algorithms that give approximations. An extensive overview of multi-objective optimization can be found in [43].

4.9 Related Work

4.9.1 Architectural Styles

The Views&Beyond approach makes a distinction between *viewtypes*, *styles* and *views* [27]. Viewtypes, like the Component-and-Connector viewtype, are broad categories of views. Styles are specializations of viewtypes that define recurring instantiations of the viewtype. Examples of C&C styles are the Client-Server style and the Pipe-and-Filter style [27]. Views are instantiations of a viewtype, possibly adhering to one

or more styles, for a specific system. In this chapter, we have introduced a specific style for documenting MOO solutions for embedded systems.

The IEEE 1471 standard describes that an architecture consists of a set of views [81]. Each of these views conforms to a certain viewpoint. Thus, in the parlance of IEEE 1471, the MO2 style can be regarded as a viewpoint.

Specific styles for control software have been developed before, like the Process Control styles described in [97]. But these styles take a more general view on control; they describe the system in terms of a controlled system, sensors, controllers and actuators. The MO2 style is applicable to a more specific type of functionality in control software, multi-objective optimization, and therefore can exploit specific properties of this functionality. Hofmeister et al. have investigated using the Safety Vision case in [62] how different architectural views for control software relate. Their work is not specific for multi-objective optimization, but could assist in realizing software using the MO2 style.

4.9.2 Theory of Multi-Objective Optimization

Multi-objective optimization was first introduced in works on decision making in economy by Edgeworth [42] in the late nineteenth century. Pareto extended the work with the concept of Pareto optimality [91]. General information about multi-objective optimization can be found in [69] and [28].

Ehrgott et al. give in [43] an extensive overview of mathematical research in multi-objective optimization. They treat subjects like *goal programming*, *fuzzy multi-objective optimization*, *evolutionary* algorithms for multi-objective optimization and *multi-criteria heuristics*.

Marler and Arora give in [84] a more concise overview of multi-objective optimization approaches, specifically for engineering. For example, they give a summary of approaches to create trade-off functions between objectives, such as the *weighted sum method* [114], the *lexicographic method* and the *weighted min-max method*, also called *weighted Tchebycheff method* [84]. There are many systematic approaches to select appropriate weights for weighted trade-off functions: the *ranking method* that ranks the different objectives [110] and the *Eigenvalue method* in which pair wise comparison of the objectives is performed to determine weights [95]. Extensive surveys of weight selection methods can be found in [41, 61, 65, 105].

Lui et al. describe in [78] how multi-objective optimization can be applied in the design of control systems. They treat subjects like how to design robust control systems using multi-objective optimization, multi-objective PI and PID controller design and multi-objective control of critical systems. Compared to the work in this chapter, Lui et al. mainly focus on more localized control problems, for example optimized behavior of a specific controller. In this chapter, system qualities are the subject of the multi-objective optimization, and the multi-objective optimization solution has an impact on the system as a whole. Also, Lui et al. primarily handle the mathematical design of an optimization algorithm, while the MO2 method focuses on how to handle the complexity of such an algorithm in the design of the system.

In hierarchical multi-objective optimization problems, the solutions of smaller MOO problems are combined to create the solution of a larger MOO problem. Geilen

et al. have developed an algebra to compose the solutions of smaller MOO problems into a solution of a larger, composed MOO problem [51]. This provides the mathematical basis behind the hierarchical application of the MO2 style.

Several (open-source) implementations of MOO algorithms exist, such as in [2, 3]. Matlab provides a function, called *fmincon*, to perform optimization of a single objective [4], which can be for example a trade-off function of multiple objectives. Our implementation of the MO2 code generator generates code that uses *fmincon* to perform optimization.

4.10 Conclusion

This chapter presented a systematic method, called the MO2 method, to design and document multi-objective optimization within the architecture of embedded control software. The foundation of this method is the MO2 architectural style to create MO2 architectural models. This style is supported by a notation and the MO2 toolchain. The MO2 style serves two purposes. First, to support the design and documentation of control software containing multi-objective optimization functionality. Second, to support the development/generation of optimization components tailored to the given software architecture. This is performed by extracting static information about the multi-objective optimization solution from the MO2 model and using this information to synthesize a mathematical representation of the multi-objective optimization solution. Analysis techniques support this process, e.g., to detect consistency conflicts in the specification of the MOO solution. The mathematical representation of the multi-objective optimization solution becomes part of the implementation of the optimization component.

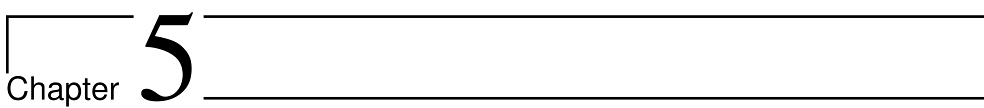
The presented MO2 method fulfills the requirements given in Section 4.3.3. First, the MO2 method has an architectural focus, by providing the MO2 architectural style. This style supports the systematic design and documentation of multi-objective optimization solutions in embedded control software by prescribing how such solutions can be modeled within the architecture. As the MO2 style uses a general view on the control software architecture as a basis, it can be applied to existing control software. Furthermore, the style supports hierarchical composition of multi-objective optimization solutions. As such, requirements 1 and 5 are satisfied.

Second, using the MO2 style, the different elements of the multi-objective optimization solution (i.e., decision variables, constraints and objective functions) can be connected to those elements in the control software architecture to which they are conceptually related within the domain. For example, a constraint on the power given to a physical component in the system can be connected to the component in the control software architecture that determines and actuates the level of power provided to this physical component in the system. This satisfies requirement 2.

Third, the MO2 style provides the possibility to specify constraints and objective functions on other variables than the decision variables, satisfying requirement 3. Designers can make references from components in the MO2 model to 20-Sim models that specify the semantics of these components. The MO2 toolchain uses these 20-Sim models to create a mathematical relationship between the variables used to specify

constraints and objective functions, and the decision variables. As such, requirement 4 has been satisfied by the MO2 method. A mathematical relationship is necessary for a consistent multi-objective optimization solution. The MO2 toolchain contains a validator to check the existence of the necessary mathematical relationships, thus checking the consistency of the multi-objective optimization solution. This satisfies requirement 6.

Fourth, the MO2 toolchain contains a code generator that generates the implementation of an optimizer from a MO2 model. Furthermore, the toolchain generates instrumentation in the implementation of the control software components, to connect the optimizer to the control software components. This delivers an implementation of multi-objective optimization in embedded control software, satisfying requirement 7.



Experimentation & Validation

In this chapter we validate the techniques proposed in this thesis. First, the chapter describes an experiment to test the performance of a system that applies the MO2 method for optimization. A comparison is made with other systems that use state-of-the-practice engineering solutions for optimization. The results of the experiment are provided and discussed. Next, a qualitative evaluation of the ability of the proposed techniques to manage software complexity in adaptive embedded systems is given. Realistic evolution scenarios are used to evaluate the maintainability and evolvability of software when using state-of-the-practice techniques and when using the techniques proposed in this thesis.

5.1 Experiment: Optimization Performance

We are going to validate the effectiveness of the MO2 method to optimize the performance of a digital document printing system with an experiment. This experiment focuses on the Warm Process case study. To perform the experiment, we created an embedded control software implementation containing multi-objective optimization, using the MO2 method presented in Chapter 4. The techniques to compose physical models with software modules presented in Chapter 2 are also applied in this implementation. The implemented embedded control software will be compared with three other embedded control software implementations for the Warm Process case study. These three other software implementations contain state-of-the-practice engineering solutions to determine the speed of the system.

The four implementations are compared concerning the system quality *productivity*, using an experimental setup that uses a Matlab Simulink model to simulate the printer hardware behavior. In different simulation scenarios, each with a limited and fluctuating amount of available power, the productivity of each of the implementations is measured.

5.1.1 Test System Requirements & Assumptions

To ensure a fair comparison between the four implementations, each implementation should be based on the following assumptions or comply to the mentioned constraint:

- The system quality on which these implementations are compared is *productivity.*
- Each implementation should ensure sufficient print quality. This should be done by controlling $T_{contact}$ to the setpoint determined by the `PrintQuality` physical model (Equation 1.1 on Page 16).
- The speed of the system can vary between 60 and 120 ppm. Furthermore, it is assumed that the speed can be changed instantaneously[1].
- The implementation should ensure that the paper heater temperature (T_{ph}) is controlled to its defined setpoint [2]. The only occasion in which T_{ph} is allowed to deviate from its setpoint is when the speed is 60 and there is not enough power available to maintain print quality. However, the experiments use (randomized) fluctuations in power that are chosen in such a way that this situation is rare.
- The amount of power available fluctuates randomly between an amount of power barely sufficient to print at the minimum speed of 60 ppm and an amount of power sufficient to print at a speed of 120 ppm. Changes in the amount of power available are not continuous, but happen in discrete steps, to simulate the behavior of other devices on the same power supply being turned on and of.

5.1.2 Experiment Setup

The performance of the four control software implementations regarding productivity is measured in an experimental setup. This setup uses a Matlab Simulink model of the Warm Proces thermodynamics, to simulate the Warm Process part of the physical printer system. This Simulink model has been provided by our industrial partner, and is a realistic model of a real printer system. Figure 5.1 shows the setup of this simulation. The Simulink model of the Warm Process is simulated using Matlab Simulink. To execute a certain scenario, the simulation model can be *seeded* with a value for the pseudo-random number generator that determines the amount of power available in the Simulink model. After the simulation, the results concerning productivity can be obtained.

Besides the Simulink model of the Warm Process, one of the four control software implementations is provided. Matlab natively supports a Java virtual machine. Before the simulation starts, an instance of the Java control implementation is created and assigned to a Matlab variable. The instance of the Java control implementation has

[1] In real printing systems, there may be a delay, for example to prevent scheduling problems: scheduled tasks first need to finish before speed can be changed. The impact of such a delay is discussed in Section 5.4.4.

[2] A lower T_{ph} in combination with a higher $T_{contact}$ leads in the simulation model to more efficient printing, so a higher possible productivity. If some of the implementations make use of this characteristics, while others do not, the results concerning productivity are not comparable anymore.

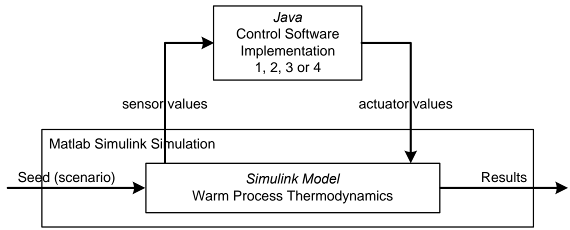


Figure 5.1: Simulation setup

a method that executes one iteration of the control loop. In the Simulink model, a *Matlab function* block is used to call this method. In this call, the current sensor values are provided and the new actuator values are returned. During the simulation, calls to execute the control loop are made with a fixed interval of 0.5 simulated seconds.

With the described setup the following experiments are performed on each of the four control software implementations:

- Fluctuating power supply: Experiments with a fluctuating but limited amount of power available are performed. The amount of power available fluctuates between an amount sufficient to print at the highest speed ($maxP$) and an amount barely sufficient to print at lowest speed ($minP$).
  - Experiments are performed for the following 7 different fluctuation intervals (i.e., the interval after which the amount of available power changes): 10 s, 25 s, 50 s, 100 s, 250 s, 500 s, 1000 s. Different power fluctuation intervals are used, to investigate whether the results differ between slower and faster changes in the amount of power available.
  - For each fluctuation interval, 20 different scenarios are tested. Each scenario is created using a pseudo-random generator which provides an even distribution between $minP$ and $maxP$.
- Fixed power supply: 11 scenarios with a fixed amount of power available are tested. Between these 11 scenarios, the amount of power available varies in fixed steps of 50 W from 700 W ($minP$) to 1200 W ($maxP$).

For each tested scenario, the simulation runs for 20000 time steps (i.e., simulated seconds).

5.2 Experiment: Control Software Implementations

The following four control software implementations are created (one has two variants):

1. **MO2 Implementation:** This implementation contains a solution for multi-objective optimization. To design the multi-objective optimization solution, the

MO2 method, presented in Chapter 4, is used. Also, the approach presented in Chapter 2 to compose the physical models with software modules is applied. The software modules are implemented in the GPL *Java*.

2. **Intelligent Speed Implementation:** This implementation contains a state-of-the-practice engineering solution to optimize productivity that we have observed in practice: speed is increased or decreased based on the difference between the amount of power available and the total amount of power consumed. Two variants of the Intelligent Speed algorithm are implemented.

3. **Eco Mode Implementation:** This implementation contains a naive solution to cope with a limited amount of power available: print at maximum speed (120 ppm) when there is sufficient power available, or in eco-mode (60 ppm) when there is insufficient power available to print at maximum speed.

Details of these four control software implementations are explained next.

5.2.1 Implementation 1: Multi-Objective Optimization

MO2 Model

The first implementation applies the MO2 method to design multi-objective optimization in the architecture of the embedded control software. Figure 5.2 shows the created MO2 model.

This MO2 model is based on the MO2 model explained in Example 4.5. However, there are some differences. The first difference is the addition of the oblivious component `Trade-off`. This component implements a function that calculates a trade-off value for the system objectives *power consumption* and *productivity*. The outcome of the trade-off function is provided on an out-port of the `Trade-off` component. This out-port is the only objective port in the architecture. By adding a trade-off function, multi-objective optimization results in a single optimal value for the decision variables. The weight of this trade-off functions is chosen in such a way that productivity is optimized.

The second difference is that the variable T_{ph}^{sp} is not a decision variable anymore, because we are going to apply instantaneous optimization. Having both T_{ph}^{sp} and v as a decision variable leads to problems when instantaneous optimization is applied, instead of optimization over time. For a detailed explanation of these problems, refer back to Section 4.7.1. Furthermore, one of the implementation requirements mentioned in Section 5.1.1 is to maintain T_{ph} at a fixed setpoint. Because T_{ph}^{sp} is not a decision variable in this MO2 model, there are also no constraints on the T_{ph}^{sp} port and the P_{ph} port of the `PaperHeaterController` component.

Some of the components in the MO2 model reference a 20-Sim model. The components that reference a 20-Sim model are `PrintQuality`, `RadiatorController`, `Power`, `Productivity` and `Trade-off`. Listings 5.1 to 5.5 show the SIDOPS+ specifications that correspond to the referenced 20-Sim models[3].

[3]Because of confidentiality reasons, we are not allowed to show the constant values. For the constants to which this applies, the value in the SIDOPS+ specification is replaced with the text `#some value#`.

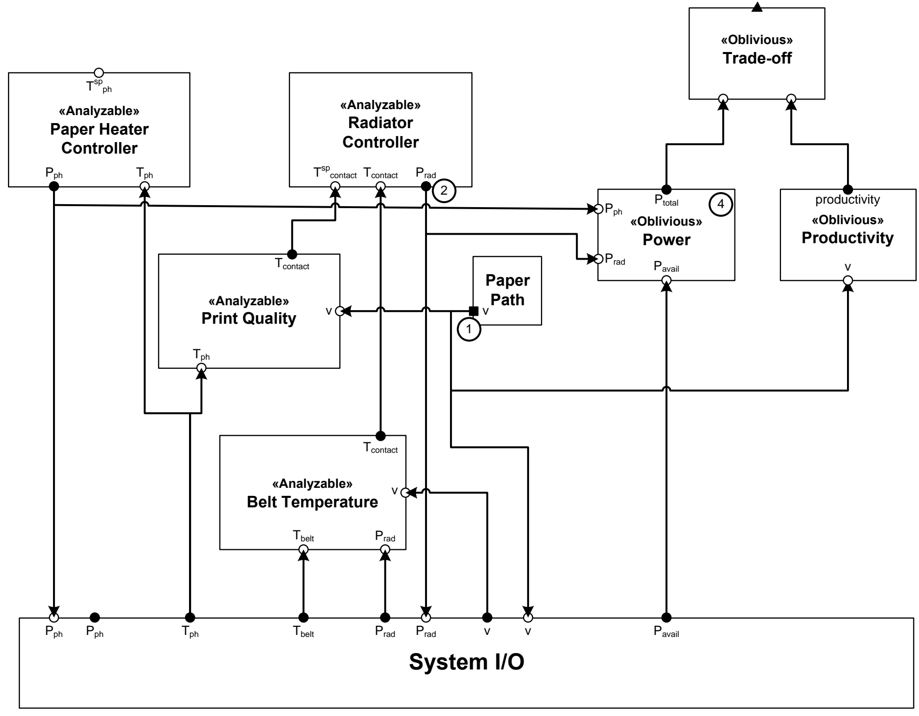


Figure 5.2: MO2 Model of the Warm Process case study

```
1  constants
2    real c1 = #some value#;
3    real c2 = #some value#;
4    real c3 = #some value#;
5  variables
6    real global Tcontact;
7    real global Tph;
8    real global v;
9  equations
10   Tcontact = c1*v - c2*Tph + c3;
```

Listing 5.1: SIDOPS+ specification referenced by the `PrintQuality` component

```
1  constants
2    real Kp_rad = #some value#;
3    real Ki_rad = #some value#;
4  variables
5    real global Tcontact;
6    real global TcontactSP;
7    real global Prad;
8  equations
9    Prad = Kp_rad * (TcontactSP - Tcontact) + Ki_rad * int(TcontactSP -
          Tcontact);
```

Listing 5.2: SIDOPS+ specification referenced by the `RadiatorController` component

```
1 variables
2   real global Pph;
3   real global Prad;
4   real global Ptotal;
5   real global Pavailable;
6   real global Pmargin;
7 equations
8   Ptotal = Pph + Prad;
9   Pmargin = Pavailable - Ptotal;
```

Listing 5.3: SIDOPS+ specification referenced by the `Power` component

```
1 variables
2   real global v;
3   real global Productivity;
4 equations
5   Productivity = 1/v;
```

Listing 5.4: SIDOPS+ specification referenced by the `Productivity` component

```
1 variables
2   real global tradeoff;
3   real global Productivity;
4   real global Ptotal;
5   real global weight;
6 equations
7   tradeoff = power(Productivity, weight) + power(Ptotal, 1-weight);
```

Listing 5.5: SIDOPS+ specification referenced by the `Trade-off` component

Listing 5.5 shows the specification of a trade-off function, that uses the variable $weight$ to adapt the trade-off between $Productivity$ and P_{total} (total power consumption. The value for $weight$ can vary between 0 and 1. A higher value for $weight$ puts more emphasis on the objective $Productivity$, a lower value for $weight$ puts more emphasis on the objective P_{total}. For the experiments, $weight$ is given the value 1, i.e., putting full emphasis on $Productivity$.

Control Software Structure

Figure 5.3 shows the software structure of the multi-objective optimization control implementation. This structure contains the following artifacts:

Optimizer

This is a software module that is generated by the MO2 toolchain from the MO2 model shown in Figure 5.2. The optimization algorithm that is used is the Matlab function `fmincon` [4]. The `Optimizer` module consists of a generated Java class and a generated Matlab `.m` file. The Java class contains logic to obtain necessary information from the other software modules (as shown by the generated data-flow edges). The generated Matlab file contains a function that is called from the Java class using a Java API provided with Matlab. The values obtained by the Java class from other software modules are passed as a parameter to the generated Matlab function. The generated Matlab function constructs the constraints and objective function, and calls `fmincon` to perform the optimization. The generated code is provided in Appendix E.

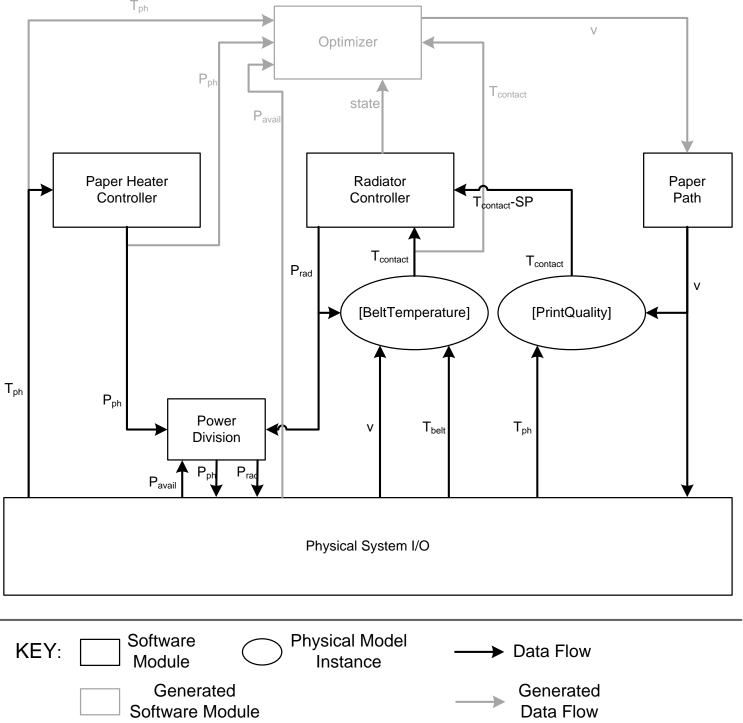


Figure 5.3: Software structure of the Multi-Objective Optimization Control implementation

PaperHeaterController
Java class implementing the control logic (PI controller) to control the temperature of the paper heater to setpoint. This class is reused in all four investigated software implementations.

RadiatorController
Java class implementing the control logic (PI controller) to control $T_{contact}$ to the provided setpoint.

BeltTemperature
Physical model instance of the `BeltTemperature` physical model. The corresponding SIDOPS+ specification is shown in Listing 5.6.
Listing 5.7 shows the composition filters specification for the `BeltTemperature` physical model instance. This composition filter specification shows the definition of one filter that matches *CheckUpdate* events for the variables v and T_{belt}. Messages are dispatched to `PhysicalSystemIO` to obtain the values for these variables. The `RadiatorController` uses the base interface of `BeltTemperature` to obtain $T_{contact}$ and to update P_{rad}.

```
1  constants
2    real c4 = #some value#;
3  variables
4    real global Tcontact;
5    real global Prad;
6    real global v;
7    real global Tbelt;
8  equations
9    Tcontact = c4 *Prad
10                / sqrt(v) + Tbelt;
```

Listing 5.6: SIDOPS+ specification of the `BelTemperature` physical model

```
1  concern BTConcern in mooprinter
2  {
3    filtermodule BTModule
4    {
5      externals
6        io : PhysicalSystemIO = ...;
7      outputfilters
8        ioFilter: Dispatch = (event.variableName=='v' &
           event.eventType=='CheckUpdate') {target=io;
           selector='getV';}
9          cor (event.variableName=='Tbelt' &
             event.eventType=='CheckUpdate') {target=io;
             selector='getTbelt';}
10   }
11
12   superimposition
13   {
14     selectors
15       pmi = {C | isModelWithName(C, 'BeltTemperature')};
16     filtermodules
17       pmi <- BTModule;
18   }
19 }
```

Listing 5.7: Composition filters specification for the `BelTemperature` physical model instance

PrintQuality

Physical model instance of the `PrintQuality` physical model. The corresponding SIDOPS+ specification is shown in Listing 5.1. Note that in this case the SIDOPS+ specification is reused from the architectural modeling phase, as the specification is also referenced from the MO2 model shown in Figure 5.2.

Listing 5.8 shows the composition filters specification for the `PrintQuality` physical model instance. This composition filter specification shows the definition of two filters. The first filter matches *CheckUpdate* events for the variables T_{ph} and v, and dispatches messages to `PhysicalSystemIO` and `PaperPath` to obtain the values. The second filter matches *Change* events for the variable $T_{contact}$ and dispatches a message to `RadiatorController` to set the new $T^{sp}_{contact}$.

```
1  concern PQConcern in mooprinter
2  {
3    filtermodule PQModule
4    {
5      externals
6        io : PhysicalSystemIO = ...;
7        pp : PaperPath = ...;
8        rad : RadiatorController = ...;
9      outputfilters
10       ioFilter: Dispatch = (event.variableName=='Tph' &
             event.eventType=='CheckUpdate') {target=io;
             selector='getTph';}
11         cor (event.variableName=='v' &
               event.eventType=='CheckUpdate') {target=pp;
               selector='getV';};
12       spFilter: Dispatch = (event.variableName=='Tcontact' &
             event.eventType=='Change') {target=rad;
             selector='setTcontactSP';}
13   }
14
15   superimposition
16   {
17     selectors
18       pmi = { C | isModelWithName(C, 'PrintQuality') };
19     filtermodules
20       pmi <- PQModule;
21   }
22 }
```

Listing 5.8: Composition filters specification for the `PrintQuality` physical model instance

PaperPath
Provides the speed. This speed is modified by the `Optimizer` component.

PowerDivision
This Java class limits the actual P_{ph}, in case the total amount of power required ($P_{ph} + P_{rad}$) is larger than the amount of power available. To do this, `PowerDivision` uses the following equation:

$$P_{ph} = min(P_{ph}, P_{avail} - P_{rad})$$

In this way, `PowerDivision` ensures that the system cannot consume more power than is available. This class is reused in all four investigated software implementations.

PhysicalSystemIO
Interface to the sensors and actuators in the physical system. This class is reused in all four investigated software implementations.

5.2.2 Implementation 2: Intelligent Speed

Instead of applying multi-objective optimization, the second implementation uses a different algorithm to optimize productivity. This algorithm, which we call *Intelligent Speed*, is an engineering solution that is state-of-the-practice, as we have witnessed at our industrial partner.

Algorithm 5.1: Intelligent speed algorithm

```
1  P_total := P_ph + P_rad
2  P_margin := P_avail − P_total
3  if P_margin ≤ 0 then
4  |  v_new := v − 20
5  end
6  else if P_margin ≥ 200 then
7  |  v_new := v + 0.05 ∗ P_margin
8  end
9  else
10 |  v_new := v
11 end
   // Ensure that new speed is within limits:
12 v_new := min(120, max(60, v_new))
```

Algorithm 5.1 shows the Intelligent Speed algorithm. This algorithm works as follows. The amount of power that is not utilized (P_{margin}) is calculated. When P_{margin} it too low, speed is decreased with 20 ppm. When P_{margin} is higher than a certain boundary (200 W), then speed is increased with an amount that is proportional to the actual value of P_{margin}.

Note that this algorithm does not optimize the power margin P_{margin} to 0 W, but maintains a certain amount of margin, which varies between 0 W and 200 W. This margin is used in real printer systems to cope with a delay in which speed can be changed; in this experiment we assume that speed can be changed instantaneously, but in real printer systems there is a delay of a number of seconds before speed can be adapted. To cope with sudden drops in the amount of power available during this delay period, the algorithm maintains a margin.

In the experiment, such a margin results in lower productivity, as not all available power is utilized. To make a fair comparison of the effectiveness of the MO2 implementation, which optimizes for a power margin (P_{margin}) of 0 W, we also test a second, adapted implementation of the Intelligent Speed algorithm, called *Intelligent Speed 2*. Algorithm 5.2 shows this algorithm. This algorithm adapts speed to optimize P_{margin} to approximately 0 W. Note that the algorithm takes a small margin of up to 10 W into account, to enable the algorithm to reach a stable speed.

Control Software Structure

Figure 5.4 shows the control software structure for the Intelligent Speed implementation. Modules `PaperHeaterController`, `PowerDivision` and `PhysicalSystemIO` are the same as in the MO2 implementation.

RadiatorController

Java class implementing the control logic (PI controller) to control $T_{contact}$ to the provided setpoint, and the physical characteristics of print quality and belt temperature, as specified by Equation 1.1 (Page 16) and Equation 1.2 (Page 16)

Algorithm 5.2: Intelligent speed algorithm 2

```
1  P_total := P_ph + P_rad
2  P_margin := P_avail − P_total
3  if P_margin ≤ 0 then
4  |  v_new := v + 0.05 * P_margin
5  end
6  else if P_margin ≥ 10 then
7  |  v_new := v + 0.05 * P_margin
8  end
9  else
10 |  v_new := v
11 end
   // Ensure that new speed is within limits:
12 v_new := min(120, max(60, v_new))
```

respectively. Listing 2.1 on Page 27 shows a code fragment of the implementation of the `RadiatorController`.

IntelligentSpeed
Java class implementing the Intelligent Speed algorithm.

5.2.3 Implementation 3: Eco Mode

The last implementation we consider uses an 'old-fashioned' two speed model: if the amount of available power is sufficient to print at the highest speed (120 ppm), then printing is performed at the highest speed. Otherwise, printing is performed at lowest

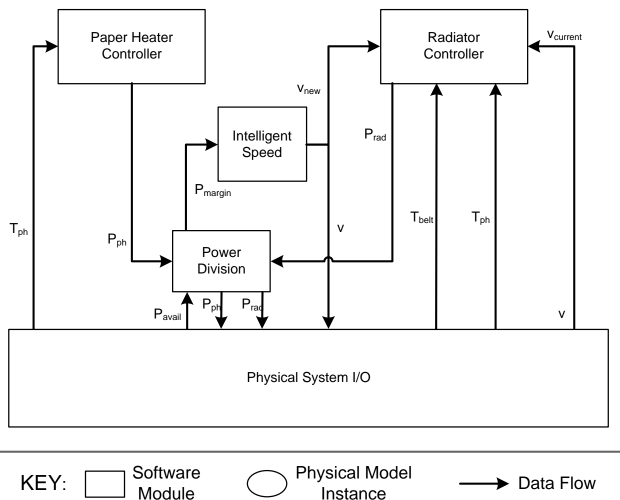


Figure 5.4: Software structure of the Intelligent Speed Control implementation

possible speed (60 ppm). This implementation is called *Eco Mode* implementation, because the low-speed mode consumes less energy. Algorithm 5.3 shows the Eco Mode algorithm.

Algorithm 5.3: Eco mode algorithm

1 $P_{rad:high-speed}$:= Amount of radiator power needed to print at 120 ppm
2 $P_{total} := P_{ph} + P_{rad:high-speed}$
3 **if** $P_{total} \leq P_{avail}$ **then**
4 | $v_{new} := 120$
5 **end**
6 **else**
7 | $v_{new} := 60$
8 **end**

Control Software Structure

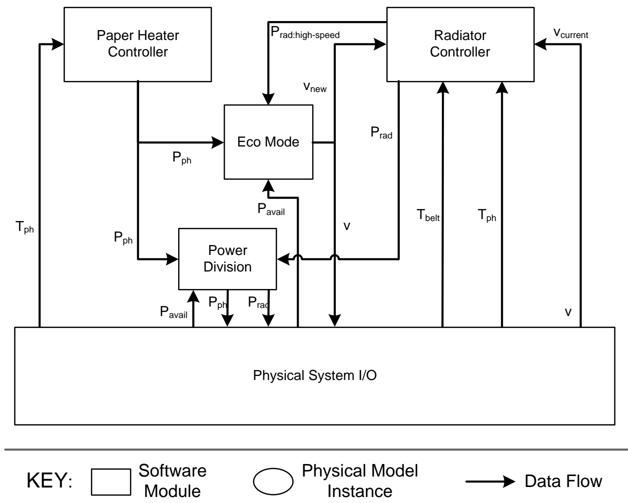


Figure 5.5: Software structure of the Eco Mode Control implementation

Figure 5.5 shows the control software structure for the Eco Mode implementation. Modules `PaperHeaterController`, `PowerDivision` and `PhysicalSystemIO` are the same as in the MO2 implementation.

RadiatorController
: Extension of the `RadiatorController` class in the Intelligent Speed implementation. In this implementation, the class can also provide the amount of power necessary if the speed is set at 120 ppm.

EcoMode
: Java class implementing the Eco Mode algorithm.

5.3 Experiment: Results

This section presents the results of the experiment. A detailed evaluation and discussion of the results is given in Section 5.4.

5.3.1 Main Results

The four different implementations are compared according to the following four criteria:

- Productivity: The average speed of the system with the given control software implementation, in pages per minute (*ppm*).
- Energy consumption: The average energy consumed per printed page is provided, in $J/page$.
- Power margin: The average power margin in W. The power margin is the difference between the amount of power available and the amount of power consumed.
- Print Quality: A measurement of average deviation from perfect print quality is given. Perfect print quality is defined as: the temperatures and speed of the system are in an ideal relationship with each other, as defined by the `PrintQuality` physical model. But in reality tiny deviations occur because there is a control latency, which means that it takes some time for the control logic to control a variable to its setpoint.

Figure 5.6, 5.7, 5.8 and 5.9 show the results of the experiment for each of these four criteria. Each figure shows the results of one criterion. For each of the seven power fluctuation intervals the mean value over the 20 scenarios is shown.

Figure 5.6 shows the results for the criterion *productivity*. The figure shows that for lower power fluctuation intervals, the MO2 implementation performs significantly better than the other three control implementations. For the MO2, Eco Mode and Intelligent Speed implementation, the performance is stable for the different power fluctuation intervals. However, the Intelligent Speed 2 implementation performs better with higher power fluctuation intervals, giving almost the same performance as the MO2 implementation for the highest two power fluctuation intervals (500 s and 1000 s).

Figure 5.7 shows the results for the criterion *energy consumption*. The figure shows that the MO2 implementation performs significantly better than the Eco Mode and Intelligent Speed implementation. The Intelligent Speed 2 implementation performs better with higher power fluctuation intervals.

Figure 5.8 shows the results for the criterion *power margin*. The figure shows a high average power margin for the Intelligent Speed implementation, as expected from the functioning of the algorithm. The Eco Mode implementation also shows a significant average power margin. The average power margin of the Intelligent Speed 2 implementation is lower for higher power fluctuation intervals. The MO2 implementation has a low power margin. Note that the power margin of the MO2

implementation is not necessarily 0, because there are situations in the scenarios when their is more power available than needed to print at the highest possible speed.

Figure 5.9 shows the results for the criterion *print quality*. The figure shows that the Eco Mode implementation performs significantly worse than the other three implementations. The MO2 implementation performs better than the other implementations, except for the lowest power fluctuation interval.

Figure 5.10 shows the average speed (i.e., productivity) of the four implementations for the 11 scenarios with a constant amounts of power available within each scenario. The figure shows that with a constant amount of power available, the productivity of the Intelligent Speed 2 implementation is almost the same as the productivity of the MO2 implementation. The Intelligent Speed and Eco Mode implementation perform less.

5.3.2 Productivity in Detail

In this section we provide detailed results concerning the *productivity* criterion.

Figure 5.11 shows the average speed (i.e., productivity) of the four implementations for the 20 test scenarios with a fluctuation interval of 1000. The figure shows that the MO2 implementation performs consistently better than the Eco Mode and Intelligent Speed implementations. The performance of the Intelligent Speed 2 implementation is consistently almost the same as the performance of the MO2 implementation.

To provide insight in the speed behavior of the four implementations, Figure 5.12 and 5.13 show for one scenario with a power fluctuation interval of 1000 s the speed of the printing system at each moment in the simulation. Figure 5.12 shows the speed of the system for the MO2, Intelligent Speed and Intelligent Speed 2 implementations. Figure 5.13 shows the speed of the system for the Eco Mode implementation. The results of the Eco Mode implementation are shown separately, because these results would obfuscate the results of the other three implementations. The figures show that the MO2, Intelligent Speed and Intelligent Speed 2 implementations have smooth speed transitions after a power change and reach stable situations. The Eco Mode implementation has very instable speed behavior: the system accelerates and decelerates continuously.

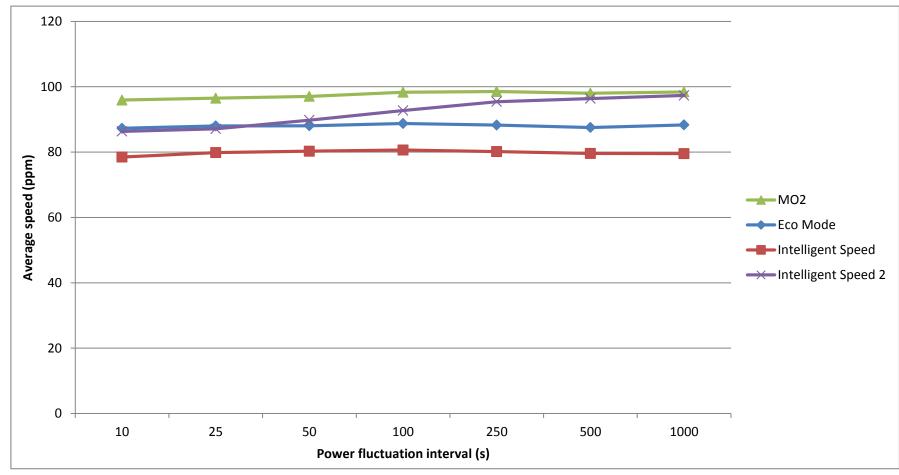


Figure 5.6: Average printing speed

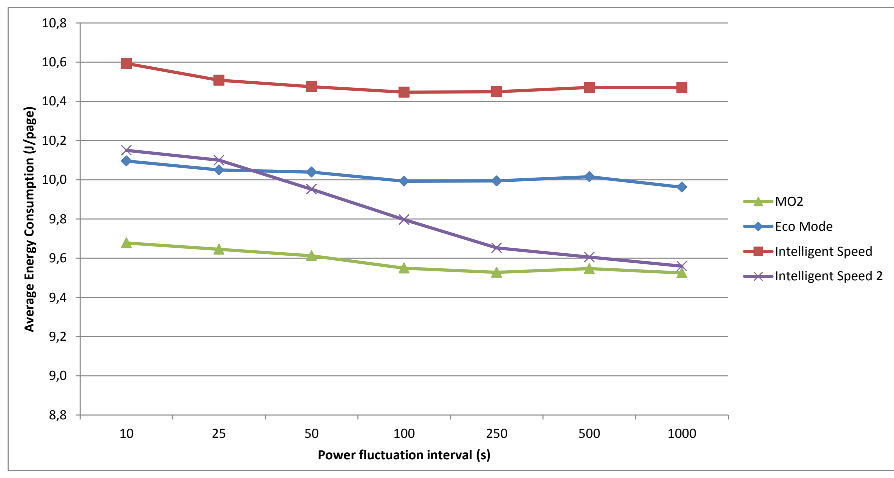


Figure 5.7: Average energy consumption (lower ⇒ better)

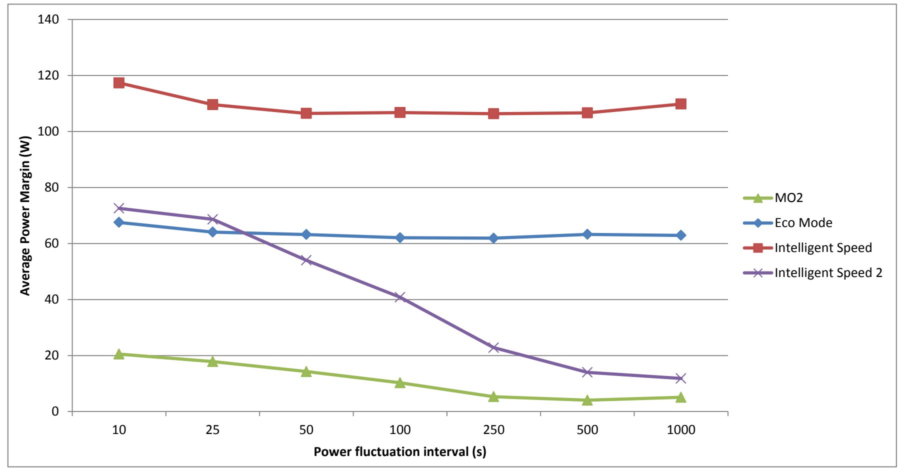


Figure 5.8: Average power margin (lower ⇒ better)

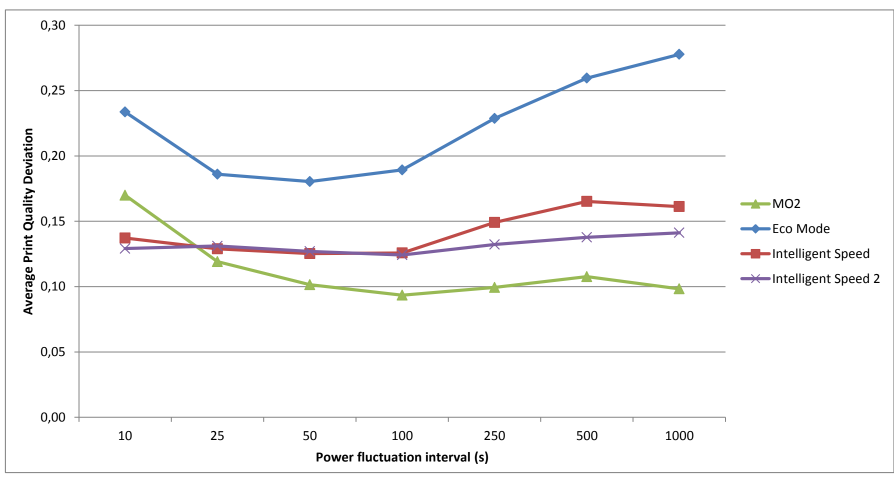


Figure 5.9: Average deviation from print quality (lower ⇒ better)

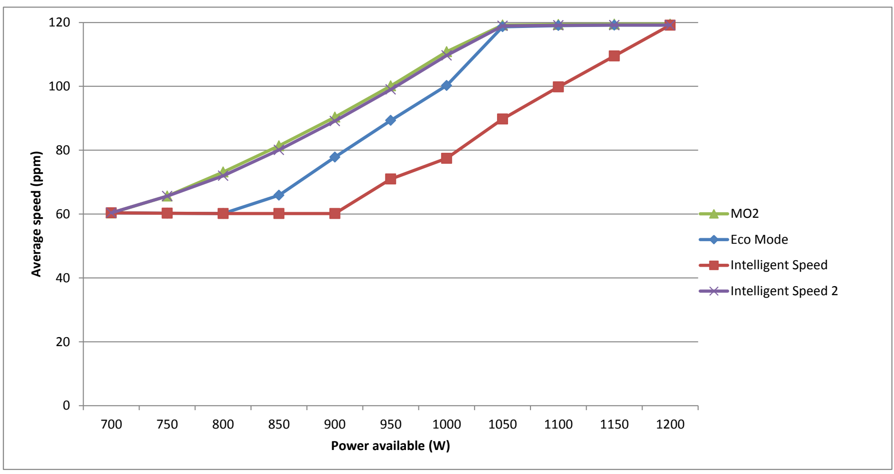


Figure 5.10: Performance of the four implementations concerning productivity with a constant power supply

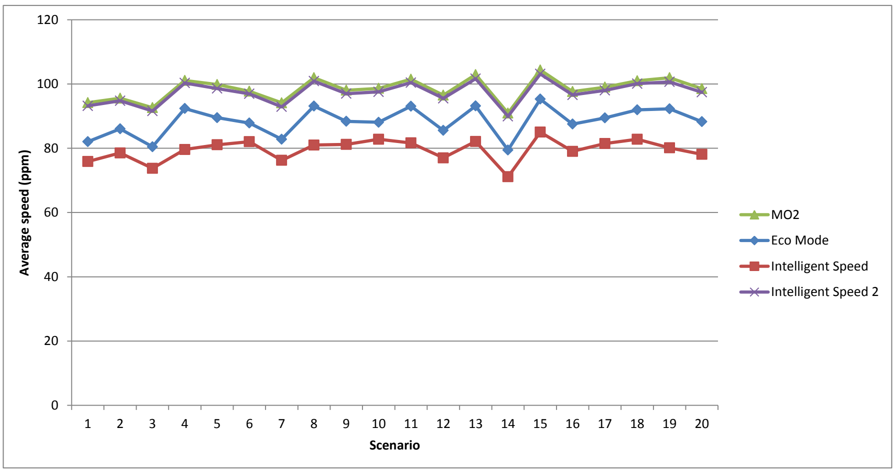


Figure 5.11: Performance of the four implementations concerning productivity for 20 scenarios with a fluctuating power supply

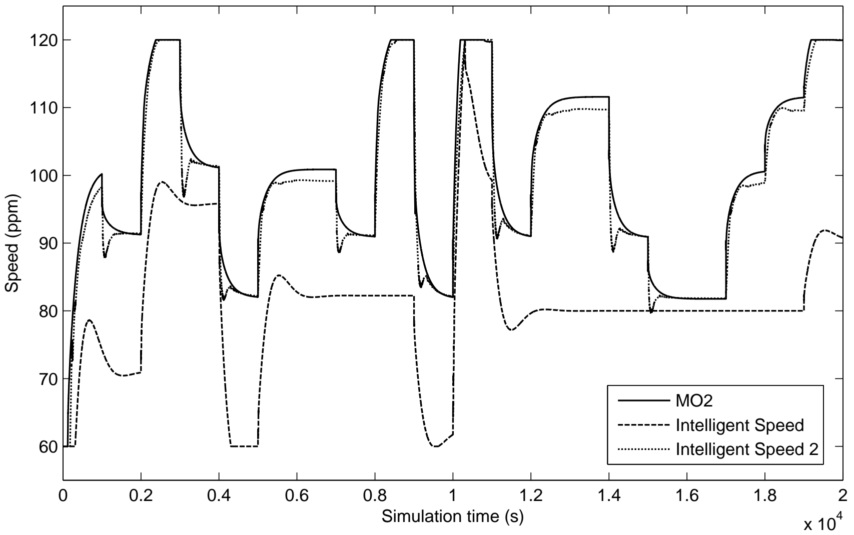


Figure 5.12: Speed in one given scenario with a fluctuating power supply

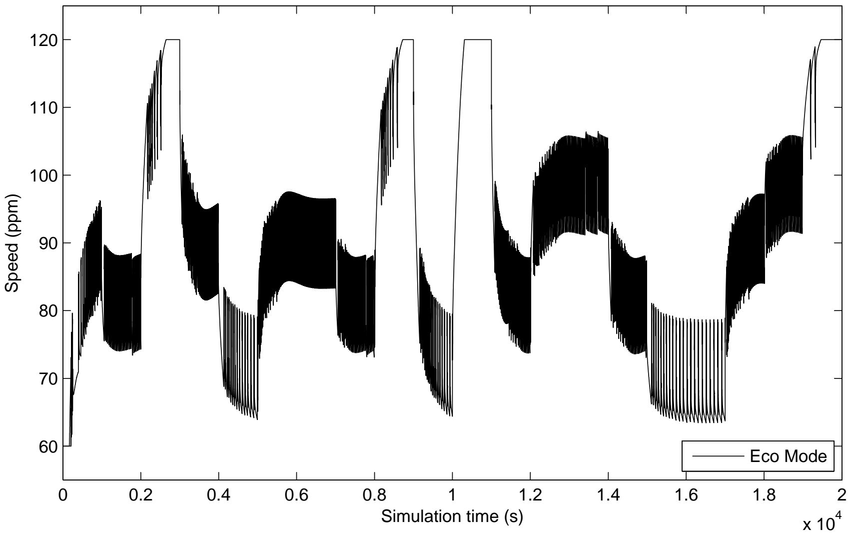


Figure 5.13: Speed in the given scenario for the Eco Mode implementation

5.4 Experiment: Evaluation & Discussion

5.4.1 Best Performance for the MO2 Implementation

Figures 5.6 and 5.10 shows that MO2 implementation has the best performance concerning productivity in all tested scenarios. The difference with the Intelligent Speed and Eco Mode implementations is considerable, but this is mainly due to the fact that these implementations have large power margins. The Intelligent Speed 2 implementation approaches the performance of the MO2 implementation for larger power fluctuation intervals, as it minimizes the power margin. For smaller power fluctuation intervals, the Intelligent Speed 2 implementation is not able to perform at the same level as the MO2 implementation.

Also on the other three criteria, energy consumption, power margin and print quality, the MO2 implementation performs equally well or better than the other three implementations. So, we can conclude that the MO2 method results in systems that are able to function at the same level or a higher level than systems applying other engineering solutions to optimize a system quality. Additionally, the MO2 method provides the ability to optimize multiple system qualities and dynamically make trade-offs between them. In this way, the MO2 method differs from the other solutions, which typically optimize for a single system quality.

5.4.2 Instable Eco Mode Implementation

Figure 5.13 shows a very instable speed behavior of the Eco Mode implementation. The reason for this is that the algorithm is very opportunistic; whenever there is just sufficient amount of power available to go to maximum speed, the algorithm increases the speed to maximum. But quite often, this proves to be too opportunistic, leading to a decrease of the speed to 60 ppm shortly after the increase to 120 ppm. The result is very nervous printer behavior: a system that accelerates and decelerates continuously. This behavior is undesirable from a user experience point of view and it increases wear-and-tear of the printer system. Also, the frequent speed changes lead to more deviation in print quality. Therefore, the Eco Mode implementation performs worst with respect to print quality, as can be seen in Figure 5.9. Compared to the Eco Mode implementation, the behavior of the other three implementations concerning speed changes can be considered smooth, as shown in Figure 5.12, and are thus preferable with respect to user experience and expected wear-and-tear.

5.4.3 Power Margins

The Intelligent Speed algorithm is designed to maintain a certain margin between the amount of power available and the amount of power consumed, as Figure 5.8 also shows. As explained in Section 5.2.2, this power margin is used in real printer systems that have a delay in which speed can be changed. In these printing systems, the power margin is used to handle sudden drops in the amount of power available during the delay in which speed can be changed.

Figure 5.14 shows the power margin of the Intelligent Speed algorithm during one simulated scenario. The figure shows that the power margin is not constant, but

varies between 0 W and 200 W. This is inherent in the design of the Intelligent Speed algorithm (Algorithm 5.1 on Page 178): the speed is increased when there is more than 200 W power margin and decreased when there is less than 0 W power margin. A problem with this design is that sometimes there is a low power margin available, making it harder to cope with sudden drops in the amount of power available. At other times there is a too high power margin available, which reduces productivity.

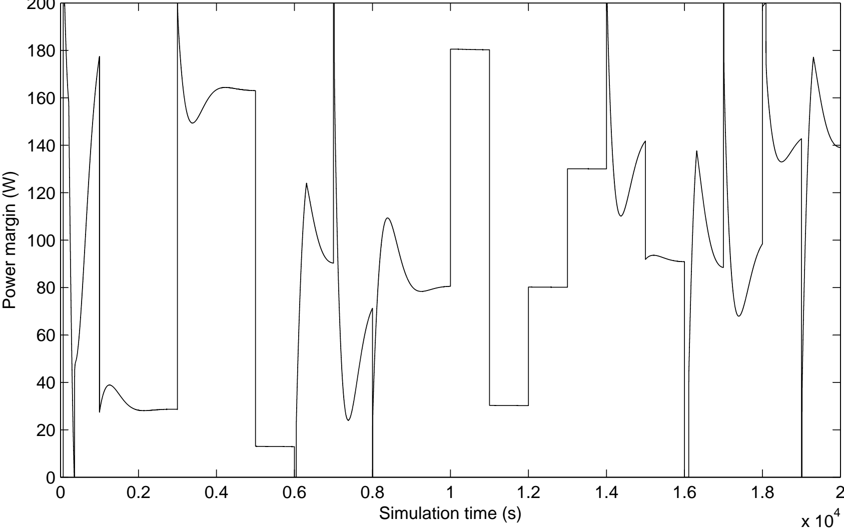


Figure 5.14: Power margin during one scenario for the Intelligent Speed implementation

With the MO2 method it is also possible to take a certain power margin into account. This is done by changing the constraint on the `Power` component (labeled 4) in the MO2 model shown in Figure 5.2 on Page 173. Currently, the constraint on the `Power` component is:

$$P_{total} \leq P_{avail}$$

This constraint does not take a power margin into account. To take a power margin of 100 W into account, the constraint is changed to the following:

$$P_{total} + 100 \leq P_{avail}$$

Figure 5.15 shows the power margin of the adapted MO2 implementation for the same scenario as was used to demonstrate the power margin of the Intelligent Speed implementation in Figure 5.14. Figure 5.15 demonstrates that the MO2 method is able to provide a precise and stable power margin of 100 W, as opposed to Intelligent Speed algorithm which gives an unpredictable power margin between 0 W and 200 W. The short peaks and drops visible in Figure 5.15 are caused by large changes in the amount of power available that the system cannot directly adapt to. The two

longer drops in the power margin (around $0.65 * 10^4$ s and $1.65 * 10^4$ s) are caused by the fact that during this period there is not enough power available to print at the lowest speed and maintain a 100 W power margin. In this case, the power margin is used to continue printing at the lowest speed.

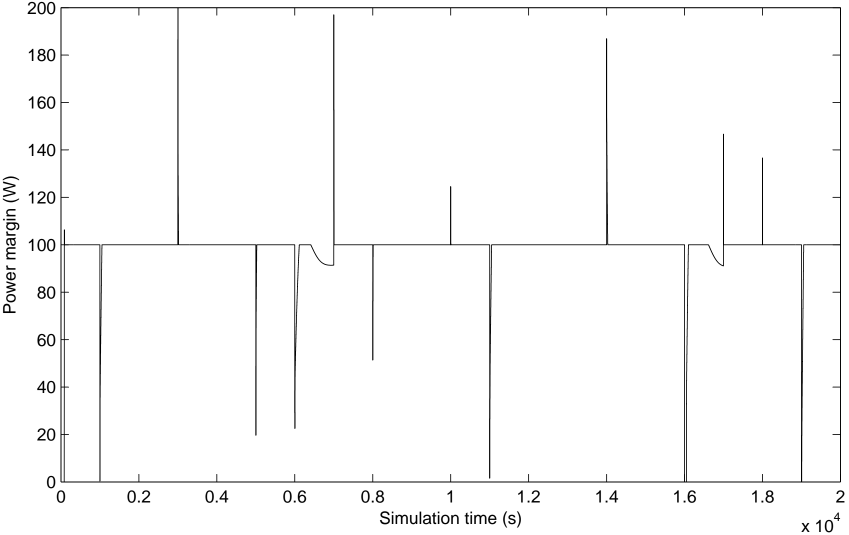


Figure 5.15: Power margin during the same scenario for the MO2 implementation with 100 W margin

5.4.4 Delay in Speed Changes

In the experiment we assumed that the speed of the printing system can be changed instantaneously. However, in real printing systems there might be a delay between when a speed change is requested and when the speed is actually changed. This delay exists, for example, to prevent scheduling problems; scheduled tasks first need to finish before speed can be changed. To cope with possible changes in circumstances during the delay, certain margins are taken into account. One example of such a margin is the power margin (see Section 5.4.3), to be able to cope with drops in the amount of power available during the delay. When such margins are taken into account, the result is lower performance of the system, as not all available resources are utilized.

Future printing systems are going to implement more flexible scheduling algorithms [56]. Such algorithms can reduce or eliminate the delay to change speed, lowering the need to take margins into account. This will result in a better performing system.

5.4.5 Reverse Correlation between Productivity and Energy Consumed

Figures 5.6 and 5.7 suggest that there is a reverse correlation between productivity and energy consumed per printed page. This reverse correlation can be explained from the fact that the printer loses an amount of energy to the environment which is unrelated to the speed of the system. If productivity is lower, the amount of energy lost to the environment is attributed to less printed pages, leading to a higher energy consumption per printed page.

5.5 Qualitative Evaluation using Evolution Scenarios

In this section, we evaluate qualitatively, using a number of realistic evolution scenarios, the ability of the proposed techniques to manage the complexity of embedded control software that implements algorithms for adaptive behavior. The evolution scenarios provide situations in which the embedded control software needs to be modified. The impact of the modifications are qualitatively evaluated with respect to development effort. In this evaluation, a comparison is made between software applying state-of-the-art techniques and software applying the techniques proposed in this thesis.

The evolution scenarios are classified in three different types based on the type of changes in the system. Each evolution scenario gives a description of the scenario, an explanation of the consequences of the scenario for the embedded control software and a summary of the development impact. The summary of the development impact describes which development actions are performed and a classification of the impact of each of these actions. The following classification of impact is used:

- **Low impact(L):** Changes are made within domain-specific abstractions.

- **Medium impact(M):** Changes are made in implementation abstractions instead of domain-specific abstractions. Implementation abstractions are less comprehensible than domain-specific abstractions. Furthermore, the implementation abstractions may be tangled with other computational logic. These issues increase development effort, and are therefore classified as medium.

- **High impact(H):** Multiple scattered changes are necessary. When an evolution scenario causes many changes at multiple locations in the software, this has a severe impact on development effort: the affected locations in the implementation need to be identified and consistent changes need to be made. Such changes are also vulnerable to errors, as certain code locations that need to be changed might be missed, or changes may be inconsistent. Therefore, such changes are classified as having a high impact on development effort.

5.5.1 Evolution Scenario Type 1: Change in Physical Characteristics

It is likely that during the lifetime of a family of embedded system (*product family*) changes are made in the design of the system, or the system is used in different circumstances. Such changes have an impact on the physical models implemented in embedded control software. Next, several examples of evolution scenarios are given that result in changes in the physical models. These scenarios refer to the example implementation of the Warm Process case study given in Example 2.11 on Page 70.

Evolution Scenarios

Evolution Scenario 1.1
Change in System Components or Structure

Description
During the lifetime of a product family, it is likely that certain components in the system are replaced by other components that are more efficient or cheaper to produce, or that certain components are removed from the system because they are redundant. Also, the structure and location of the components in the system is subject to change. An example of this is the removal of the paper heater in the Warm Process case study, introduced in Section 1.3.1. In this case, heating will be entirely done by the radiator.

Consequences for the software implementation
Components in the system often have a direct correspondence with modules that implement control logic. For example, the paper heater has corresponding control logic, which is implement in the `Paper Heater Controller` module. So, adding or removing a component from the system can lead to the addition or removal of a module. Furthermore, changing system components or changing the structure of the system has an impact on implemented physical models that relate to the component. For example, the removal of the paper heater has an impact on the physical model that represents print quality (Equation 1.1). The implementation of the physical model needs to be located in the embedded control software and updated to the new situation.

Summary of Development Impact

State-of-the-art techniques:

- **M:** Identify and change in the GPL the implementation of the affected physical model.

Proposed techniques:

- **L:** Change the affected physical model.

Evolution Scenario 1.2
Different Operating Conditions

Description

Certain operating conditions of an embedded system can change from being fixed to being variable. For example, because the system is targeted at a different market with another climate, or is applied in different situations than initially expected.

An example of such a change is the introduction of different paper weights. At first, the printing system was designed to handle only one paper weight. But later on, to target new customers, the system is adjusted to handle different paper weights.

Consequences for the software implementation

If a system is used in different operating conditions, the physical models implemented in software may change. For example, physical variables that were assumed to be constant, and as such are abstracted away using constants in the physical model, may now be variable in the system and need to be taken into account.

For example, introducing variable paper weight means that the variable m_{paper} (mass of the paper) now becomes important in the `PrintQuality` physical model (Equation 1.1). Before, this variable was abstracted away in the constant c_2, as m_{paper} was constant. But now that m_{paper} is variable, it needs to be explicitly represented in the physical model. The new physical relationship is:

$$T_{contact} = c_1 \cdot v - \frac{c_2' \cdot T_{ph}}{\sqrt{m_{paper}}} + c_3$$

This physical relationship needs to be identified in the software implementation and updated accordingly.

Summary of Development Impact

State-of-the-art techniques:

- **M:** Identify and change in the GPL the implementation of the affected physical model.

Proposed techniques:

- **L:** Change the affected physical model.

Evolution Scenario 1.3
More (or Less) Available Information

Description
During the lifetime of a product family, sensors are added or removed from the system. A sensor might be added to provide more information, so that the system can be controlled more effectively. A sensor might be removed to decrease production costs.
For example, a sensor that measures temperature of the environment can be added to the system. This sensor provides additional information to the control software, with which the `PrintQuality` and `BeltTemperture` physical models (Equations 1.1 and 1.2) can be refined.

Consequences for the software implementation
If the available information from sensors changes, the implemented physical models need to be adjusted to cope with these changes. Furthermore, it may be possible that the interaction of the implemented physical models with other code that provides the sensor values needs to be updated.
For example, if a sensor is added to measure environment temperature, this value can be used in the physical relationships for print quality and belt temperature, as follows:

$$T_{contact} = c_1 \cdot v - c_2 \cdot T_{ph} + c_3 + c_5 \cdot T_{environment}$$

$$T_{contact} = c_4 \cdot \frac{P_{rad}}{\sqrt{v}} + T_{belt} + c_6 \cdot T_{environment}$$

The implementations of these relationships need to be located in the software and updated. Furthermore, additional interaction with `PhysicalSystemIO` should be implemented, to obtain $T_{environment}$.

Summary of Development Impact

State-of-the-art techniques:

- **M:** Identify and change in the GPL the implementation of the affected physical model.
- **M:** Change the interaction in the GPL to obtain additional information.

Proposed techniques:

- **L:** Change the affected physical model.
- **L:** Implement a composition filter for additional interaction.

Qualitative Evaluation

The evolution scenarios mainly show changes in implemented physical models.

State-of-the-art The application of a GPL to implement physical models leads to the following disadvantages concerning the evolution scenarios:

- The affected physical relationships may be hard to locate and identify, as they are implemented using implementation abstractions and as they may be tangled with software modules that implement control logic. Listing 2.1 on Page 27 demonstrates such tangling for the print quality physical relationship: the relationship is implemented in the `RadiatorController` software module. When physical relationships are hard to locate, making the adaptations, such as mentioned in evolution scenarios 1.1 to 1.3, costs more development effort. Therefore, identifying and changing physical models implemented in a GPL is classified as having medium development impact.

- Affected physical relationships may be duplicated through the code, as they are needed at different locations in the software and the engineers implementing the physical relationship were unaware of the other implementations. When physical models are duplicated, the adaptations, such as mentioned the evolution scenarios, also need to be duplicated. This costs more development effort, and can be error-prone, as it is easy to miss duplicates or make updates to the duplicates that are inconsistent.

- Engineers might not be aware that a certain physical model has been implemented in software, because the model is hidden in a software module implementing control logic. This unawareness might lead to errors if the corresponding physical characteristics changes, and these changes are not (properly) propagated to the implemented physical model.

Proposed Techniques The application of the physical model composition approach presented in Chapter 2 provides the following advantages:

- The physical models affected by a change are easier to locate and change, as they have been implemented in a DSML and are separate from other control logic.

- Separation using a DSML lowers the risk of duplication of the physical relationship through the code base.

- The application of the Composition Filters model makes it easy to modify the composition of the physical models with software modules. For example, in scenario 1.2, a new composition filter can be added that dispatches $CheckUpdate$ events for the newly introduced variable m_{paper} to `PhysicalSystemIO`. An equivalent addition of a composition filter can be done for scenario 1.3, in which a new variable $T_{environment}$ is introduced in the `PrintQuality` and `BeltTemperature` physical models. In this case, the aspect-oriented superimposition mechanism

of the Composition Filters model allows the definition of a single composition filter that is applied on both the `PrintQuality` and `BeltTemperature` physical model.

These advantages reduce development effort and cost and decrease the risk for making errors.

5.5.2 Evolution Scenario Type 2: Adding or Changing Runtime Verification

The physical models implemented in embedded control software may not correspond to physical reality. Chapter 3 proposes a technique to verify the consistency of implemented physical models at runtime. The proposed technique can also be used to identify and diagnose problems in the physical system, such as a broken sensor or wear and tear of a component.

Evolution Scenarios

We will now give several evolution scenarios. These evolution scenarios make use of examples in this thesis.

Evolution Scenario 2.1
Adding Consistency Checking

Description
Example 3.1 on Page 92 in combination with Listing 3.4 on Page 94 gives an example in which the consistency of the `BeltTemperature` physical model is checked.

Consequences for the software implementation
To perform such consistency checking, the consistency between multiple values for the same physical variable is checked. The software needs to implement such a check and additional code to handle the inconsistency.
Example 3.1, for example, implements this by using the approach proposed in Chapter 3. Consistency checking is performed by adding an additional physical relationship to the physical model (Equation 3.1) and adding a composition filter specification that monitors for inconsistencies in the `BeltTemperature` physical model (Listing 3.4).

Summary of Development Impact

State-of-the-art techniques:

- **M:** Implement the additional physical relationships in the GPL.
- **M:** Implement checking code in the GPL to detect and handle inconsistencies.

Proposed techniques:

- **L:** Implement redundancy in the physical model.
- **L:** Implement composition filters to monitor for inconsistencies.

Evolution Scenario 2.2
Adding Physical Model Calibration

Description
Example 3.5 on Page 107 gives an example for the Drum Shuttling case study in which the `Shuttling2` physical model instance is calibrated for missed steps by the stepper motor, using a *home sensor*.

Consequences for the software implementation
In this example, the physical model already exists. To add calibration, two composition filters are defined: one that updates the value of *zPos* in the physical model instance when the home sensor is active and one filter that handles resulting inconsistencies in *zPos*. Note that the example shows a third filter definition, to log large deviations. For calibration, this filter definition is not necessary.

Summary of Development Impact

State-of-the-art techniques:

- **M:** Identify in the GPL the implementation of the physical model and add GPL code to retrieve the sensor value.
- **M:** Add GPL code to update the state in the GPL with the sensor value.

Proposed techniques:

- **L:** Implement a separate composition filter to retrieve the sensor value.
- **L:** Implement a separate composition filter to handle inconsistency between state and sensor (calibration).

Evolution Scenario 2.3
Adding Broken Sensor Detection

Description
It is a likely that sensors in the system break down over time. Therefore, detection of such broken sensor needs to be implemented in the software.
Examples 3.6 and 3.7 showed how detection of a broken sensor can be implemented for the Drum Shuttling case study.

Consequences for the software implementation
To detect a broken sensor, certain code that monitors for this situation needs to be implemented in the software. This code has to monitor the sensor readings and compare this to an internal model of what the expected sensor readings would be.
In the given example, detection is implemented on top of the already existing `Shuttling2` physical model, by adding a number of composition filters, in combination with a *helper* software module to maintain state between different executions of the composition filters. The specified filter monitors whether calibration using the sensor is actually performed, and how often calibration happens. From this information, the composition filters can deduce whether the home sensor is broken.

Summary of Development Impact

State-of-the-art techniques:

- **M:** Identify in the GPL the implementation of the physical model and add GPL code to check for situations that indicate a broken sensor.

Proposed techniques:

- **L:** Implement a separate composition filter to monitor for situations that indicate a broken sensor.

Evolution Scenario 2.4
Adding Generic Inconsistency Monitoring

Description
As part of a general logging and diagnosis mechanism, logging of inconsistencies in physical models throughout the software can be added to the software.
Listing 3.5 on Page 98 gives an example of a generic inconsistency monitor that provides logging for significant inconsistencies in all physical model instances.

Consequences for the software implementation
To implement such a logging mechanism, all physical models implemented in software should be monitored. Using a state-of-the-art GPL, such as C++, this means that monitoring code is added at all places in the code at which a physical model is implemented. The example in Listing 3.5 makes use of the aspect-oriented mechanisms in the Composition Filters model to specify generic monitoring in a concise way.

Summary of Development Impact

State-of-the-art techniques:

- Identify in the GPL all code locations that implement a physical model with redundancy.
- **H:** At all code locations, implement inconsistency checking and logging code.

Proposed techniques:

- **L:** Implement a separate composition filter that performs logging.

Evolution Scenario 2.5
Changing Runtime Verification because of Changes in Physical Characteristics

---

Description
Section 5.5.1 explains a number of evolution scenarios that have an effect on the physical characteristics of the system. These effects are propagated to the physical models implemented in software. Runtime verification can be affected if these changes affect variables on which monitoring and inconsistency handling has been defined.

Consequences for the software implementation
When runtime verification is affected by changes in physical models, the implemented monitors need to be updated. When a state-of-the-art GPL is used, these monitors need to be located in the code. When our approach is used, the specified composition filters to monitor for and handle inconsistencies need to be updated.

Summary of Development Impact

State-of-the-art techniques:

- **M:** Identify in the GPL the implementation of the physical model and modify the implemented runtime verification code.

Proposed techniques:

- **L:** Update affected composition filters.

Evaluation

State-of-the-art Due to the *cross-cutting* nature of runtime verification with the physical models and other software modules, implementation using a GPL leads to the following issues:

- When physical models are implemented using a GPL, the logic to monitor and verify the physical model needs to be implemented at the same location as the physical relationship. This leads to *tangling* of application logic having different purposes: the primary purpose of the implemented physical model, and on top of that the purpose of runtime verification. For example, if the case referenced in evolution scenario 2.2 is implemented using a GPL, then there is a software module that maintains the current value of *zPos* and implements the functionality that updates this value based on information from the `StepperController`. This software module also has to implement the additional functionality of monitoring whether the home sensor is active and updating the value of *zPos* according to this information.

  Tangling of functionality makes the software modules hard to comprehend and maintain. Therefore, the impact of development actions to implement runtime verification of physical models in the GPL is classified as having medium development impact.

- Generic runtime monitoring and verification leads to many different interaction points in the software. If a GPL is used to implement this type of functionality, all interaction points need to be manually implemented. This would be necessary for evolution scenario 2.4. Because multiple development actions are necessary, scattered through the software, the implementation of such functionality in the GPL is classified as having high development impact.

Proposed Techniques The different evolution scenarios all refer to examples that apply the composition and verification approach proposed in Chapters 2 and 3. This approach has the following advantages, as demonstrated by the examples:

- The monitoring functionality needed to perform runtime verification is specified separately from the physical models that are monitored, using the Composition Filters model. This makes it easy to add, change or remove runtime verification, without affecting the physical model. For the examples referred to in evolution scenarios 2.1 to 2.4, the Composition Filters model is used to implement the

monitoring functionality. This improves comprehensibility and maintainability both of the physical model, as it does not contain verification code, and of the specified monitoring code.

- The aspect-oriented quantification mechanism in the Composition Filters model supports the implementation of more generic runtime monitoring and verification, such as proposed by evolution scenario 2.4, without having to explicitly specify all interaction points with the physical models in the software. This reduces the development effort to add such runtime monitoring and verification functionality.

These advantages significantly reduce development effort, compared to when runtime verification has to be implemented using only the GPL.

Note that this evaluation assumes that the used GPL does not contain general aspect-oriented mechanisms, such as a language like AspectJ [7] would have. An aspect-oriented language may be able to separately specify the runtime monitoring and verification logic. However, this does leave the question how a general aspect-oriented language would be able to quantify specifically over the physical models and events in their execution, while these physical models are implicitly implemented in a GPL.

5.5.3 Evolution Scenario Type 3: Adding or Changing Multi-Objective Optimization Solutions

Chapter 4 proposed the MO2 method to specify a multi-objective optimization solution in the architecture of embedded control software. The following evolution scenarios will be used to evaluate the benefits of the MO2 method.

Evolution Scenarios

Evolution Scenario 3.1
Adding Multi-Objective Optimization to Existing Embedded Control Software

---

Description
It is a common scenario in practice that a multi-objective optimization solution is added to already existing embedded control software. An example of this scenario is given in Examples 4.3, 4.4 and 4.5. In these examples the application of the MO2 method to add multi-objective optimization to the embedded control software of the Warm Process case study is demonstrated.

Consequences for the software implementation
To add multi-objective optimization to existing embedded control software, the different decision variables, constraints and objective functions need to be identified. Furthermore, information necessary from different modules in the

software needs to be identified. This information is, for example, the values of variables in the constraints that are independent from the decision variables. A software module needs to be developed that obtains the necessary information from the other software modules and implements an optimization algorithm, or calls an existing generic algorithm, providing the details of the specific multi-objective optimization solution.

Summary of Development Impact

State-of-the-art techniques:

- GPL implementation of an optimization module.
- **M:** Within this module, the constraints and objective functions need to be expressed in the decision variables.
- **M:** GPL implementation of interaction with other modules, to retrieve values or update the value of the decision variables.

Proposed techniques:

- **L:** Specification of the multi-objective optimization solution in an MO2 architectural model.

Evolution Scenario 3.2
Adding or Changing a Constraint

Description
Many constraints of a multi-objective optimization solution in embedded control software are based on characteristics of the physical system, for example how much power a certain heating component can handle. Changes in these components affect the constraints in the multi-objective optimization solution. An example of such a change is when the existing type of radiator in the system is replaced with a different type of radiator that can handle more power. Such a change affects the constraint on the port P_{rad} (labeled 2) of the `RadiatorController` component in the MO2 model in Example 4.5.

Consequences for the software implementation
The constraint has to be identified in the implemented optimization algorithm, and updated accordingly.
For example, if the MO2 method is used, the constraint on the power to the radiator can be located at the corresponding port P_{rad} and updated. Executing the tooling generates a new implementation of the optimizer.

Summary of Development Impact

State-of-the-art techniques:

- **M:** Add or update the implementation of the constraint in the GPL optimization module.

Proposed techniques:

- **L:** Add or update the constraint in the MO2 model.

Evolution Scenario 3.3
Adding an Objective Function

Description
A given multi-objective optimization solution might be extended with an additional objective function. For example, the multi-objective optimization solution for the Warm Process case study, presented in Example 4.5, might be extended with the objective *print quality*.

Consequences for the software implementation
To add an additional objective function in a multi-objective optimization solution, the function needs to be implemented. Furthermore, it is possible that interaction with other software modules needs to be implemented, to obtain necessary information to compute the objective.
When the MO2 method is used, an additional `Oblivious` component that implements the logic to calculate the objective (e.g., print quality) is added to the MO2 model. An out-port providing the value of this objective is added to this component. This out-port has the `isObjective` flag set. Furthermore, in-ports are added to obtain necessary information. These in-ports are connected to ports of other components in the MO2 model.

Summary of Development Impact

State-of-the-art techniques:

- **M:** Implement the objective function (expressed in the decision variables) in the GPL optimization module.
- **M:** Implement interaction with other modules to obtain necessary information for the objective function, for example values of independent variables.

Proposed techniques:

- **L:** Add an `Oblivious` component that calculates the objective value to the MO2 model, with a reference to the SIDOPS+ specification that contains the objective function.
- **L:** Add connection between ports of the added component and other ports in the MO2 model.

Evolution Scenario 3.4
Change in Control Logic or Implemented Physical Model

Description
Constraints and objective functions are mathematically related to the decision variables by the implemented control logic and physical models. The implemented control logic and physical models are subject to change.

Consequences for the software implementation
When such control logic or implemented physical model changes, the mathematical relationship between constraints/objective functions and the decision variables should be updated. This mathematical relationship needs to be identified in the implementation of the optimizer and updated accordingly.
The MO2 method uses SIDOPS+ specifications of the control logic or physical models implemented in the components, to determine how the decision variables mathematically relate to the specified constraints and objective functions. Examples of changes in physical models were given in the evolution scenarios in Section 5.5.1. Changes in control logic or physical models should be reflected in the referenced SIDOPS+ specifications.

Summary of Development Impact

State-of-the-art techniques:

- **H:** Update affected constraints and objective functions in the GPL, to reflect the new control logic. There can be multiple affected constraints and objective functions.

Proposed techniques:

- **L:** Update this SIDOPS+ specification that contains the changed control logic or physical model.

Evaluation

State-of-the-art Manual implementation of multi-objective optimization in embedded control software leads to the following problems:

- Manual implementation of multi-objective optimization solutions is often not considered, because the complex mathematical relationships between decision variables, constraints and objective functions are hard to comprehend and lead to complex interactions in the embedded control software. This would reduce the comprehensibility and maintainability of the software. For example, the constraint $P_{rad} \leq 1500$ attached to the P_{rad} port of the `RadiatorController` component in Figure 4.7 is translated to the following constraint, expressed in the decision variables and independent variables:

$$\begin{aligned} &10.0 * (v * c_1 - c_2 * T_{ph} + c_3 - T_{contact}) \\ &+ 0.1 * (PIState + v * c_1 - c_2 * T_{ph} + c_3 - T_{contact}) \\ &- 1500 < 0 \end{aligned}$$

  Because of these complex interactions, the development impact of implementing or changing such constraints and objective functions is classified as medium.

  If certain control logic in the embedded control software changes, such as suggested by evolution scenario 3.4, the complex mathematical relationships in the related constraints and objective functions have to be updated accordingly. Because there can be multiple affected constraints and objective functions, the development impact is classified as high.

- Typical engineering solutions, e.g. the Intelligent Speed algorithm shown in Section 5.2.2, to optimize certain objectives in the system lead to solutions that are:

  - Less effective in optimizing the objectives, as was demonstrated with the experiment in this chapter.
  - Less flexible to change trade-offs between different objectives, based on user needs: these solutions are typically designed to optimize one objective, as optimizing multiple objectives using a dynamically changing trade-off increases the complexity. An example of this is the Intelligent Speed algorithm, presented in Section 5.2.2.
  - Less capable to maintain certain margins in the system, such as a power margin, as was demonstrated in Section 5.4.3.

  As such, an engineering trade-off has to be made between software engineering quality (e.g., comprehensibility and maintainability) and software functionality.

Proposed Techniques As compared to state-of-the-art techniques, the MO2 method has the following benefits:

- The different elements of a multi-objective optimization solution (decision variables, constraints, objective functions) are decomposed and related to corresponding components and ports in the architecture of the embedded control software. This improves the comprehensibility of the multi-objective optimization solution, and makes it easier to locate and change elements. For example,

evolution scenario 3.2 gave an example of changing the constraint on the amount of power the radiator can handle. This constraint is attached to the P_{rad} outport of the `RadiatorController` component, so easy to locate.

- The techniques of the MO2 method, implemented in tooling, provide automated analysis of the often complex mathematical relationships between the decision variables, constraints and objective functions of the multi-objective optimization solution. For example, changes in control logic or implemented physical models (evolution scenario 3.4) or the addition of an objective function (evolution scenario 3.3) have an impact on these mathematical relationships. The engineer does not manually have to update these mathematical relationship in case of such changes; this is automatically performed by the techniques in the MO2 method.

- The MO2 method provides techniques and tooling to automatically generate an optimizer module, together with instrumentation to apply the optimization functionality to the implemented software modules. In this way, the multi-objective optimization solution specified in the architecture of the embedded control software does not have to be manually implemented.

This evaluation clearly shows that the proposed MO2 method leads to engineering advantages: the MO2 method manages the essential complexity of multi-objective optimization solutions, so that engineers are able to add multi-objective optimization to embedded control software and to properly handle changes in the multi-objective optimization solution during the life-time of the product family. These advantages of the MO2 method lead to reduced development effort and cost, and a better performing system.

5.5.4 Summary of Development Impact

Table 5.1 provides a summary of the development impact of the different evolution scenarios, by providing the total number of actions with low, medium and high impact in the evolution scenarios. This summary shows that with state-of-the-practice techniques, the impact on development effort for the evolution scenarios is generally medium with some occasions of high impact. With the proposed techniques, the development effort is generally low.

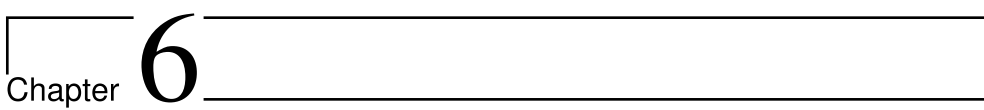
	Low	Medium	High
State-of-the-practice techniques	0	15	2
Proposed techniques	16	0	0

Table 5.1: Summary of development impact

5.6 Conclusion

In Chapter 4 we introduced the MO2 method, which enables the implementation of multi-objective optimization in embedded control software. This chapter discusses an experiment in which the performance concerning productivity of embedded control software with multi-objective optimization (designed using the MO2 method) is compared to three other embedded control software implementations. The main result of this experiment is that software that contains multi-objective optimization performs at least as good as, and typically better than state-of-the-practice solutions to optimize productivity. Furthermore, the added benefit of multi-objective optimization is that it enables dynamic trade-offs between multiple objectives, as opposed to state-of-the-practice techniques, which optimize for one objective. Besides that, multi-objective optimization provides more stable and smooth transitions between states in the system. Furthermore, we have shown that multi-objective optimization enables precise control of margins in the system, for example a power margin, as opposed to state-of-the-practice techniques, in which these margins are unpredictable. As such, we can conclude that the MO2 method enables a better performing system, as the MO2 method enables the implementation of multi-objective optimization algorithms.

To provide a qualitative analysis on how well the proposed techniques in this thesis manage complexity, a number of evolution scenarios have been presented. The development effort for these evolution scenarios is discussed. From this analysis it can be concluded that the physical model composition technique presented in Chapter 2 reduces the effort to cope with changes in the physical characteristics in the system, as the implemented physical models are separate from other application logic and have been implemented using a DSML. Furthermore, it can be concluded that the application of the Composition Filters model makes it easier to add and modify compositions between physical models and software modules. The MO2 method manages the complexity of multi-objective optimization solutions by decomposing the different elements (decision variables, constraints and objective functions), and the mathematical relations between them, in the architecture of the embedded control software. This simplifies the constraints and objective functions, making it easier to modify, add or remove them. The often complex mathematical relationships between the decision variables and the constraints and objective functions are generated by the MO2 tooling, and as such do not have to be manually maintained. In general, with state-of-the-practice techniques the evolution scenarios have a medium to high impact on development effort. With the proposed techniques, the evolution scenarios have generally a low impact on development effort.



Conclusion & Future Directions

There is a constant pressure for embedded system manufacturers to build better embedded systems for lower costs. These systems need to cope with an increased variation in customer needs, operating conditions, etc. To enable this, more advanced algorithms to control the system are being applied. However, the implementation of these advanced algorithms increases the complexity of embedded control software. Higher complexity has a negative impact on software quality, such as comprehensibility, maintainability and evolvability. This either leads to higher development costs, or to the decision, made by engineers, not to implement more advanced algorithms to control the system. As such, opportunities to gain a competitive advantage can be missed.

This thesis presents structured methods and techniques to better manage the complexity caused by more advanced algorithms for adaptive control behavior. The application of these structured methods and techniques decreases the impact of this complexity on software quality. This reduces the development costs, and as such enables engineers to develop more adaptive systems.

This chapter presents the conclusions of the thesis. First, we summarize the problems addressed in this thesis. Then, an integrated overview of the approaches in this thesis is presented. Next, we discuss the contributions provided by these approaches. Finally, we propose directions of future work that extend or complement the work presented in this thesis.

6.1 Problems Addressed

In this thesis, we focused on the following two approaches for adaptive behavior:

- The implementation of physical models in embedded control software, to provide adaptive behavior.
- Multi-objective optimization of system qualities under varying conditions, while making dynamic trade-offs between these qualities based on user needs.

We will now summarize the problems addressed in these thesis concerning these two approaches for adaptive behavior.

6.1.1 Physical Models in Embedded Control Software

Implementation & Composition

Making embedded systems more adaptive is desirable, as it improves the ability of the embedded system to function in a wider range of circumstances. This can be translated into additional features or quality improvement of the embedded system, and as such adaptivity provides a competitive advantage. One strategy to increase the adaptivity of the system is to apply more knowledge about the physical characteristics of the system in the control logic. For example, in earlier designs of control logic, the variables in the system are controlled to fixed setpoints. These setpoints are selected by engineers based on the physical characteristics of the system. However, flexible control of variables, allowing variations in their values, increases the working area of the embedded system and enables the system to react more effectively to changing circumstances. But for the system to behave correctly, certain constraints between different variables should be preserved. Before, these constraints were implicitly preserved by the determined setpoints for the variables. But when variables are controlled flexibly, the embedded control software also actively has to monitor and preserve the constraints. It is common that such constraints are based on the physical characteristics of the embedded system. Therefore, to actively preserve these constraints, model of physical characteristics are implemented in embedded control software.

As was illustrated in Chapter 2, there are two commonly used approaches to implement physical models in embedded control software. The first approach is to implement physical models using a general-purpose programming language (GPL). The second approach is to use a domain-specific modeling language (DSML) in combination with code generation techniques to specify and compile physical models.

However, both approaches have a number of drawbacks. Implementing physical models using a GPL leads to the following problems:

- Introduction of accidental complexity: The GPL does not support the domain-specific abstractions of physical models. Therefore, these domain-specific abstractions need to be implemented by a number of program elements in the GPL, introducing unnecessary complexity.

- Physical models are harder to recognize: Domain-specific abstractions are lost when the physical models are implemented in a GPL, which makes the physical models harder to recognize as such in embedded control software. We call this implicit implementation of physical models. Such implicit implementation makes these physical models harder to locate when changes are necessary and reduces the comprehensibility of the implemented physical models.

- Increased possibility for tangling of physical models with other control logic, and for scattering and duplication of physical models through the software. Because physical models are implicitly implemented in embedded control software, their are no clear boundaries between them and other implemented application logic. This leads to decomposition of physical models and scattering of the physical

relationships through the software, possibly even duplication of physical relationships in several software modules.

These issues reduce certain software quality characteristics such as comprehensibility, maintainability and reusability, resulting in higher development costs.

Using a DSML in combination with code generation techniques reduces or eliminates the problems related to the use of a GPL, because domain-specific abstractions do not have to be expressed using implementation abstractions. However, embedded control software usually also contains other, application-specific functionality, which cannot easily be implemented with a DSML for physical models. Examples of other functionality in embedded control software are scheduling of (discrete) tasks, recovering from errors, processing user input, security and communication. Usually, a GPL is used to implement such application-specific functionality. Code-generation techniques are applied to compose physical models specified in a DSML with GPL modules. However, code generation usually leads to black boxes with inflexible interfaces resulting in a tight integration of these modules with other software modules. This reduces the comprehensibility and maintainability of the embedded control software.

What is lacking is a technique that offers flexible composition of physical models specified in a DSML with other software modules specified in a GPL, on the abstraction level of both the DSML and the GPL.

Verification

Implemented models of physical characteristics may not always accurately reflect physical reality, e.g., because the physical system has evolved, the system is used in different circumstances than it was tested for, the characteristics of the physical system have changed because of wear and tear, etc. As inaccuracies in physical models may lead to incorrect behavior of the system, the accuracy of these models needs to be verified. One cannot test or statically verify the system for all possible conditions, thus runtime verification is necessary.

In Chapter 3 we explain that traditional runtime verification techniques cannot be applied to verify whether physical models conform to physical reality. The reason for this is that faults in implemented physical models lead to observable failures in the physical behavior of the system, but not necessarily to observable failures in software behavior. This hinders the application of common runtime verification techniques that focus on monitoring software behavior only.

6.1.2 Designing Multi-Objective Optimization Functionality in Embedded Control Software

In traditional embedded systems development, trade-offs between (conflicting) system qualities (e.g., productivity, energy consumption) are made at design time. This results in embedded systems in which these qualities are fixed. For example, the result is either a highly productive system with high energy consumption, or an energy-saving system with low productivity. Nowadays, market forces demand more flexible and adaptable machines, in which the trade-offs between system qualities can be made at runtime. For example, a customer of a printing system might sometimes need a

high-productive machine (e.g., for time-critical print jobs), while in general he prefers an energy-saving system. The system needs to take many variables and constraints into account, and needs to solve an optimization problem involving multiple objectives (reflecting the system qualities), to adapt itself for optimal operation. Furthermore, the system has to make dynamic trade-offs between the objectives, to accommodate changing user preferences.

As system qualities are not localized to a specific part of the system, but arise from the system as a whole, the optimization algorithm has an impact on the entire system. Algorithms to perform multi-objective optimization already exists. What is lacking is a structured method to design a control system that applies such algorithms.

Chapter 4 concludes that the lack of systematic methods to design multi-objective optimization functionality in embedded control software results in the following issues:

- **Ad-hoc solutions:** Because there is no systematic method to design multi-objective optimization, a software engineer may fail to make the proper abstractions and corresponding mapping to the domain of multi-objective optimization. This leads to solutions that are tailored to the specific system and therefore inflexible when the system changes or evolves, to solutions that are difficult to understand, because they do not apply standard multi-objective optimization abstractions and to solutions that may not optimally control the system, as an ineffective optimization algorithm has been chosen.

- **Tight coupling:** The different elements of an multi-objective optimization solution (decision variables, constraints and objective functions) are related to different architectural elements (ports, components) in the control software architecture. This creates a scattered relationship between the multi-objective optimization solution and the control software architecture. As such, it may be difficult to implement a proper decomposition of the chosen multi-objective optimization solution in embedded control software. This results in a multi-objective optimization solution that is tightly coupled and integrated with, and spread out over multiple control software components, making the control software less comprehensible and more difficult to maintain and reuse.

The above described problems lead to reduced software quality: inflexible and tightly coupled solutions are more difficult to comprehend, their inflexible nature hinders their evolvability and reusability. The lack of a common methodology and corresponding terminology makes it harder to document and communicate the design decisions. The result is higher development and maintenance costs [14]. Alternatively, it can lead to the decision to remove the runtime optimization requirement, as the benefits of a more optimal system do not outweigh the reduced software quality and increased development and maintenance costs.

6.2 Integrated Overview of the Approaches

Figure 6.1 shows an integrated overview of the approaches presented in this thesis.

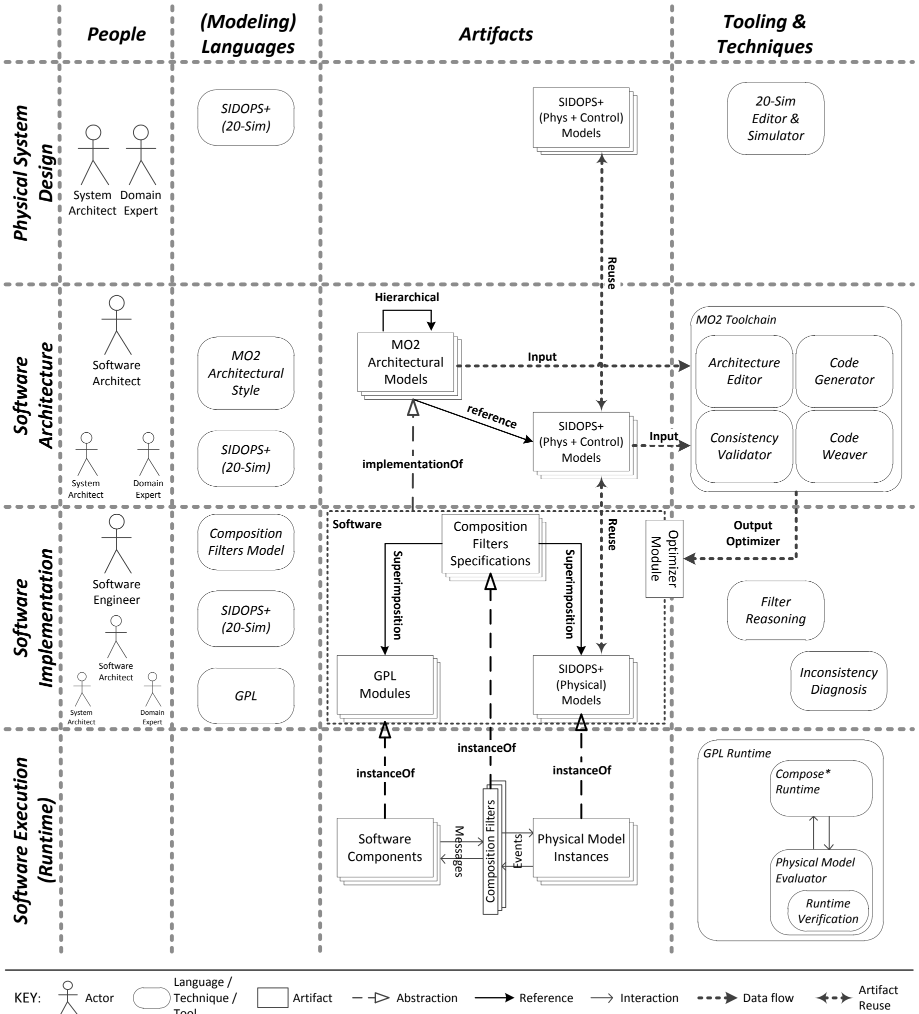


Figure 6.1: Integration of our approach

Vertically, this figure is divided into four different design levels:

- *Physical System Design.* This level refers to the all design stages of the physical system. Mainly *system architects* and *domain experts*, for example control engineers, are involved at this level.
- *Software Architecture.* This level refers to the design and documentation of the architecture of the embedded control software. At this level, mainly *software*

architects are involved, with some involvement of system architects and domain experts.

- *Software Implementation.* This level refers to the implementation of the embedded control software. *Software engineers* provide the implementation of the software, in cooperation with software architects and to a lesser extent with system architects and domain experts.

- *Software Execution.* This level refers to the executing software[1].

Horizontally, the figure is divided in four parts, showing for each design level the *people*, *(modeling) languages*, *artifacts* and *tooling & techniques* involved in the design level.

6.2.1 Physical System Design

System architects and several *domain experts* design the physical system. Models of physical characteristics and of control logic are specified using tools as 20-Sim/SIDOPS+ [1] and Matlab Simulink [6], to simulate and validate the behavior of the physical system. The specified physical models can be reused in the other design levels.

6.2.2 The MO2 Method

The MO2 method, presented in Chapter 4 of this thesis, is mainly targeted at the *software architecture* level. The MO2 method includes the *MO2 architectural style.* The MO2 architectural style is a modeling language to specify *MO2 architectural models* (or *MO2 models*).

A MO2 model is a model of a control software architecture that includes multi-objective optimization. MO2 models are created by software architects and they are part of the *artifacts* at the *software architecture level.* MO2 models can reference other MO2 models, to provide *hierarchical composition.* MO2 models can also reference SIDOPS+ models. These SIDOPS+ models specify part of the computational logic of the components in the MO2 model. They mainly specify the part of the computational logic necessary to analyze the mathematical relationships between the constraints and objective functions, and the decision variables.

The *MO2 toolchain* implements techniques to analyze a MO2 model and referenced SIDOPS+ models, and to generate a software module that implements an optimization algorithm specific for the specified multi-objective optimization solution in the MO2 model. The generated software module becomes part of the artifacts at the *software implementation* level, as shown in Figure 6.1.

[1] Technically, this is not a design level, but a result of the design. This level is added to get a complete picture of the different artifacts and techniques introduced in this thesis, and how they are related.

6.2.3 Composition of Physical Models with Software Modules

Chapter 2 proposes a novel approach to compose physical models, specified in the domain-specific modeling language (DSML) SIDOPS+, with software modules specified in a general-purpose programming language (GPL). This approach is mainly targeted at the *software implementation* and *software execution* levels.

The approach to compose physical models with GPL modules consists of two main parts:

- The techniques to instantiate physical models at runtime and to execute them. These techniques have been implemented in a runtime interpreter called the *physical model evaluator*.

- An extension of the Composition Filters model, so that composition filters can filter and match *execution events* that conform to a given *event model*. Such an event model has been specified for the execution of physical model instances. A composition between physical models and software modules is specified in a separate *composition filters specification*, which is *superimposed* on software modules and/or physical models.

  In this way, the Composition Filters model can be applied to compose software modules implemented in a GPL with physical models specified in SIDOPS+ at the abstraction level of both languages. Interaction takes places in terms of messages (GPL) and events (physical model instance). This interaction is shown in Figure 6.1 in the *artifacts* part of the *software execution* level. The Compose* toolset and interpreter have been adapted to support the extended Composition Filters model.

Filter reasoning techniques are explained in Chapter 3 to analyze the composition filter specifications that compose physical model instances with GPL modules. Filter reasoning takes place at the *software implementation* level.

6.2.4 Verification of Physical Models

Chapter 3 introduces a technique to verify implemented physical models for their correspondence with the physical system. This technique makes use of redundancy in the physical model to detect inconsistencies. The technique has been implemented as part of the *physical model evaluator*, as shown in Figure 6.1 (*runtime verification* part of the *physical model evaluator*). Monitors to observe and handle inconsistencies can be specified using the extended Composition Filters model.

When inconsistencies are detected, diagnosis of the causes can be performed. This diagnosis takes place at the *implementation level*. The diagnosis algorithm analyses the physical model and composition filters specification to determine the possible causes of a detected inconsistency.

6.3 Contributions

The techniques proposed in this thesis serve to manage the complexity of control software for adaptive embedded systems. Additionally, supporting techniques for analysis and verification have been introduced in this thesis. This section summarizes the contributions of the proposed techniques. First the contributions for managing the complexity are given, followed by the contributions concerning analysis and verification.

6.3.1 Managing Complexity

Adaptive Embedded System

Contribution 1 Chapter 1 provides a novel definition of adaptive embedded systems. This definition unifies two existing definitions of an adaptive system: one definition that defines an adaptive system from a structural perspective and one definition that gives a formal mathematical definition of adaptive behavior. The two definitions are combined into a definition of an adaptive system from both a structural as a formal mathematical perspective and how these two perspectives relate.

Physical models implemented in Embedded Control Software

Contribution 2 In Chapter 2 we observe that control software for adaptive embedded systems necessarily integrates implementations of (pieces of) physical models. This observation is supported by observations at our industrial partner. In Chapter 2 we also observe that the implementation of these physical models in a general-purpose programming language increases the complexity of embedded control software.

Implementing and Executing Physical Models using a DSML

As a solution to better manage the complexity of physical models in embedded control software, we propose in Chapter 2:

Contribution 3 A novel approach to implement physical models separately from other application logic in embedded control software using the domain-specific modeling language SIDOPS+, and to instantiate and execute physical models implemented using SIDOPS+.

Benefits Applying a DSML to specify and execute physical models provides the following benefits:

- Reduction of accidental complexity that is introduced when domain-specific abstractions have to be expressed as implementation abstractions in a GPL. For example, the SIDOPS+ language naturally supports the definition of equations, a common concept in physical models. A GPL does not directly support equations and equation solving, and as such other language elements need to be used

to implement the same semantics. The reduction of accidental complexity improves comprehensibility, maintainability and reusability of embedded control software.

- Separation of physical models from other application logic, reducing tangling and scattering of physical models in embedded control software. This makes the physical models easier to locate and modify, as was demonstrated using evolution scenarios (1.1 to 1.3) in Chapter 5. These evolution scenarios demonstrate that the impact on development effort is generally reduced from medium impact to low impact, by applying the proposed techniques and according to the classification introduced in Section 5.5.

- Reuse of physical models from system development stages. Such reuse between development stages reduces development costs and improves software maintainability, as updates in physical models from the system development stage are directly part of the software.

The techniques to instantiate and execute physical models implemented using SIDOPS+ have been implemented in a toolset. This toolset has been applied to the examples in this thesis.

Composition of Physical Models with Software Modules

Physical models implemented in SIDOPS+ need to interact with software modules implemented in a GPL. As explained in Chapter 2, code generation approaches utilized by tools as 20-Sim lead to black boxes in software that have inflexible interfaces and are tightly integrated with other software modules. To prevent the problems of code generation approaches, Chapter 2 proposes:

Contribution 4 A novel extension of the Composition Filters model that enables filtering and matching of events in the execution of a physical model. This extension of the Composition Filters model enables the specification of interaction between physical models and software modules in terms of messages (software modules) and execution events (physical models). In this way, the Composition Filters model can be applied to compose physical models specified in SIDOPS+ with software modules specified in a GPL at the abstraction level of both languages.

Benefits The application of the extended Composition Filters model to compose physical models with software modules provides the following benefits:

- The unified message and event mechanism enables interaction between physical models and software modules at the abstraction level of both the DSML and the GPL. In this way, software modules do not have to be tailored to code generated from DSML models. As such, the comprehensibility and maintainability of embedded control software is improved.

- The application of the Composition Filters model provides loose coupling between physical models and software modules. The Composition Filters model

separates the interaction between physical models and software modules from their implementation. Evolution scenarios 1.2, 1.3 and 2.1 to 2.4 in Chapter 5 demonstrate the maintainability and evolvability advantages of this characteristic: interactions can be modified, added and removed without having to modify the software modules or physical models. The impact on development effort is in general reduced from medium impact to low impact.

Also, comprehensibility is improved, as software modules do not have to be tailored to specific interfaces of physical models (generated code). Finally, reusability is improved as separated interaction makes it easier to reuse physical models and software modules in a different context.

- Aspect-oriented quantification enables the specification of cross-cutting interaction between physical models and software modules, for example logging of all inconsistencies in all physical models. Instead of having to implement cross-cutting interaction in the affected physical models and software modules, it can be specified separately using the aspect-oriented quantification mechanism of the Composition Filters model. This enhanced the comprehensibility and maintainability of the cross-cutting interaction, as is demonstrated using evolution scenario 2.5 in Chapter 5. In this case, the impact on development effort is reduced from a high impact to a low impact.

- The declarative composition filters language enhances the analyzability of the composition, so better verification techniques can be applied to reduce the possibility of errors remaining undetected. The verification techniques are presented in Chapter 3 and their contributions will be given in Section 6.3.2.

The technical feasibility of the physical model composition approach has been demonstrated by implementing a proof-of-concept toolset and applying this toolset to the experiment in Chapter 5.

Multi-Objective Optimization of System Qualities

Contribution 5 Chapter 4 proposes a systematic method, called the MO2 method, to design and document multi-objective optimization within the architecture of embedded control software. The MO2 method includes:

- The MO2 architectural style that supports the architectural design and documentation of control software containing multi-objective optimization functionality.

- A notation that supports the specification of architectural models, called MO2 models, according to the MO2 architectural style.

- Support for hierarchical composition of multi-objective optimization solutions.

- Techniques to support the code generation of an optimization component that is tailored to the given software architecture, based on the specification in a MO2 model.

Benefits The MO2 method provides the following benefits:

- The ability to solve the design of multi-objective optimization at the architectural level, instead of only at the implementation level. This enables the analysis of a multi-objective optimization solution and code generation of a software module that contains the optimization logic. Chapter 4 presented the analysis and code generation techniques. Furthermore, the architectural focus provides documentation of the multi-objective optimization solution.

- Decomposition of the different elements of the multi-objective optimization solution (i.e., decision variables, constraints and objective functions) within the control software architecture. This decomposition improves the maintainability and adaptability of the multi-objective optimization solutions, because relating the elements to corresponding components in the architecture makes them easier to locate. This benefit has been demonstrated with evolution scenarios 3.2 and 3.3 in Chapter 5.

- Possibility to introduce multi-objective optimization to existing control software. This is possible because the MO2 style enables adding the different elements of a multi-objective optimization solution to an existing control software architecture. Furthermore, the MO2 toolset is able to generate an optimization software module and the instrumentation needed to compose this module with the existing control software modules. These characteristics of the MO2 method enable software engineers to include multi-objective optimization into their embedded software systems without having to extensively redesign the software. This benefit is demonstrated using evolution scenario 3.1 in Chapter 5.

The evolution scenarios 3.1 to 3.4 demonstrate that the application of the MO2 method reduces the impact of these evolution scenarios from a medium or high (scenario 3.4) impact to a low impact. This impact is determined according to the classification introduced in Section 5.5.

A proof-of-concept of the MO2 toolset has been implemented. In Chapter 5, the toolset is applied to an industrial example in which the productivity of a printer system has to be optimized. The generated optimizer is compared to three state-of-the-practice engineering solutions for optimization. The four solutions are compared using 140 different scenarios in which the amount of power available is limited and fluctuating with several different intervals, making speed adjustments necessary (i.e., the system cannot print at highest speed). Furthermore, 11 different scenarios with a limited but fixed amount of power available are tested.

The main result of this experiment is that software that contains multi-objective optimization performs equally well or even better than state-of-the-practice solutions to optimize productivity[2]. The added benefit of multi-objective optimization is that

[2]Note, however, that we used the Matlab function `fmincon` to perform the optimization. `fmincon` is a very powerful optimization algorithm, but also computationally inefficient. It is therefore unlikely that this algorithm in particular will be applied within an embedded system. However, as explained in Section 4.8.3, there exist other, more efficient (approximation) algorithms, which are suitable for application in an embedded system.

it enables dynamic trade-offs between multiple objectives, as opposed to state-of-the-practice techniques, which optimize for one objective. Furthermore, we have shown that multi-objective optimization provides precise control of margins in the system, for example a power margin, as opposed to state-of-the-practice techniques, in which these margins are unpredictable. As such, we have shown that the MO2 method enables a better performing system, as the MO2 method enables the implementation of multi-objective optimization by managing the complexity of MOO solutions and as such improving the maintainability and evolvability of software containing these solutions.

6.3.2 Analysis & Verification

Runtime Verification of Physical Models

Contribution 6 Chapter 3 proposes a novel technique to verify at runtime the consistency of the models of physical characteristics used in control software with the physical reality.

- This technique utilizes redundancy in the physical model[3] to detect inconsistencies.
- The Composition Filters model is applied to monitor and handle detected inconsistencies.
- Detected inconsistencies can be diagnosed using the introduced concept of derivation graph of the physical model.

Benefits The approach provides the following benefits:

- Runtime verification of the physical models implemented in software enables the detection of faults in these models and enables monitoring for wear-and-tear or broken components in the embedded system. Several case studies in Chapter 3 demonstrate these possibilities.
- By applying the Composition Filters model, monitoring code can be added, modified and removed without having to modify the monitored physical model. This improves the comprehensibility of the physical model and the maintainability of the software, as has been demonstrated with evolution scenarios 2.1 to 2.3 in Chapter 5. These evolution scenarios show that by applying the proposed techniques, the impact on development effort of adding or changing monitoring code is reduced from medium impact to low impact.
- Furthermore, the Composition Filters model provides aspect-oriented mechanisms to apply more generic monitors to multiple physical models. This enables a concise and separated specification of such interaction, instead of having to implement and maintain monitoring code at the different places in software at

[3]Redundancy in the physical model means that there are multiple ways to determine the value for certain physical variables in the model.

which physical models are used. This improves the comprehensibility and maintainability of the software, as has been demonstrated with evolution scenario 2.4 in Chapter 5. The impact on development effort is reduced from a high impact to a low impact.

- Diagnosis of detected inconsistencies enables decision making about what kind of action to take to resolve the inconsistency, as is explained in Section 3.3.2.

Analysis of Composition Filters

The analysis of the behavior of a composition filters specification is beneficial for various purposes. Chapter 3 proposes a technique to analyze the behavior of a composition filters specification that uses the extended Composition Filters model.

Contribution 7 A novel technique, called filter reasoning, to statically analyze the behavior of composition filters that compose physical models with GPL modules. Filter reasoning is able to analyze:

- The behavior of a specific message or event in a set of composition filters.
- All possible behaviors of a set of composition filters. This is determined by creating equivalence classes of messages and events with the same behavior in the filter set, and simulating the filter set with a message or event from each equivalence class.

Benefits Filter reasoning results in a state space of the possible executions of a set of composition filters. This resulting state space has several applications, which include:

- Detecting inconsistencies in the filter set.
- Detecting redundancy in the physical model. This is useful to detect and handle inconsistencies in the physical model.
- Further diagnosing the cause of detected failures/inconsistencies in the physical model.
- Compiling the filter set to efficient GPL code.

6.4 Future Directions

The aim of this thesis was to introduce techniques to manage the complexity of software in adaptive embedded systems. This thesis mainly focused on complexity introduced by physical models and on complexity introduced by multi-objective optimization functionality. We will now discuss directions in which this work can be extended.

6.4.1 Managing Complexity of other Domain-Specific Functionality

As was already explained in Chapter 2, embedded control software usually also contains other application-specific functionality that does not fall in the categories of physical models and continuous control logic. Examples of such functionality are managing system states (e.g., idle, start-up, available), scheduling of (discrete) tasks, recovering from errors, monitoring the available resources (e.g., the amount of toner, number of sheets of paper in the paper tray), processing of user input, security, communication, maintenance tasks and interaction with third-party libraries.

Some of these types of application-specific functionality may also apply more advanced algorithms to make the embedded system more adaptive. For example, flexible scheduling algorithms enable the system to react quicker to changes in tasks. Applying more advanced algorithms in other domains of application logic may also introduce complexity, possibly in a way that is equivalent to the complexity introduced by physical models. Further research needs to be performed to analyze these other domains of functionality in embedded control software and to find solutions to manage introduced complexity. Ideally, a generalization of the techniques presented in this thesis is derived, so that similar complexity problems in other domains of functionality in embedded control software can be addressed. Such a generalization might for example be possible for the event model with which the Composition Filters model has been extended; this event model is now specific for the execution of physical models, but might be generalized to execution events in any DSML.

6.4.2 Composition of Domain-Specific Concerns

To design embedded systems, such as digital document printing systems, multiple engineering disciplines cooperate. As a result, the design of embedded control software contains many concerns resulting from different domains. Examples of relevant concerns in embedded control software are continuous control logic, scheduling of tasks, timing of control tasks and multi-objective optimization of system qualities at runtime.

Often, these concerns have different views on the system. For example, continuous control logic and multi-objective optimization of system qualities have a continuous view on the system: the system consists of physical variables and continuous relationships between them. On the other hand, scheduling of tasks has a discrete view on the system: the system consists of different components that can execute certain actions. Tasks can be divided into actions that need to be executed and these actions can be allocated and scheduled to components.

Having different views is not necessarily a problem for the design and implementation of the concerns themselves, as long as a concern consistently uses one view. However, correct functioning of an embedded system relies on the integration and interaction of the different concerns in embedded control software. When these concerns have different views on the system, the composition of these concerns becomes tedious: the same concept can be represented differently between the concerns. This leads to complex interactions between the different concerns. This problem of integrating dif-

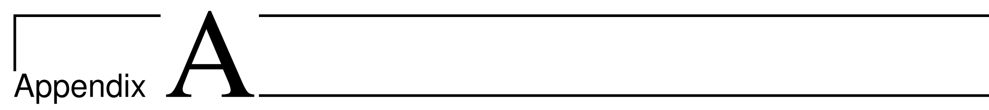
ferent domain-specific concerns is not effectively addressed by current composition techniques. Therefore, in current practice correct composition and integration relies on the skills and craftsmanship of system architects and software architects. Future research needs to be performed on how different domain-specific concerns, with different views on the system, can be composed in a systematic and conceptually sound way. Possibly, the Speeds approach [92] can be adopted to enable the composition of domain-specific concerns in embedded control software.

6.4.3 Vertical Integration of Engineering Disciplines

In this thesis we saw the utilization of domain-specific models (mainly physical models), derived from earlier engineering stages, in embedded control software. However, in current practice there is still a considerable separation between software engineering and other engineering disciplines that focus on the design of the physical system. Better integration between software engineering and the other engineering disciplines gives a number of advantages, which include:

- Reuse of models of the system (e.g., physical models) between different engineering stages and within embedded control software. This improves the efficiency with which embedded systems are developed.
- Better adjustment of the design of the physical system to the capabilities and limitations of software. The structure and functionality of embedded control software depends to a certain extent on the design of the physical system, e.g., on the structure of the physical system, on the physical interaction between components, etc. As such, the design of the physical system has an influence on the quality of the software. When the capabilities and limitations of software and software structuring techniques is taken into account in the early design of an embedded system, better design decisions for the physical system that result in better software quality can be made.

Further research is necessary on how to improve the integration of software engineering with the other engineering disciplines in embedded system development.



Terminology

A.1 Dependency Graph

A dependency graph is a directed graph structure that relates variables and equations in the physical model to each other.

Definition A.1.1 (Dependency Graph) *A dependency graph is an ordered pair* (vN, eN, E)*, where:*

- vN *is a set of* variable nodes.
- eN *is a set of* equation nodes.
- $vN \cap eN = \emptyset$
- *The total set of nodes* N *is defined as* $N = vN \cup eN$.
- E *is a set of edges, which are ordered pairs of nodes:*$E \subseteq N \times N$.

Furthermore, a proper dependency graph fulfills the following constraints:

- *There are no edges between two* equation nodes[1]*:* $E \cap eN \times eN = \emptyset$.
- *There are no self-edges:* $E \cap \{(n, n)|n \in N\} = \emptyset$.

In addition, the edges attached to equation nodes adhere to the following rules:

- *The equation node has edges to and/or from those and only those variable nodes that correspond to the variables in the equation.*
- *If the left-hand side of an equation contains only one variable, then the edges between the equation node and the variable nodes are directed:*
  - *For all variable nodes that correspond to variables on the right-hand side of the equation, the edge is directed from the variable node to the equation node.*

[1] Edges between two *variable nodes* are allowed. For certain applications of dependency graphs this is used to indicate that two connected variables have the same value.

  - *For the single variable node that corresponds to a variable on the left-hand side of the equation, the edge is directed from the equation node to the variable node.*

- *If the number of variables on the left-hand side of the equation is not equal to* 1*, there is no directional relationship between the equation node and the variable nodes. This is represented in the dependency graph as two edges, one in each direction*[2]*.*

Information about how a dependency graph is created from a SIDOPS+ specification is given in Section 2.4.2.

A.2 Derivation Graph

Definition A.2.1 (Derivation Graph) *A derivation graph of a physical model is a directed graph that reflects how the values of the physical variables in the physical model are derived from the values of other physical variables after the execution of a physical model instance. A derivation graph is an ordered pair* (vN, eN, E)*, where:*

- vN *is a set of* variable nodes*.*
- eN *is a set of* equation nodes*.*
- $vN \cap eN = \emptyset$
- *The total set of nodes* N *is defined as* $N = vN \cup eN$*.*
- E *is a set of edges, which are ordered pairs of nodes:*$E \subseteq N \times N$*.*

Information about the execution algorithm for physical model instances can be found in Section 2.4.5. Section 2.4.6 contains an example of a derivation graph.

[2]The visual representations of the dependency graph use two-headed single arrows to represent the existence of two edges, one in each direction.

Appendix B

Derivation of I_{gs}^{act}

The set of actual inputs to the goal system is limited by the adaptation process, as this process determines the i_a part of the input. The set of actual inputs to the goal system is given by the following equation:

$$I_{gs}^{act} = \{i_{gs} \in \mathcal{I}_{gs} | i_{gs}[k+1 \ldots k+m] = f_a(b_{ap})\}$$

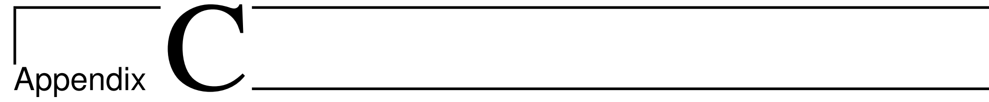
This equation can be rewritten into:

$$I_{gs}^{act} = \{i_{gs} \in \mathcal{I}_{gs} | i_{gs}[k+1 \ldots k+m] = f_a(s_{ap}(i_{gs}[1 \ldots k] || f_g(s_{gs}(i_{gs}))))\}$$

This is done through the following derivation steps. Each step is explaned below.

$$\begin{aligned}
I_{gs}^{act} &\stackrel{1}{=} \{i_{gs} \in \mathcal{I}_{gs} | i_{gs}[k+1 \ldots k+m] = f_a(b_{ap})\} \\
&\stackrel{2}{=} \{i_{gs} \in \mathcal{I}_{gs} | i_{gs}[k+1 \ldots k+m] = f_a(s_{ap}(i_{ap}))\} \\
&\stackrel{3}{=} \{i_{gs} \in \mathcal{I}_{gs} | i_{gs}[k+1 \ldots k+m] = f_a(s_{ap}(i_{gs}[1 \ldots k] || i_g))\} \\
&\stackrel{4}{=} \{i_{gs} \in \mathcal{I}_{gs} | i_{gs}[k+1 \ldots k+m] = f_a(s_{ap}(i_{gs}[1 \ldots k] || f_g(b_{gs})))\} \\
&\stackrel{5}{=} \{i_{gs} \in \mathcal{I}_{gs} | i_{gs}[k+1 \ldots k+m] = f_a(s_{ap}(i_{gs}[1 \ldots k] || f_g(s_{gs}(i_{gs}))))\}
\end{aligned}$$

1. The i_a input of the goal system is the filtered behavior of the adaptation process. So, the set contains those i_{gs} for which the i_a part (i.e., $i_{gs}[k+1 \ldots k+m]$) is equal to the filtered behavior of the adaptation process ($f_a(b_{ap})$).
2. By definition: $b_{ap} = s_{ap}(i_{ap})$.
3. By definition: $i_{ap} = i || i_g$. i is part of i_{gs}: $i = i_{gs}[1 \ldots k]$.
4. i_g is the filtered behavior of the goal system: $i_g = f_g(b_{gs})$.
5. By definition: $b_{gs} = s_{gs}(i_{gs})$.



Appendix C

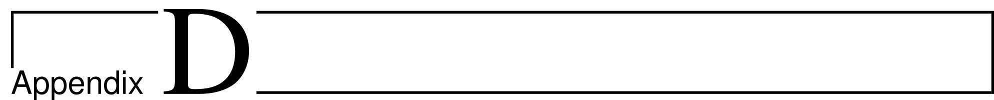
xADL Schema Extension

```
<xsd:schema xmlns="http://www.ewi.utwente.nl/~roo/mo2/mo2.xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:types= "http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
  targetNamespace="http://www.ewi.utwente.nl/~roo/mo2/mo2.xsd"
  elementFormDefault="qualified"
  attributeFormDefault="qualified">

<xsd:import namespace=
    "http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
  schemaLocation=
    "http://www.isr.uci.edu/projects/xarchuci/core/types.xsd"/>

  <xsd:annotation>
    <xsd:documentation>
      xArch MO2 XML Schema 1.0

      Change Log:


    </xsd:documentation>
  </xsd:annotation>

  <!--
    TYPE: MO2Component

    The MO2Component type is an extension of component and contains
        the properties 20SimReference, constraint,
    isOblivious and subModelReference.
   -->
  <xsd:complexType name="MO2Component">
    <xsd:complexContent>
      <xsd:extension base="types:Component">
        <xsd:sequence>
          <xsd:element name="20SimReference" type="StringValue" />
          <xsd:element name="constraint" type="MO2Constraint"
              minOccurs="0" maxOccurs="unbounded" />
          <xsd:element name="isOblivious" type="BooleanValue" />
          <xsd:element name="subModelReference" type="StringValue" />
        </xsd:sequence>
```

```
38        </xsd:extension>
39      </xsd:complexContent>
40    </xsd:complexType>
41
42
43
44    <!--
45      TYPE: MO2Interface (Port)
46
47     -->
48    <xsd:complexType name="MO2Interface">
49      <xsd:complexContent>
50        <xsd:extension base="types:Interface">
51          <xsd:sequence>
52            <xsd:element name="constraint" type="MO2Constraint"
                  minOccurs="0" maxOccurs="unbounded" />
53            <xsd:element name="isDecisionVariable" type="BooleanValue"
                  />
54            <xsd:element name="isObjective" type="BooleanValue" />
55          </xsd:sequence>
56        </xsd:extension>
57      </xsd:complexContent>
58    </xsd:complexType>
59
60
61
62
63
64    <!--
65      TYPE: MO2Constraint
66     -->
67    <xsd:complexType name="MO2Constraint">
68      <xsd:sequence>
69        <xsd:element name="operator" type="Operator" />
70        <xsd:element name="formula" type="StringValue" />
71      </xsd:sequence>
72    </xsd:complexType>
73
74    <xsd:simpleType name="Operator">
75      <xsd:restriction base="xsd:string">
76        <xsd:enumeration value="leq"/>
77        <xsd:enumeration value="less"/>
78        <xsd:enumeration value="equal"/>
79        <xsd:enumeration value="geq"/>
80        <xsd:enumeration value="greater"/>
81      </xsd:restriction>
82    </xsd:simpleType>
83
84    <!--
85      TYPE: StringValue
86
87
88     -->
89    <xsd:complexType name="StringValue">
90      <xsd:simpleContent>
91        <xsd:extension base="xsd:string"/>
92      </xsd:simpleContent>
93    </xsd:complexType>
```

```
94
95    <!--
96      TYPE: BooleanValue
97
98
99     -->
100   <xsd:complexType name="BooleanValue">
101     <xsd:simpleContent>
102       <xsd:extension base="xsd:boolean"/>
103     </xsd:simpleContent>
104   </xsd:complexType>
105
106
107 </xsd:schema>
```

Listing C.1: xADL Schema Extension



Object Models

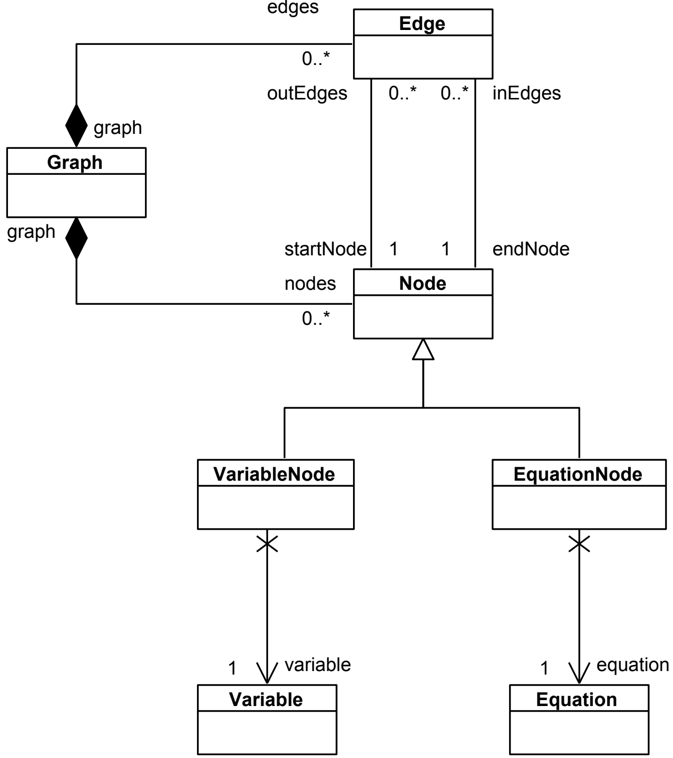


Figure D.1: Dependency/Derivation graph object model

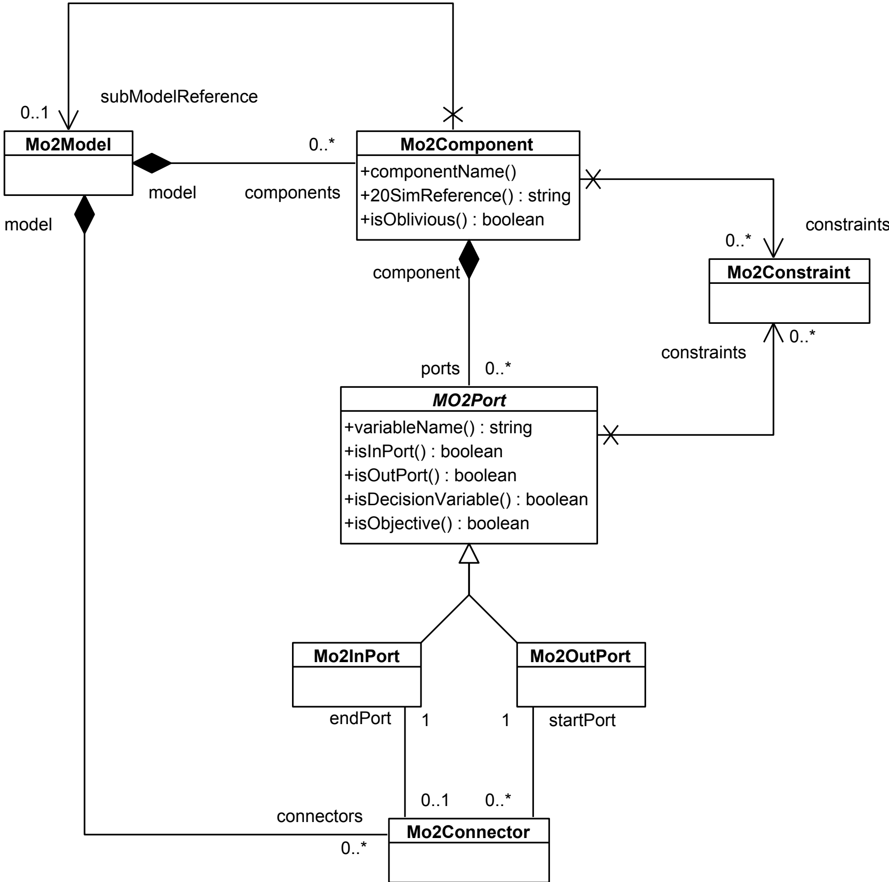


Figure D.2: MO2 object model

Generated Multi-Objective Optimization Code

This appendix presents the code that is generated for the multi-objective optimization control software implementation, demonstrated in Chapter 5. Listing E.1 shows the generated Java class. This class obtains *independent values* from the other modules in the implementation, and passes these values to the Matlab optimization function. Listing E.2 shows the Matlab function that performs the optimization. Note the complexity of the generated constraints and utility functions[1].

```
1  import com.mathworks.jmi.Matlab;
2
3  public class Optimizer{
4    ...
5
6    public Optimizer(...)
7    {
8      ...
9    }
10
11   private double[] getIndependentValues(){
12     double[] result = new double[6];
13     result[0] = _BeltTemperature.getValue("Tcontact");
14     result[2] = _PaperHeaterController.getPph();
15     result[4] = _TradeOffSlider.getWeight();
16     result[1] = _PhysicalSystemIO.getTph();
17     result[3] = _PhysicalSystemIO.getPavailable();
18     result[5] = _RadiatorController.getState();
19     return result;
20   }
21
22   private void communicateResult(double[] result){
23     _PaperPath.setV(result[0]);
24   }
25
26   private double[] previousResult = new double[] { 120d };
27
28   public void evaluate() throws Exception{
29     double[] indepValues = getIndependentValues();
30
```

[1]Be aware of the fact that this generated code is not optimized for human readability.

```
31     double[] result = (double[]) Matlab.mtFeval("optimize",
32       new Object[] { previousResult, indepValues }, 0);
33
34     communicateResult(result);
35     previousResult = result;
36   }
37 }
```

Listing E.1: Generated Java class

```
1 function result=optimize(previousResult,p)
2 A=[0];
3 b=[0];
4 Aeq=[0];
5 beq=[0];
6 lb=[1];
7 ub=[120];
8 result=fmincon(@utility, previousResult, A, b, Aeq, beq, lb, ub,
      @constraints);
9
10 function result=utility(x)
11 result=((power(((1/x(1))), p(5)) + power(((p(3) + (((10.0 *
      (((((((x(1) * ((0.210+0.040)/60)) * 50) + 106.6) + 10.0)) -
      (0.42 * p(2)))) - p(1))) + (0.1 * (p(6) + (((((((x(1) * ((0.210
      + 0.040)/60)) * 50) + 106.6) + 10.0)) - (0.42 * p(2)))) -
      p(1)))))))),(1 - p(5)))));
12 end
13
14 function [c, ceq]=constraints(x)
15 c=[0-((p(3) + (((10.0 * (((((((x(1) * ((0.210 + 0.040)/60)) * 50) +
      106.6) + 10.0)) - (0.42 * p(2)))) - p(1))) + (0.1 * (p(6) +
      (((((((x(1) * ((0.210+0.040)/60)) * 50) + 106.6) + 10.0)) -
      (0.42 * p(2)))) - p(1)))))))); 0 - ((p(4) - ((p(3) + (((10.0 *
      (((((((x(1) * ((0.210 + 0.040)/60)) * 50) + 106.6) + 10.0)) -
      (0.42 * p(2)))) - p(1))) + (0.1 * (p(6) + (((((((x(1) * ((0.210
      + 0.040)/60)) * 50) + 106.6) + 10.0)) - (0.42 * p(2)))) -
      p(1)))))))))); (((10.0 * (((((((x(1) * ((0.210 + 0.040)/60)) *
      50) + 106.6) + 10.0)) - (0.42 * p(2)))) - p(1))) + (0.1 * (p(6) +
       (((((((x(1) * ((0.210 + 0.040)/60)) * 50) + 106.6) + 10.0)) -
      (0.42 * p(2)))) - p(1)))))) - 1500; 0 - (((10.0 * (((((((x(1) *
      ((0.210 + 0.040)/60)) * 50) + 106.6) + 10.0)) - (0.42 * p(2)))) -
       p(1))) + (0.1 * (p(6) + (((((((x(1) * ((0.210 + 0.040)/60)) *
      50) + 106.6) + 10.0)) - (0.42 * p(2)))) - p(1)))))); x(1) -
      120;60 - x(1)];
16
17 ceq=[];
18 end
19
20 end
```

Listing E.2: Generated Matlab function

Bibliography

[1] The 20-sim tooling. http://www.20sim.com.

[2] Gnu linear programming kit. URL `http://www.gnu.org/software/glpk/`. [Online; accessed 19-March-2008].

[3] lp_solve. URL `http://lpsolve.sourceforge.net/`. [Online; accessed 19-March-2008].

[4] Find minimum of constrained nonlinear multivariable function. URL `http://www.mathworks.nl/help/toolbox/optim/ug/fmincon.html`.

[5] IBM rational rose technical developer. URL `http://www-01.ibm.com/software/awdtools/developer/technical/`.

[6] Simulink - simulation and model-based design. URL `http://www.mathworks.com/products/simulink/`.

[7] AspectJ. http://www.eclipse.org/aspectj/.

[8] Web services business process execution language version 2.0. OASIS Standard, April 2007.

[9] R. Abreu, P. Zoeteweij, and A. van Gemund. Spectrum-based multiple fault localization. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 88 –99, 16-20 2009. doi: 10.1109/ASE.2009.25.

[10] M. Akşit and A. Tripathi. Data abstraction mechanisms in sina/st. In *Proceedings of the conference Object-Oriented Systems, Languages and Applications*, volume 23 of *ACM Sigplan Notices*, pages 267–275, 1988.

[11] M. Akşit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In O. L. Madsen, editor, *Proc. 7th European Conf. Object-Oriented Programming*, pages 372–395. Springer-Verlag Lecture Notes in Computer Science, 1992. URL `http://trese.cs.utwente.nl/publications/paperinfo/LanguageDbase.pi.top.htm`.

[12] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. pages 152–184. Springer-Verlag, 1993.

[13] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094839.

[14] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36:81–94, November 1993. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/163359.163375. URL `http://doi.acm.org/10.1145/163359.163375`.

[15] T. Bapty, S. Neema, J. Scott, J. Sztipanovits, and S. Asaad. Model-integrated tools for the design of dynamically reconfigurable systems. Technical Report ISIS-99-01, Institute for Software Integrated Systems, Vanderbilt University, 1999.

[16] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In B. Steffen and G. Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004. ISBN 3-540-20803-8.

[17] T. Basten, E. van Benthum, M. Geilen, M. Hendriks, F. Houben, G. Igna, F. Reckers, S. de Smet, L. Somers, E. Teeselink, N. Trcka, F. Vaandrager, J. Verriet, M. Voorhoeve, and Y. Yang. Model-driven design-space exploration for embedded systems: The octopus toolset. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *Lecture Notes in Computer Science*, pages 90–105. Springer Berlin / Heidelberg, 2010. URL `http://dx.doi.org/10.1007/978-3-642-16558-0_10`. 10.1007/978-3-642-16558-0_10.

[18] Y. Berbers, P. Rigole, Y. Vandewoude, and S. V. Baelen. CoConES: CoConES: An approach for components and contracts in embedded systems. *Lecture Notes in Computer Science*, 3778:209–231, 2005.

[19] L. Bergmans and M. Akşit. Principles and design rationale of composition filters. In Filman et al. [49], pages 63–95. ISBN 0-321-21976-7.

[20] R. H. Bishop. *Modern Control Systems Analysis and Design Using MATLAB and SIMULINK*. Addison Wesley, 1996.

[21] C.-M. Bockisch. *An Efficient and Flexible Implementation of Aspect-Oriented Languages*. PhD thesis, Technische Universität Darmstadt, Germany, July 2008.

[22] N. Bouraqadi and T. Ledoux. Supporting AOP using reflection. In Filman et al. [49], pages 261–282. ISBN 0-321-21976-7.

[23] J. Broenink. Modelling, simulation and analysis with 20-sim. *Journal A*, 38(3): 22–25, 1997.

[24] F. Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10 –19, april 1987. ISSN 0018-9162. doi: 10.1109/MC.1987.1663532.

[25] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 569–588, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297069.

[26] M. K. Chmarra, J. Verriet, R. Waarsing, and T. Tomiyama. State transition in reconfigurable systems. *ASME Conference Proceedings*, 2010(44090):241–248, 2010. doi: 10.1115/DETC2010-28723. URL `http://link.aip.org/link/abstract/ASMECP/v2010/i44090/p241/s1`.

[27] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston, 2002. ISBN 0201703726.

[28] Y. Collette and P. Siarry. *Multiobjective Optimization: Principles and Case Studies*. Springer-Verlag, Berlin, first edition, 2003.

[29] D. Darcy, C. Kemerer, S. Slaughter, and J. Tomayko. The structural complexity of software: an experimental test. *Software Engineering, IEEE Transactions on*, 31(11):982 – 995, nov. 2005. ISSN 0098-5589. doi: 10.1109/TSE.2005.130.

[30] E. Dashofy, H. Asuncion, S. Hendrickson, G. Suryanarayana, J. Georgas, and R. Taylor. Archstudio 4: An architecture-based meta-modeling environment. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 67–68, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2892-9. doi: http://dx.doi.org/10.1109/ICSECOMPANION.2007.21.

[31] E. M. Dashofy, A. V. d. Hoek, and R. N. Taylor. A highly-extensible, xml-based architecture description language. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, page 103, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1360-3. doi: http://dx.doi.org/10.1109/WICSA.2001.948416.

[32] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/0004-3702(87)90063-4.

[33] A. de Roo. Towards more robust advice: Message flow analysis for composition filters and its application. Master's thesis, March 2007. URL `http://doc.utwente.nl/67050/`.

[34] A. de Roo, M. Hendriks, W. Havinga, P. Durr, and L. Bergmans. Compose*: a language- and platform-independent aspect compiler for composition filters. In *First International Workshop on Advanced Software Development Tools and Techniques, WASDeTT 2008, Paphos, Cyprus*, July 2008.

[35] A. de Roo, H. Sözer, and M. Akşit. An architectural style for optimizing system qualities in adaptive embedded systems using multi-objective optimization. In *Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture, WICSA/ECSA 2009*, pages 349–352, Cambridge, UK, 2009.

[36] A. de Roo, H. Sözer, and M. Akşit. Runtime verification of domain-specific models of physical characteristics in control software. In *Secure Software Integration and Reliability Improvement (SSIRI), 2011 Fifth International Conference on*, pages 41 –50, june 2011. doi: 10.1109/SSIRI.2011.14.

[37] N. Delgado, A. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *Software Engineering, IEEE Transactions on*, 30(12):859 – 872, dec. 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.91.

[38] L. Dohmen and L. Somers. Experiences and lessons learned using UML-RT to develop embedded printer software. In M. Oivo and S. Komi-Sirvi, editors, *Product Focused Software Process Improvement*, volume 2559 of *Lecture Notes in Computer Science*, pages 475–484. Springer Berlin / Heidelberg, 2002. ISBN 978-3-540-00234-5.

[39] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188. Springer Berlin / Heidelberg, 2002. ISBN 978-3-540-44284-4. URL http://dx.doi.org/10.1007/3-540-45821-2_11. 10.1007/3-540-45821-2_11.

[40] P. E. A. Durr. *Resource-based Verification for Robust Composition of Aspects*. PhD thesis, University of Twente, Enschede, 2008.

[41] R. T. Eckenrode. Weighting multiple criteria. *Management Science*, 12(3): pp. 180–192, 1965. ISSN 00251909. URL http://www.jstor.org/stable/2627577.

[42] F. Y. Edgeworth. *Mathematical Psychics: An Essay on the Application of Mathematics to the Moral Sciences*. C. Kegan Paul & Co., London, 1881.

[43] M. Ehrgott and X. Gandibleux. *Multiple criteria optimization: state of the art annotated bibliographic surveys*. Kluwer Academic Publishing, 2002. ISBN 1402071280.

[44] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127 – 144, jan 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2002.805829.

[45] Embedded System Institute. Octopus project website, 2012. URL `www.esi.nl/octopus`.

[46] M. Ezzeldin, P. van den Bosch, A. Jokic, and R. Waarsing. Model-free optimization based feedforward control for an inkjet printhead. In *2010 IEEE International Conference on Control Applications (CCA)*, pages 967 –972, sept. 2010. doi: 10.1109/CCA.2010.5611064.

[47] A. Feldman, G. Provan, and A. van Gemund. The Lydia approach to combinational model-based diagnosis. In *Proceedings of the Twentieth International Workshop on Principles of Diagnosis (DX'09), Stockholm Sweden*, pages 403–408. Erik Frisk and Mattias Nyberg and Mattias Krysander and Jan slund, June 2009.

[48] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1997. ISBN 0534954251.

[49] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7.

[50] K. D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24(1-3):85 – 168, 1984. ISSN 0004-3702. doi: DOI:10.1016/0004-3702(84)90038-9.

[51] M. Geilen, T. Basten, B. Theelen, and R. Otten. An algebra of pareto points. *Fundamenta Informaticae*, 78(1):35–74, 2007. ISSN 0169-2968.

[52] M. Glandrup. Extending C++ using the concepts of composition filters. Master's thesis, University of Twente, 1995. URL `http://trese.cs.utwente.nl/publications/paperinfo/glandrup.thesis.pi.top.htm`.

[53] C. Glaßer, C. Reitwießner, H. Schmitz, and M. Witek. Approximability and hardness in multi-objective optimization. In *Proceedings of the Programs, proofs, process and 6th international conference on Computability in Europe*, CiE'10, pages 180–189, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13961-2, 978-3-642-13961-1. URL `http://dl.acm.org/citation.cfm?id=1876420.1876440`.

[54] M. Goldman and S. Katz. Modular generic verification of ltl properties for aspects. *Proceedings of Foundations of Aspect Languages Workshop (FOAL06)*, 2006.

[55] J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, and B. Natarajan. An approach for supporting aspect-oriented domain modeling. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 151–168, New York, NY, USA, 2003. Springer-Verlag New York, Inc. ISBN 3-540-20102-5.

[56] K. Hatun, C. Bockisch, H. Sözer, and M. Akşit. A feature model and development approach for schedulers. In *Proceedings of the 1st workshop on Modularity*

in systems software, MISS '11, pages 1–5, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0647-8. doi: http://doi.acm.org/10.1145/1960518.1960520. URL http://doi.acm.org/10.1145/1960518.1960520.

[57] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development, Vancouver, Canada*, pages 85–95, New York, NY, USA, 2007. ACM Press. URL http://doi.acm.org/10.1145/1218563.1218574.

[58] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. *ACM SIGPLAN Notices*, 25(10):169–180, 1990.

[59] T. Henzinger, C. Kirsch, M. Sanvido, and W. Pree. From control models to real-time code using Giotto. *Control Systems, IEEE*, 23(1):50 – 64, feb 2003. ISSN 1066-033X. doi: 10.1109/MCS.2003.1172829.

[60] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 26–35. ACM Press, Mar. 2004. doi: http://doi.acm.org/10.1145/976270.976276.

[61] B. F. Hobbs. A comparison of weighting methods in power plant siting*. *Decision Sciences*, 11(4):725–737, 1980. ISSN 1540-5915. doi: 10.1111/j.1540-5915.1980.tb01173.x. URL http://dx.doi.org/10.1111/j.1540-5915.1980.tb01173.x.

[62] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley Professional, 1st edition, 2000.

[63] I. M. Holland. Specifying reusable components using contracts. pages 287–308. Springer-Verlag, 1992.

[64] A. Hommersom, P. J. F. Lucas, R. Waarsing, and P. Koopman. Applying Bayesian networks for intelligent adaptable printing systems. In M. Conti, S. Orcioni, N. M. Martnez Madrid, and R. E. E. D. Seepold, editors, *Solutions on Embedded Systems*, volume 81 of *Lecture Notes in Electrical Engineering*, pages 201–213. Springer Netherlands, 2011. ISBN 978-94-007-0638-5. URL http://dx.doi.org/10.1007/978-94-007-0638-5_14. 10.1007/978-94-007-0638-5_14.

[65] C. Hwang and K. Yoon. *Multiple attribute decision making: methods and applications: a state-of-the-art survey*, volume 13. Springer-Verlag, Berlin, 1981.

[66] ISO 80000-2:2009. *Quality management systems – Requirements*. ISO, Geneva, Switzerland, 2009.

[67] ISO 9001:2000. *Quality management systems – Requirements*. ISO, Geneva, Switzerland, 2000.

[68] ISO 9126-1:2001. *Software engineering – Product quality – Part 1: Quality model*. ISO, Geneva, Switzerland, 2001.

[69] R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley series in probability and mathematical statistics. John Wiley & Sons, Inc, New York, 1976.

[70] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[71] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 542–552, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7246-3.

[72] D. Kitchin, A. Quark, W. R. Cook, and J. Misra. The Orc programming language. In D. Lee, A. Lopes, and A. Poetzsch-Heffter, editors, *Proceedings of FMOODS/FORTE 2009*, volume 5522 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2009. ISBN 978-3-642-02137-4. doi: 10.1007/978-3-642-02138-1_1.

[73] C. Kleijn. *20-sim 4.1 Reference Manual*, 2009.

[74] G. Kniesel. Detection and resolution of weaving interactions. In A. Rashid and H. Ossher, editors, *Transactions on Aspect-Oriented Software Development V*, volume 5490 of *Lecture Notes in Computer Science*, pages 135–186. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-02058-2. URL `http://dx.doi.org/10.1007/978-3-642-02059-9_5`.

[75] H. Komoto and T. Tomiyama. Computational support for system architecting. *ASME Conference Proceedings*, 2010(44137):25–34, 2010. doi: 10.1115/DETC2010-28683. URL `http://link.aip.org/link/abstract/ASMECP/v2010/i44137/p25/s1`.

[76] P. Koopman. Embedded system design issues (the rest of the story). In *Computer Design: VLSI in Computers and Processors, 1996. ICCD '96. Proceedings., 1996 IEEE International Conference on*, pages 310 –317, oct 1996. doi: 10.1109/ICCD.1996.563572.

[77] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. *SIGSOFT Softw. Eng. Notes*, 29(6):137–146, 2004. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1041685.1029916.

[78] G. Lui, J. Yang, and J. Whidborne. *Multiobjective optimisation and control*. Research Studies Press, Baldock,Hertfordshire,England, 2002. ISBN 0863802648.

[79] J. M. Maciejowski. *Predictive control with constraints*. Prentice Hall, Essex, England, 2002.

[80] P. Maes. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 22(12):147–155, 1987.

[81] M. W. Maier, D. Emery, and R. Hilliard. Software architecture: Introducing IEEE Standard 1471. *IEEE Computer*, 34(4):107–109, 2001.

[82] S. Malakuti Khah Olun Abadi, C. M. Bockisch, and M. Akşit. Applying the composition filter model for runtime verification of multiple-language software. In *The 20th annual International Symposium on Software Reliability Engineering, ISSRE 2009, Mysore, India*, pages 31–40. IEEE Computer Society Press, 2009.

[83] J. Markovski, D. van Beek, R. Theunissen, K. Jacobs, and J. Rooda. A state-based framework for supervisory control synthesis and verification. In *Decision and Control (CDC), 2010 49th IEEE Conference on*, pages 3481 –3486, dec. 2010. doi: 10.1109/CDC.2010.5717095.

[84] R. Marler and J. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26:369–395, 2004. ISSN 1615-147X. URL `http://dx.doi.org/10.1007/s00158-003-0368-6`. 10.1007/s00158-003-0368-6.

[85] J. A. Martín H., J. Lope, and D. Maravall. Adaptation, anticipation and rationality in natural and artificial systems: computational paradigms mimicking nature. *Natural Computing*, 8:757–775, 2009. ISSN 1567-7818. URL `http://dx.doi.org/10.1007/s11047-008-9096-6`. 10.1007/s11047-008-9096-6.

[86] J. McCall, P. Richards, and G. Walters. Factors in software quality. Technical Report RADC TR-77-369, Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York, November 1977.

[87] S. McConnell. *Code Complete*. Microsoft Press, first edition, 1993.

[88] Merriam-Webster Online Dictionary. Complex, 2011. URL `http://www.merriam-webster.com/dictionary/complexity`. Retrieved May 9, 2011.

[89] Merriam-Webster Online Dictionary. Complexity, 2011. URL `http://www.merriam-webster.com/dictionary/complexity`. Retrieved May 9, 2011.

[90] G. A. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers*, pages 329–400. Academic Press, 1998.

[91] V. Pareto. *Cours D'Économie Politique*. F. Rouge, Lausanne, Switzerland, 1896.

[92] R. Passerone, W. Damm, I. Ben Hafaiedh, S. Graf, A. Ferrari, L. Mangeruca, A. Benveniste, B. Josko, T. Peikenkamp, D. Cancila, A. Cuccuru, S. Gerard, F. Terrier, and A. Sangiovanni-Vincentelli. Metamodels in europe: Languages, tools, and applications. *Design Test of Computers, IEEE*, 26(3):38 –53, May-June 2009.

[93] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32 (1):57–95, 1987. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/0004-3702(87) 90062-2.

[94] B. Ryder. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, SE-5(3):216 – 226, may 1979. ISSN 0098-5589. doi: 10.1109/TSE.1979.234183.

[95] T. L. Saaty. A scaling method for priorities in hierarchical structures. *Journal of Mathematical Psychology*, 15(3):234 – 281, 1977. ISSN 0022-2496. doi: DOI:10. 1016/0022-2496(77)90033-5. URL `http://www.sciencedirect.com/science/article/pii/0022249677900335`.

[96] B. Selic. Using UML for modeling complex real-time systems. In F. Mueller and A. Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 250–260. Springer Berlin / Heidelberg, 1998. ISBN 978-3-540-65075-1.

[97] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[98] E. D. Sontag. *Mathematical Control Theory: Deterministic Finite Dimensional Systems*. Springer, New York, second edition, 1998. ISBN 0-387-984895.

[99] M. Strzer and J. Krinke. Interference analysis for AspectJ. In *Proceedings of workshop FOAL 2003, held in conjunction with AOSD 2003*, 2003.

[100] R. Theunissen, R. Schiffelers, D. van Beek, and J. Rooda. Supervisory control synthesis for a patient support system. In *Proceedings of the European control conference*, 2009.

[101] University of Twente. COMPOSE* toolset. http://composestar.sourceforge.net.

[102] *Compose* Annotated Reference Manual*. University of Twente.

[103] A. van Deursen and P. Klint. Little languages: little maintenance. *Journal of Software Maintenance*, 10:75–92, March 1998. ISSN 1040-550X.

[104] W. van Dijk and J. Mordhorst. CFIST, Composition Filters in Smalltalk. Graduation Report, HIO Enschede, The Netherlands, May 1995.

[105] H. Voogd. *Multicriteria evaluation for urban and regional planning*. Pion Ltd, 1983.

[106] P. Ward and S. Mellor. *Structured development for real-time systems: Introduction & tools*. Yourdon Press computing series. Yourdon Press, 1985. ISBN 9780138547875.

[107] W. Weaver. Science and complexity. *American scientist*, 36(4):536–544, 1948.

[108] J. C. Wichman. The development of a preprocessor to facilitate composition filters in the Java language. Master's thesis, University of Twente, 1999. URL `http://trese.cs.utwente.nl/publications/paperinfo/wichman.thesis.pi.top.htm`.

[109] W. Winston and J. Goldberg. *Operations research: applications and algorithms*. Thomson Brooks/Cole, 2004. ISBN 978-0-534-42358-2.

[110] K. Yoon and C. Hwang. *Multiple attribute decision making: an introduction*. Sage Publications, 1995.

[111] M. Yoshioka, Y. Umeda, H. Takeda, Y. Shimomura, Y. Nomaguchi, and T. Tomiyama. Physical concept ontology for the knowledge intensive engineering framework. *Advanced Engineering Informatics*, 18(2):95 – 113, 2004. ISSN 1474-0346. doi: DOI:10.1016/j.aei.2004.09.004.

[112] E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press computing series. Prentice Hall, 1979. ISBN 9780138544713.

[113] L. Zadeh. On the definition of adaptivity. *Proceedings of the IEEE*, 51(3):469 – 470, March 1963. ISSN 0018-9219. doi: 10.1109/PROC.1963.1852.

[114] L. Zadeh. Optimality and non-scalar-valued performance criteria. *Automatic Control, IEEE Transactions on*, 8(1):59 – 60, jan 1963. ISSN 0018-9286. doi: 10.1109/TAC.1963.1105511.

[115] P. Zoeteweij, J. Pietersma, R. Abreu, A. Feldman, and A. van Gemund. Automated fault diagnosis in embedded systems. In *Secure System Integration and Reliability Improvement, 2008. SSIRI '08. Second International Conference on*, pages 103 –110, 14-17 2008. doi: 10.1109/SSIRI.2008.48.

Samenvatting

Het beheersen van de software complexiteit van adaptieve systemen

Om te overleven in de huidige markt zijn embedded systeem fabrikanten genoodzaakt om producten te ontwikkelen die zichzelf kunnen aanpassen aan veranderende behoeften van klanten, veranderende gebruiksomstandigheden en slijtage van componenten gedurende de levensduur van het product. Het is een uitdaging voor de ingenieurs en ontwerpers van deze bedrijven om dergelijke *adaptieve* producten te ontwikkelen zonder dat de productiekosten enorm toenemen. Om dit te bereiken worden er steeds geavanceerdere strategieën voor het regelen en besturen van embedded systemen ontwikkeld en geïmplementeerd. Een voorbeeld van zo'n geavanceerde strategie is het op *runtime* optimaliseren van meerdere systeemkwaliteiten (ook wel doelstellingen genoemd), zoals energieverbruik en productiviteit, aangevuld met het maken van afwegingen tussen zulke systeemkwaliteiten op basis van (veranderende) behoeften van de gebruiker.

De regel- en besturingsalgoritmen en -strategieën van embedded systemen zijn voor een groot gedeelte geïmplementeerd in software. Het implementeren van geavanceerde regel- en besturingsalgoritmen en -strategieën introduceert extra complexiteit in de software van embedded systemen. Deze complexiteit is deels onvermijdelijk: het is een resultaat van essentiële complexiteit in het geselecteerde regel- en besturingsalgoritme of -strategie. Echter, door het ontbreken van systematische methoden voor het ontwikkelen en implementeren van geavanceerde regel- en besturingsalgoritmen en -strategieën, en door het ontbreken van geschikte abstractie mechanismen in programmeertalen om deze algoritmen en strategieën op een conceptueel juiste manier uit te drukken wordt er ook extra, vermijdbare complexiteit geïntroduceerd. De kwaliteit van software, met betrekking tot kwaliteitscriteria zoals begrijpelijkheid, betrouwbaarheid, onderhoudbaarheid en herbruikbaarheid, wordt gereduceerd door vermijdbare complexiteit. Het onderwerp van dit proefschrift is hoe de extra complexiteit geïntroduceerd door geavanceerde regel- en besturingsalgoritmen en -strategieën beter beheerst kan worden en hoe de invloed van deze complexiteit op de kwaliteit van de software verminderd kan worden.

Dit proefschrift levert drie belangrijke bijdragen. Ten eerste wordt er een nieuwe techniek voorgesteld voor het componeren van domein-specifieke modellen van fysische karakteristieken (fysische modellen) met software modules geschreven in een algemene programmeertaal. Op deze manier worden de voordelen van domein-specifieke ab-

stracties in een domein-specifieke modeleertaal gecombineerd met de vrijheid die een algemene programmeertaal biedt. Ten tweede biedt dit proefschrift een methode om de correctheid van de modellen van fysische karakteristieken die gebruikt worden in de regel- en besturingssoftware van embedded systemen op runtime te verifiëren, omdat deze modellen fouten of onnauwkeurigheden kunnen bevatten. Ten derde presenteert dit proefschrift een systematische methode om oplossingen voor het gelijktijdig optimaliseren van meerdere systeemkwaliteiten te verwerken in de architectuur van de regel- en besturingssoftware van een embedded systeem.

Een methode om embedded systeem meer adaptief te maken is toestaan dat de waardes van een aantal geregelde fysische variabelen (bijvoorbeeld printsnelheid en bepaalde temperaturen noodzakelijk voor het correct printen in professionele printersystemen) in het embedded systeem kunnen variëren (binnen een bepaald bereik). In plaats van dat deze fysische variabelen naar bepaalde vaste waardes geregeld worden, biedt het toestaan van variaties in hun waardes de mogelijkheid om het systeem effectiever te laten werken in veranderende omstandigheden. Echter, correct gedrag van een embedded systeem is over het algemeen afhankelijk van een set van mathematische relaties tussen de fysische variabelen waaraan moet worden voldaan. Als het toegestaan is om de waardes van een aantal fysische variabelen te variëren, wordt het de verantwoordelijkheid van de regel- en besturingssoftware om te garanderen dat voldaan wordt aan deze mathematische relaties. De mathematische relaties zijn gebaseerd op de fysische karakteristieken van het systeem. Hierdoor worden modellen van deze fysische karakteristieken (fysische modellen) onderdeel van de regel- en besturingssoftware. In de praktijk worden er voornamelijk algemene programmeertalen (APTs), zoals C en C++, gebruikt voor het ontwikkelen van regel- en besturingssoftware voor embedded systemen. Hoewel een APT geschikt is voor het uitdrukken van generieke berekeningen, is het minder geschikt voor het uitdrukken van fysische modellen: Domein-specifieke abstracties worden vertaald naar algemene implementatie abstracties. Dit verminderd de begrijpelijkheid van de implementatie (in het bijzonder voor domeinexperts) en heeft een negatieve invloed op de evolueerbaarheid en herbruikbaarheid van de software. Daarbij kunnen domein-specifieke statische en dynamische controles niet meer effectief toegepast worden. Echter, er bestaan domein-specifieke modeleertalen (DSMTs) en bijbehorende applicaties voor het specificeren van fysische modellen. Voorbeelden hiervan zijn 20-Sim en Matlab Simulink. Deze talen en applicaties worden voornamelijk gebruikt voor het simuleren en documenteren van fysische systemen, en niet zozeer voor het implementeren van regel- en besturingssoftware voor embedded systemen. De reden hiervoor is dat deze talen minder geschikt zijn voor het uitdrukken van generieke logica in software. Daarnaast kunnen de modellen die uitgedrukt zijn in deze talen niet eenvoudig gecomponeerd worden met software modules geïmplementeerd in een APT. Dit proefschrift stelt een nieuwe techniek voor om 20-Sim modellen, welke fysische karakteristieken bevatten, toe te passen in de regel- en besturingssoftware voor embedded systemen en om deze modellen to componeren met software modules geschreven in een APT, door gebruik te maken van het Compositie Filters model. Deze techniek combineert de voordelen van DSMT voor het modelleren van fysische karakteristieken (bijv. domein-specifieke abstracties, hergebruik van modellen tussen fases in het ontwerpproces) met de vrijheid van een APT voor het implementeren van de generieke logica in software. Door

het toepassen van het Compositie Filters model is aspect-georiënteerde compositie van fysische modellen met software modules mogelijk, waardoor de afhankelijkheden tussen de software modules geabstraheerd en verminderd worden. Het Compositie Filters model biedt daarnaast een declaratief compositie mechanisme, waardoor de mogelijkheden voor het uitvoeren van statische analyse toenemen. Dit proefschrift biedt een methode waarmee de compositie filters die 20-Sim modellen met APT software modules componeren geanalyseerd kunnen worden. Dit is bijvoorbeeld nuttig voor het verifiëren van de compositie filters tijdens het ontwerpen of voor het genereren van code.

De geïmplementeerde fysische modellen komen mogelijk niet altijd overeen met de fysische realiteit, bijvoorbeeld doordat het fysische systeem veranderd is, doordat het systeem in andere omstandigheden gebruikt wordt dan waarvoor het getest is of doordat het fysische systeem veranderd is door slijtage. Omdat onnauwkeurigheden in fysische modellen kunnen leiden tot onjuist gedrag van het systeem moet de nauwkeurigheid van fysische modellen geverifieerd worden. Dit is het tweede probleem dat aangepakt wordt in dit proefschrift. Het is niet mogelijk om het systeem te testen en statisch te verifiëren voor alle mogelijke omstandigheden, wat verificatie tijdens het gebruik van het systeem (*runtime verificatie*) noodzakelijk maakt. Echter, traditionele runtime verificatie technieken kunnen niet toegepast worden, omdat deze als doel hebben om te verifiëren of een software systeem voldoet aan een model van het software systeem. De doelstelling van onze aanpak is het verifiëren of een model van fysische karakteristieken dat gebruikt wordt in software overeenkomt met de fysische werkelijkheid. Dit proefschrift presenteert een nieuwe aanpak voor het uitvoeren van runtime verificatie van fysische modellen in regel- en besturingssoftware voor embedded systemen waarbij redundantie in deze modellen (bijvoorbeeld redundante of overlappende sensor informatie) gebruikt wordt.

Van embedded systemen wordt verwacht dat ze zich optimaal gedragen onder veranderende omstandigheden, rekening houdend met de variërende behoeften van klanten. Daarom moeten deze systemen op runtime meerdere doelstellingen met betrekking tot de systeemkwaliteiten optimaliseren (zoals het maximaliseren van productiviteit en het minimaliseren van energieverbruik), en daarbij dynamisch afwegingen maken tussen deze doelstellingen. Er bestaan algoritmen voor het uitvoeren van optimalisatie van meerdere systeemkwaliteiten (*MOO algoritmen*). Echter, de systeem-brede invloed van deze MOO algoritmen gecombineerd met het ontbreken van systematische methoden voor het ontwerpen van optimalisatie van meerdere systeemkwaliteiten (*MOO oplossingen*) in de regel- en besturingssoftware voor embedded systemen leidt tot oplossingen die op maat gemaakt zijn voor het specifieke systeem en nauw geïntegreerd met de verschillende software modules. Dit proefschrift presenteert een systematische methode, genaamd de MO2 methode, om MOO oplossingen toe te voegen aan de architectuur van de regel- en besturingssoftware van een embedded systeem. De MO2 methode bestaat uit de MO2 architecturale stijl en een serie applicaties. De MO2 architecturale stijl maakt het mogelijk regel- en besturingsarchitecturen die MOO oplossingen bevatten te specificeren. De MO2 method bevat applicaties voor het bewerken van architecturale modellen volgens de MO2 stijl, voor het valideren van de consistentie van deze modellen met betrekking tot de geïmplementeerde MOO oplossing, en voor het genereren van een implementatie van

het optimaliseringsalgoritme in de regel- en besturingssoftware. De MO2 methode geeft ingenieurs de mogelijkheid om MOO oplossingen op een systematische manier te introduceren in regel- en besturingssoftware voor embedded systemen, en om MOO oplossingen en algoritmen te hergebruiken tussen verschillende systemen. Daarnaast ondersteunt de methode de documentatie van MOO oplossingen in de architectuur van de software.

De voorgestelde technieken in dit proefschrift zijn gevalideerd met een kwalitatieve evaluatie van het vermogen van deze technieken om de software complexiteit in adaptieve embedded systemen te beheersen. Realistische evolutie scenario's zijn gebruikt om een vergelijking te maken van de onderhoudbaarheid en evolueerbaarheid van software die gebruik maakt van huidige toegepaste technieken en van software die gebruik maakt van de technieken voorgesteld in dit proefschrift. Daarnaast is er een experiment uitgevoerd om de prestaties te testen van een systeem dat gebruikt maakt van de MO2 method voor optimalisatie. Er is hierbij een vergelijking gemaakt met andere systemen die gebruik maken van huidig toegepaste technieken voor optimalisatie.

Notes

“Technical skill is mastery of complexity, while creativity is mastery of simplicity.”

Erik Christopher Zeeman

ISBN: 978-90-365-3319-5