LUND UNIVERSITY

Languages and Tools for Optimization of Large-Scale Systems

Åkesson, Johan

2007

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Åkesson, J. (2007). *Languages and Tools for Optimization of Large-Scale Systems*. Department of Automatic Control, Lund Institute of Technology, Lund University.

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Languages and Tools for Optimization of Large-Scale Systems

Languages and Tools for Optimization of Large-Scale Systems

Johan Åkesson

Department of Automatic Control
Lund University
Lund, November 2007

To Gustav

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

ISSN 0280–5316
ISRN LUTFD2/TFRT--1081--SE

© 2007 by Johan Åkesson. All rights reserved.
Printed in Sweden by Media-Tryck.
Lund 2007

Preface

During the course of my PhD studies, I have deliberately sought research problems in diverse fields. I have also been involved in several projects, which I have benefited from by collecting experiences from different fields. My engagement in these projects has resulted in rewarding cooperations with fellow researchers. The impact of such collaborations are also evident in this thesis. In this preface, I briefly describe the projects in which I have been involved during my time as a PhD student.

One of my most important experiences in the area of control is the internship I did at the Department of Automatic Control in Lund, in the summer of 1999. The purpose of the internship was to develop a demonstrator of safe manual control, based on the Furuta pendulum process. Knowledge of control theory turned out to be important, but also quite insufficient in order to solve the problem at hand. In addition to control theory, understanding of computer control, sensors and electronics, and derivation of dynamic models of mechanical systems proved to be equally important. These initial experiences from hands-on control system development later became my main motivation to pursue further studies in the subject.

The projects that I have participated in include:

Safe manual control of pendula. Starting as a seven-week internship at the Department of Automatic Control and supervised by Karl Johan Åström, the work on safe manual control of pendula was expanded into my master's thesis. Initially, the main focus of the project was experiments, and the development of a demonstrator, but came to also involve some theoretical stability results. This project has been active, however with low intensity, throughout my time as a PhD student.

Operator support systems and grade changes. The second project was originally part of the EU-sponsored project CHEM. My part was to study operator support systems for grade changes in process

industry. The approach relied on model-based optimization and a tool for sequence control, JGrafchart. The ideas from this project later resulted in an industrial cooperation with Assidomän Frövi.

Assidomän Frövi. In an attempt to increase the industrial input to the operator support project, the paper mill Assidomän Frövi was approached. A cooperation was initiated, and together we ran a master's thesis project, that was quite successful. Nevertheless, in August of 2005, it was clear that the cooperation would not continue, due to Assidomän Frövi being acquired by another company.

Model predictive control and MPCTools. As part of the work on support systems for operators I worked on model predictive control (MPC). This work resulted in an MPC toolbox for Matlab, MPCtools. MPCtools has been freely available on-line since January 2006, and the download count is about 500, as of October 2007.

Integral action. As a spin-off from the work on MPC, I diverted into a study of control structures that guarantees integral action in MIMO systems. The results of the work also propagated back into MPCtools.

Real-time aspects of MPC. Following the tradition of integration of control and real-time systems at the department, I was involved in a project dealing with real-time aspects of MPC, where I collaborated with Dan Henriksson. As part of the project, MPCtools was integrated with TrueTime, an in-house toolbox for simulation of real-time control systems.

YAIP. Another on-going, mostly low intensity project during my time as a PhD student has been the construction of a miniature robot on two wheels (effectively an inverted pendulum on wheels). The idea originated from a course in Computer Mechatronics taught by Martin Nilsson, at SICS. The current design is based on (at least) two previous prototypes. The mechanical and electronics design, as well as the actual construction of the robot has been performed by Rolf Braun. In addition, Anders Blomdell has contributed many ideas on various design issues.

Paper machine modeling and DryLib. After the cooperation with Assidomän Frövi, I continued to work on paper machine modeling, but now in cooperation with Ola Slätteke, currently at Pöyry. My role in this project was not so much related to physical modeling (that topic is Ola's domain of expertise) but to the construction of a Modelica library, DryLib. In a second step, DryLib was used in a case

study on parameter optimization, and was also developed further in a master's thesis project.

JModelica and Optimica. After the termination of the cooperation with Assidomän Frövi, the JModelica project was initiated. The project was partly motivated by my experiences from working with dynamic optimization for large-scale systems, notably in the DryLib project. The coding effort in that project was significant, and the JModelica project was started based upon the hypothesis that improved high-level support for optimization in the Modelica language would yield improved productivity in the design process.

Start-up optimization of a plate reactor. Early on in the JModelica project, it was decided that a case study related to large-scale dynamic optimization should be performed, in order to provide a proof of concept. Staffan Haugwitz, who was working with control and optimization of a plate reactor system of suitable size and structure, agreed to use the JModelica/Optimica tools to solve a start-up problem that was part of his research. Staffan also became the first beta-tester of the Optimica compiler and supported the JModelica project by providing input from a user's perspective.

In this thesis, results from the following projects are presented: Safe manual control of pendula, YAIP, Paper machine modeling and DryLib, JModelica and Optimica, and Start-up optimization of a plate reactor. Material related to some of the other projects were presented in my Licentiate thesis [Åkesson, 2003]. The included material is organized into three main parts, which constitutes three out of five parts of the thesis: Languages and Tools for Dynamic Optimization, Case Studies, and Manual Control of Pendula. Together, these parts reflect some of the characteristic aspects of automatic control, ranging from high-level languages for physical modeling to implementation of control algorithms using fixed-point arithmetic for embedded processors. Given my motivations for pursuing PhD studies, I have enjoyed writing this thesis. I hope that you will find it equally enjoyable to read.

Johan Åkesson, October 2007

Abstract

Modeling and simulation are established techniques for solving design problems in a wide range of engineering disciplines today. Dedicated computer languages, such as Modelica, and efficient software tools are available. In this thesis, an extension of Modelica, Optimica, targeted at dynamic optimization of Modelica models is proposed. In order to demonstrate the Optimica extension, supporting software has been developed. This includes a modularly extensible Modelica compiler, the JModelica compiler, and an extension that supports also Optimica.

A Modelica library for paper machine dryer section modeling, DryLib, has been developed. The classes in the library enable structured and hierarchical modeling of dryer sections at the application user level, while offering extensibility for the expert user. Based on DryLib, a parameter optimization problem, a model reduction problem, and an optimization-based control problem have been formulated and solved.

A start-up optimization problem for a plate reactor has been formulated in Optimica, and solved by means of the Optimica compiler. In addition, the robustness properties of the start-up trajectories have been evaluated by means of Monte-Carlo simulation.

In many control systems, it is necessary to consider interaction with a user. In this thesis, a manual control scheme for an unstable inverted pendulum system, where the inputs are bounded, is presented. The proposed controller is based on the notion of reachability sets and guarantees semi-global stability for all references.

An inverted pendulum on a two wheels robot has been developed. A distributed control system, including sensor processing algorithms and a stabilizing control scheme has been implemented on three on-board embedded processors.

Acknowledgments

Firstly, I would like to express my gratitude to my supervisor Karl-Erik Årzén, for giving me the opportunity to pursue research in the areas that I have found interesting and stimulating. Karl-Erik's sound judgement and guidance have given me the confidence to take on challenges that I might have otherwise left unexplored. As my co-supervisor, Per Hagander has been the ultimate discussion partner, and I have very much enjoyed our, not always control-related, discussion sessions. Görel Hedin became my co-supervisor in the fall of 2006, and she has been my JastAdd mentor ever since. Görel's interest in discussing the JModelica project, implementational details as well as how to improve the clarity of the presentation, has very much affected the outcome of the corresponding parts of this thesis. I also would like to thank Karl Johan Åström for bringing my attention to the field of automatic control. When Karl Johan hired me as a trainee during the summer of 1999, he also introduced me to the exciting field of control, and for this I am very grateful.

Further, I would like to thank Staffan Haugwitz for a rewarding collaboration related to dynamic start-up optimization of a plate reactor. It is not feasible to mention Staffan without also mentioning his exceptional ability to make people, including me, get into that good mood that is characteristic of him. And accordingly, the plate reactor project was a true joyride. Magnus Gäfvert has been willing to discuss all kinds of Modelica-related topics, also the slightly more philosophical ones. It has been very rewarding. I also would like to thank Magnus for taking part in the discussion on the final version of the Optimica syntax and semantics. Torbjörn Ekman has helped me to sort out many issues related to JastAdd and the implementation of the JModelica compiler. His entusiasm for this project has been most appreciated. During the work on DryLib and parameter optimization of a paper machine model, I collaborated with Ola Slätteke and Jenny Ekvall. I would like to thank Ola and Jenny for eagerly educating me in the construction of physical paper machine models. In the

beginning of my time as a PhD student, I collaborated with Dan Henriksson, on a project in which we integrated MPCtools and TrueTime, a collaboration I very much enjoyed.

During 2004–2005, I was involved in a collaboration with Assidomän Frövi. This collaboration resulted in a master's thesis project, and I would like to thank Bengt Nilsson, Niklas Jansson and Lars Jonhed for giving me insight into the process of carton-board manufacturing.

I would like to thank the people at Modelon, including: Johan Andreasson, Jonas Eborn, Magnus Gäfvert, and Hubertus Tummescheit for interesting discussions on modeling in general and Modelica in particular. I also would like to thank the Dynasim crew, in particular Hilding Elmqvist, Hans Olsson, and Sven-Erik Mattson for helping me with various Dymola and Modelica-related issues.

Twice, I have had the pleasure of visiting Professor. Lorenz T. Biegler's research group at Carnegie Mellon University, Pittsburgh, USA. The work on dynamic optimization and Optimica has been very much influenced by my experiences from these visits. I would like to thank Professor Biegler, Carl Laird, Victor Zavala and Juan Arrieta for all the rewarding discussions that we have had, and I certainly look forward continue them in the future. I also would like to thank Carl for letting me stay in his house during my visit in January 2007, and for all the late-night discussions that we had.

It is a very pleasant experience to work at the Department of Automatic Control in Lund. I would like to thank my friends and colleagues at the Department for making my working days brighter and for eagerly discussing whatever issue I have come up with—control-oriented or not. Henrik Sandberg is the perfect room-mate, and I had the pleasure of sharing office with him around 2003-2004. Apart from being a terrific source of information about linear systems, he also served as the DJ of our room. He even played Gessle from time to time and, needless to say, he was thus a very good DJ. Pop on!

Leif Andersson keeps our computer systems healthy—his work has undoubtedly saved me many hours of computer hassle. Also, Leif is a keen teacher of the mysteries of Linux and LaTeX, and I have very much appreciated his teaching efforts. Anders Blomdell knows the answer to any questions about programming (at least as far as I can tell), and I would like to thank him for generously sharing his knowledge with me whenever I run into a segmentation fault. I also would like to thank Rolf Braun for letting me mess around in his mechanical workshop, it has been very rewarding. The dynamic trio on the fifth floor consisting of Eva Schildt, Britt-Marie Mårtensson and Agneta Tuszynski has helped me with numerous administrative (and other) matters. I don't know what the Department would be like without you girls, but I am quite sure that I

don't want to know. I also would like to Anders Rantzer for his stimulating and inspiring leadership.

I have benefited from the teaching of some truly outstanding teachers. Especially, I would like to mention Olle Christensson (physics, Lagaholmsskolan), Erland Ejder (mathematics and physics, Osbecksgymasiet), Ann-Sofi Gustavsson (Swedish language and literature, Osbecksgymnasiet), Margareta Jonsson (English language, Osbecksgymnasiet), Lars-Erik Jonsson (history, Osbecksgymnasiet), Sven Spanne (mathematics, LTH), and Robert E. Bitmead (automatic control, UCSD).

I am indebted to the following persons for reading manuscripts of this thesis, and for providing me with numerous comments and suggestions for improvements: David Broman, Anton Cervin, Magnus Gäfvert, Staffan Haugwitz, Carl Laird, Katarina Lundin Åkesson, Brad Schofield, Ola Slätteke, and Victor Zavala.

I would like to acknowledge the financial support provided by the EC/-GROWTH CHEM project, aimed at developing advanced decision support systems for different operator-assisted tasks, e.g., process monitoring, fault detection and isolation, and operations support, within the chemical process industries. The CHEM project is funded by the European Community under the Competitive and Sustainable Growth Programme (1998-2002) under contract G1RD-CT-2001-00466. The work presented in Chapter 10 was supported by the Swedish Research Council for Engineering Sciences, grant TFR 281-1998-618. I also would like to gratefully acknowledge financial travelling support from The Royal Physiographic Society in Lund, The Royal Swedish Academy of Sciences, Knut and Alice Wallenbergs Foundation, and Civilingenjören Hakon Hanssons Foundation.

Sometimes, when my mind has been a bit too set on work, my dear friends have helped me focus on non-control related essentials in life such as whisky, computer gaming and cooking. I am very happy to have you all. My parents, Lena and Bo-Evert Åkesson, have given me endless support, and always encouraged me to do the things I have set my mind to do. You are the best! Erik, my brother, balances my slightly more theoretical mindset by bringing my attention to more practical matters, such as driving tractors, welding, and lately, how to sell, en masse, fertilizers to the farmers in Scania. It is very rewarding to be your brother.

I also would like to express a special thanks to Katarina, for her support and encouragement. When my PhD studies were the most difficult, she gave me the strength to carry on. Finally, I would like to dedicate this thesis to you, my dear son, Gustav. You are the light of my life.

Johan

Contents

1. Introduction 19
1.1 Motivation and Objectives 20
1.2 Contributions 22
1.3 Managing Complexity 26
1.4 Modeling and Simulation 30
1.5 Manual Control 36
1.6 Organization of the Thesis 42

Part I. Background 43

2. Modelica 45
2.1 Simulation of Large Dynamic Systems 46
2.2 The Modelica Translation Process 47
2.3 The Modelica Language 48
2.4 Graphical Annotations 56
2.5 Modelica Tools 57

3. Dynamic Optimization 59
3.1 Calculus of Variations and The Maximum Principle . . 60
3.2 Direct Methods 63
3.3 Applications 73
3.4 Tools for Dynamic Optimization 75
3.5 Summary 77

Part II. Languages and Software Tools 79

4. JModelica – A Modelica Compiler 81
4.1 Introduction 81
4.2 Objectives 86
4.3 Development Platform—JastAdd 86
4.4 JastAdd 88
4.5 An Executable Specification 100

4.6 PicoModelica . . . 101
4.7 The JModelica Compiler . . . 102
4.8 Benchmarks . . . 104
4.9 Extensibility of the JModelica Compiler . . . 106
4.10 Summary and Conclusions . . . 108
5. Modelica Name and Type Analysis with JastAdd . . . 110
5.1 Ambiguous Names . . . 111
5.2 Classification of Names . . . 112
5.3 Binding Names . . . 117
5.4 Type Analysis – Subtype Computation . . . 127
5.5 Summary . . . 131
6. Flattening of Modelica Models with JastAdd . . . 132
6.1 Simplified Flattening without Modifications . . . 133
6.2 Flattening and Modifications . . . 134
6.3 The Instance AST . . . 136
6.4 Conceptual Construction of the Instance AST . . . 138
6.5 Construction of the Instance AST Using NTA:s . . . 138
6.6 Modification Trees . . . 142
6.7 Environments . . . 143
6.8 Declarative Construction of Merged Environments . . . 146
6.9 Handling of Structural Modifications . . . 149
6.10 A Flattening Algorithm . . . 153
6.11 Summary . . . 155
7. Optimica . . . 156
7.1 Introduction . . . 156
7.2 Motivation of the Optimica Extension . . . 158
7.3 Scope of Optimica . . . 160
7.4 The Optimica Extension . . . 164
7.5 The Optimica Compiler . . . 171
7.6 Examples . . . 175
7.7 Generalizations . . . 181
7.8 Summary and Conclusions . . . 183
Part III. Case Studies . . . 185
8. DryLib . . . 187
8.1 Introduction . . . 187
8.2 Physical Modeling . . . 189
8.3 DryLib . . . 196
8.4 Parameter Optimization . . . 203
8.5 Model Reduction . . . 209
8.6 NMPC of Output Moisture . . . 218
8.7 Software Tools . . . 222
8.8 Summary and Conclusions . . . 224

9. Start-up Control of a Plate Reactor 225
9.1 Introduction 225
9.2 The Plate Reactor 228
9.3 Problem Formulation 233
9.4 The Optimization Problem 235
9.5 Feedback Control 247
9.6 Simulation with Feedback Control 250
9.7 Monte Carlo Simulations 253
9.8 Summary and Conclusions 256
Part IV. Control of Pendula 257
10. Safe Manual Control of an Inverted Pendulum 259
10.1 Introduction 259
10.2 Equations of Motion 260
10.3 Reachability Set Analysis 262
10.4 A Stabilizing Controller 265
10.5 Tracking 266
10.6 Extensions 270
10.7 Conclusions 272
11. Design and Control of YAIP 273
11.1 Introduction 273
11.2 System Design 274
11.3 Encoder Processing 276
11.4 Gyro and Accelerometer Processing 282
11.5 Dynamic System Model 282
11.6 Stabilizing Control 284
11.7 Implementational Issues 285
11.8 Experimental Results 286
11.9 Conclusions and Suggested Improvements 287
Part V. Conclusions 289
12. Conclusions and Future Work 291
12.1 Languages and Tools for Dynamic Optimization 291
12.2 Case studies 293
12.3 Control of Pendula 294
13. Bibliography 296
A. Collocation and Runge-Kutta Methods 310
B. PicoModelica Syntax and Abstract Grammar 313
B.1 Concrete Syntax 313
B.2 Source Abstract Grammar 314
B.3 Instance Abstract Grammar 315
B.4 Flat Abstract Grammar 315
C. Complete AST for the Examples in Chapter 5 316

D. A Flat Optimica Description for an Optimization Problem 317
E. A Modelica Model of a Plate Reactor 322

1

Introduction

Automatic control is a research field that intersects with several other disciplines, such as mathematics, physics, electronics, and computer science. Another characteristic feature of automatic control is that systems are studied at different levels of detail. In some applications, it is necessary to consider the behavior of sensor and actuator hardware, filters implemented by means of analog electronics, low-level control loops, and the properties of the embedded computing hardware on which signal processing and control algorithms are implemented. Other applications require plant-wide models composed from hundreds of processing units and their interconnections, in order to solve design problems at the system-level. An additional aspect of automatic control is the extensive use of computers, since literally all control functions are implemented in software. Accordingly, modern control engineering requires a wide frame of reference, where understanding of products and processes at the system-level, as well as detailed knowledge of sub-systems at the component-level are required. This thesis reflects the richness of the field, and is composed of five parts:

Part I. Background

Part II. Languages and Software Tools

Part III. Case Studies

Part IV. Control of Pendula

Part V. Conclusions

The main contributions of the thesis are given in Parts II–IV. Part II is concerned with computer languages and tools for modeling, simulation, and optimization of large-scale systems. The results presented are strongly related to the engineering need for high-level descriptions in order to manage increasing system complexity, as well as the need to enable use of efficient numerical algorithms based on high-level model descriptions.

In Part III, two cases studies are presented. The work in this part is focused on optimization of large-scale systems, where it is shown how efficient numerical algorithms for dynamic optimization can be applied to Modelica models. In addition, the case studies in Part III serve as a motivation for the language and tool development in Part II. Finally, in Part IV, theoretical and experimental results on manual control of pendula are presented. This work is based on classical control theory, mechatronics and signal processing. Certainly, Part IV is not directly related to Parts II-III, which will also be reflected in this introductory chapter, where the backgrounds of these subjects will be given separately. Nevertheless, if the four parts are considered together, they cover characteristic aspects in the field of automatic control, ranging from sensor-processing and low-level control loops to modeling and optimization of large-scale systems.

This chapter is organized as follows. The main objectives of this thesis are stated in Section 1.1, and its contributions are listed in Section 1.2. In Section 1.3, the current and future need for languages and tools to manage the increasing complexity of products and process systems is discussed. Section 1.4 gives an overview of modeling and simulation, and future directions in the field are discussed. In Section 1.5, control systems where humans are part of the control loop are introduced. This section serves as an introduction to the work on manual control of inverted pendula in Chapters 10 and 11. The chapter ends with an outline of the thesis in Section 1.6.

1.1 Motivation and Objectives

Languages

There is currently an increasing need to integrate different languages, tools and algorithms in the engineering design process. Also, the increasing complexity of systems calls for high-level description frameworks. Based on the observation that Modelica is becoming an increasingly used language for modeling, a main objective in this thesis is to extend Modelica with new constructs, in order to accommodate formulation of dynamic optimization problems. The proposed extension is entitled Optimica. While dynamic optimization is an important extension of Modelica, there are certainly other extensions which are equally important. Modularity of the Optimica extension, both at the language level and at the implementation level, is therefore considered as an additional objective.

Software

In order to demonstrate the effectiveness of the proposed extension Optimica, a related objective is to develop prototype software tools which translate Modelica/Optimica descriptions into a format suitable for efficient numerical solution. The requirements for this software is essentially to perform a series of transformation steps based on the Modelica/Optimica source code, including flattening of Modelica models and code generation. In particular, it was decided that the compiler should produce as its output a description of a dynamic optimization problem in AMPL [Fourer *et al.*, 2003], format. This in turn enables efficient use of state of the art numerical solvers. A desired property in the software design is that of modularity of the implementation. It was therefore set out to develop a compiler for pure Modelica, entitled the JModelica compiler, and in addition, a modular extension thereof, referred to as the Optimica compiler.

Case Studies

The specific target of the proposed language extension is to enable optimization of large-scale dynamic systems. In order to demonstrate the feasibility of the proposed concept, an additional objective of the thesis is to perform case studies based on such systems. In the first case study, which is based on a large-scale model of a paper machine dryer section, the objectives have been to perform model calibration by means of parameter estimation to reduce the complexity of the model using an optimization approach, and to implement an on-line optimization moisture control scheme. This work was performed prior to the development of the Optimica compiler, and involved a significant coding effort in standard programming languages, primarily in C. As such, this case study also inspired the work on Optimica. The second case study deals with start-up of a plate reactor. The main objective is to start the highly exothermic reaction without violating specified temperature bounds, also in the face of modeling errors. In order to meet this objective, careful design of the objective function and constraints in the start-up optimization problem is required. In the context of this thesis, a secondary objective of this case study has been to show the effectiveness of the Optimica extension, when applied to a realistically sized application.

Safe Manual Control of Pendula

Control of unstable systems subject to input saturation and manual control is challenging, and also, many real processes belongs to this class. In Part IV, this class of systems is represented by inverted pendula. The theoretical part of the work is concerned with developing a control structure

which achieves semi-global stability, in the case of manual control. In particular, the objective is to derive such a controller for a planar pendulum system.

The experimental work in Part IV is concerned with design and control of an inverted pendulum on a two wheels robot. The problem is challenging, in particular since accurate control relies on high-precision sensors and actuators. The main objective has been to build a system with high performance, in terms of maneuverability, while using low-cost components.

These areas do have interconnections, but the presented results may also be considered as separate contributions. Indeed, the attempts to solve optimization problems using software dedicated to simulation, in the applications related to DryLib, motivated the work on Optimica and the JModelica compiler. The work on start-up optimization of the plate reactor then followed as a consequence of the progress made with the software tools development.

1.2 Contributions

This thesis contains contributions mainly in three areas:

- Languages and software tools.
- Two case studies dealing with modeling and optimization of a paper machine dryer section and a plate reactor, respectively.
- Safe manual control of an inverted pendulum and control of a pendulum on two wheels robot.

In this section, the contributions of this thesis will be presented in some detail.

The JModelica Compiler

A Modelica compiler front-end, the JModelica compiler, capable of transforming Modelica source code into a flat representation has been developed. The compiler supports a subset of the Modelica language and is developed using the compiler construction framework JastAdd [Hedin and Magnusson, 2003]. In this work, Torbjörn Ekman provided expertise in issues related to JastAdd, and also, some of the strategies used for name and type analysis originates from the JastAddJ compiler, [Ekman and Hedin, 2007], but were adapted to the context of Modelica. An overview of this work is given in Chapter 4, and details on the implementation are given in Chapters 5 and 6. Part of the work is based on the publication:

Åkesson, J., T. Ekman, and G. Hedin (2007): "Development of a Modelica compiler using JastAdd." In *Seventh Workshop on Language Descriptions, Tools and Applications*. Braga, Portugal.

Based on this paper, the authors have been invited to submit an extended manuscript to the Elsevier journal Science of Computer Programming.

Optimica and the Optimica Compiler

An extension of the Modelica language, Optimica, has been defined in order to support formulation of dynamic optimization problems based on Modelica descriptions. In addition, a modular extension of the JModelica compiler has been implemented which supports Optimica. The Optimica language definition is presented in Chapter 7. An overview of the work on Optimica and a draft specification of Optimica have been presented in:

Åkesson, J. (2007): "Dynamic optimization of modelica models – language extensions and tools." In *1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, Linköping Electronic Conference Proceedings. Linköping University Electronic Press, Linköping, Sweden.

Åkesson, J. and K.-E. Årzén (2007): "Tools and languages for modeling and optimization of large-scale dynamical systems." In *23rd IFIP TC 7 Conference on System Modelling and Optimization*. Krakow, Poland.

DryLib and Applications

A Modelica library for modeling of paper machine dryer sections, DryLib, has been developed. The mathematical model upon which the library is based was developed by Ola Slätteke in [Slätteke, 2006], and the contribution given in this thesis is concerned with structuring of the model into a reusable and extensible Modelica library. This work was done in cooperation with Ola Slätteke, where Slätteke provided expertise in the area of physical modeling of paper machines. Also, DryLib can be viewed as an evolution of a Modelica model that Slätteke developed during his PhD work. In addition, three application examples related to optimization are presented, namely parameter estimation, model reduction and non-linear model predictive control. The work on parameter optimization was done in cooperation with Jenny Ekvall, where Ekvall provided measurement data and process expertise. This work is presented in Chapter 8, and is based on the publications:

Åkesson, J. and J. Ekvall (2006): "Parameter optimization of a paper machine model." In *Proceedings of Reglermöte 2006*. Stockholm, Sweden.

Åkesson, J. and O. Slätteke (2006a): "A Modelica Library for paper machine dryer section modeling—drylib — and applications." Technical Report ISRN LUTFD2/TFRT--7615--SE. Department of Automatic Control, Lund University, Sweden.

Åkesson, J. and O. Slätteke (2006b): "Modeling, calibration and control of a paper machine dryer section." In *5th International Modelica Conference 2006*. Modelica Association, Vienna, Austria.

Start-up Optimization of a Plate Reactor

The applicability and effectiveness of the compiler tools for Modelica and Optimica have been demonstrated in a case study, where these tools have been used to compute start-up trajectories for a plate reactor. The process is challenging both due to its size in terms of the number of equations as well as due to its non-linear dynamics. In this context, the Optimica compiler proved to be a valuable tool by providing a flexible environment promoting high-level formulation of the optimization problem rather than attention to the details of the numerical algorithms. This work was done in close cooperation with Staffan Haugwitz, where Haugwitz provided process expertise, and the author of this thesis provided the Optimica compiler, and expertise in the area of dynamic optimization. The material is presented in Chapter 9 and is based on the publications:

Haugwitz, S., J. Åkesson, and P. Hagander (2007a): "Dynamic optimization of a plate reactor start-up supported by Modelica-based code generation software." In *Proceedings of 8th International Symposium on Dynamics and Control of Process Systems*. Cancun, Mexico.

Haugwitz, S., J. Åkesson, and P. Hagander (2007b): "Dynamic start-up optimization of a plate reactor with uncertainties." *Journal of Process Control*. Submitted.

Manual Control of an Inverted Pendulum

A controller enabling safe manual control of an inverted pendulum system has been developed. The controller is based on an explicit characterization of the reachability set of the system when subject to input saturation. The controller guarantees semi-global stability in a region of the state space which is an arbitrarily large subset of the reachability set. The controller also provides a mechanism for trading performance for robustness. This material is presented in Chapter 10 and is based on the publications:

Åkesson, J. (1999): "Safe reference following on the inverted pendelum." Technical Report ISRN LUTFD2/TFRT--7587--SE. Department of Automatic Control, Lund University, Sweden.

Åkesson, J. (2000): "Safe manual control of unstable systems." Master's Thesis ISRN LUTFD2/TFRT--5646--SE. Department of Automatic Control, Lund University, Sweden.

Åkesson, J. and K. J. Åström (2001): "Safe manual control of the Furuta pendulum." In *Proceedings 2001 IEEE International Conference on Control Applications (CCA'01)*, pp. 890–895. Mexico City, Mexico.

Åkesson, J. and K. J. Åström (2005): "Manual control and stabilization of an inverted pendulum." In *Proc. 16th IFAC World Congress*. Prague, Czech Republic.

YAIP – An Inverted Pendulum on Two Wheels Robot

A pendulum on two wheels robot, entitled YAIP, has been designed. Important parts of this work have included selection of mechanical and electrical components. The current design is the result of at least two preceding hardware prototypes. The mechanical and electrical design, as well as the construction of the robot has been performed by Rolf Braun, and many ideas have been contributed by Anders Blomdell. The control system and sensor processing algorithms have been designed and implemented by the author. The work is presented in Chapter 11 and is based on the publication:

Åkesson, J., A. Blomdell, and R. Braun (2006): "Design and control of YAIP—An inverted pendulum on two wheels robot." In *Proceedings of the IEEE International Conference on Control Applications*. Munich, Germany.

Other Contributions

Other contributions by the author, or in which the the author has taken part, but which are not included in this thesis include:

Åkesson, J. (2003): "Operator interaction and optimization in control systems." Licentiate Thesis ISRN LUTFD2/TFRT--3234--SE. Department of Automatic Control, Lund University, Sweden.

Åkesson, J. and K.-E. Årzén (2004): "A framework for grade changes: An optimization and sequential control approach." In *Proceedings of ESCAPE-14*. Lisbon, Portugal.

Henriksson, D. and J. Åkesson (2004): "Flexible implementation of model predictive control using sub-optimal solutions." Technical Report ISRN LUTFD2/TFRT--7610--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Åkesson, J. (2006): "MPCtools 1.0—Reference Manual." Technical Report ISRN LUTFD2/TFRT--7613--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Åkesson, J. and P. Hagander (2003): "Integral action – A disturbance observer approach." In *Proceedings of European Control Conference*. Cambridge, UK.

Henriksson, D., A. Cervin, J. Åkesson, and K.-E. Årzén (2002a): "Feedback scheduling of model predictive controllers." In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*. San Jose, CA.

Henriksson, D., A. Cervin, J. Åkesson, and K.-E. Årzén (2002b): "On dynamic real-time scheduling of model predictive controllers." In *Proceedings of the 41st IEEE Conference on Decision and Control*. Las Vegas, NV.

1.3 Managing Complexity

The complexity of industrial systems as well as consumer products is increasing rapidly. This trend is driven by increasingly competitive markets, where reducing development time and production costs while increasing product quality and adding functionality have become not only means to increase profits, but also necessities for survival of companies. Accordingly, improved methods and procedures for development of new products and operation of existing processes are of integral importance. In particular, the problem of managing increasing system complexity needs to be addressed. This complexity stems both from the increasing *size* of systems, and increasing *heterogeneity* of systems. For example, it is not uncommon for a vehicle model to include 30.000-40.000 equations and it is commonly composed of mechanical, hydraulic and electronic subsystems.

Computer languages and software for supporting the engineering design process have been available for several decades, and followed the introduction of the digital computer in the late fifties. Today, computer based design tools are available for any engineer that needs them, and accordingly, such tools are extensively used. It is also clear that designs of products and processes made today, would not have been possible without the support of computer software.

Several challenges in this area lie ahead, however, and some of them will be discussed in this section. A striking observation can be made if the impact of computer software on the efficiency of the design process,

is compared to the performance increase of CPUs. The latter development is commonly considered to follow Moore's law, which says that the number of transistors (which is closely related to, for example, computing performance) that can be fit onto an integrated circuit is doubled every two years. This rule has been valid for at least 35 years, and accordingly, the increase in performance of digital computers is about $2^{35/2} \approx 185.000$ during this period. This number can be compared to the gain in productivity that follows the introduction of modern design software, such as Matlab/Simulink, which is about 5-30, [Åström, 2007]. Of course, it can be argued that this is not a fair comparison, but it may still be an indication that in the increase in computing hardware performance, lies a challenge for the design-software community to take full advantage of this development.

In this section, some aspects of *design-software frameworks* will be discussed. Design-software framework here refers to software developed to support the engineering design process in a wide sense, but special attention will be given to model-based approaches. In particular, the growing demand for design-software frameworks which are based on a unifying approach, supporting high-level model and problem descriptions and which enable use of a wide range of algorithms will be discussed.

Large-scale systems

Modern component-based modeling languages enable detailed modeling at the *component-level* as well as effective management of model complexity at the *system-level*. Typically, expert knowledge, at the component-level, is encoded by experts in a particular application domain into generic and reusable model libraries. As a result, high-fidelity models can be constructed, at the system-level, by application engineers with a reasonable effort. The resulting models may be very large and complex, since the behavior of the model sub-systems are typically modeled using a high level of detail. Performing transformations and operations, such as simulation, on such systems is inherently challenging simply because the number of variables and equations is potentially very large. Currently, the need to model processes with a high level of detail is increasing. This development is largely driven by the advantages derived from performing extensive assessments regarding performance, safety and operation cost before the product or process is constructed. A design-software framework must therefore be able to handle, efficiently, the complexity of current and future large-scale models.

Heterogeneity of systems

Historically, several design software packages have been developed, targeted at a particular application domain. For example, SPICE [Nagel and

Pederson, 1973], was developed for modeling and simulation of analog electric circuits. SPICE later evolved into VHDL-AMS [IEEE, 1997], which is an object-oriented modeling language, used mainly to model analog and discrete electric circuits. In the domain of mechanical modeling, the software package ADAMS [MSC Software, 2007] is commonly used, and in chemical engineering gPROMS [Process Systems Enterprise, 2007] has a dominating position. While these tools offer tailored solutions for their respective domains, there is currently an increasing need to consider *interactions of sub-systems in different application domains*. As an illustrative example, consider a car. In order to perform, for example, simulation, it might be necessary to consider not only a mechanical model of the body of the car, but also the hydraulic braking system, the electronic engine control systems, and maybe even combustion. In addition, a modern car is equipped with several processors, ECUs, which execute embedded software code. As a consequence, accurate simulation of the car behavior requires a *heterogeneous* approach to system modeling and design. This conclusion holds for modern product and process design in general, since it is typical that the overall behavior of the system needs to be assessed prior to construction.

Heterogeneity of Methods and Algorithms

There are currently a significant number of methods, tools and algorithms for performing various design tasks. Such tasks include, for example, design optimization, control system design, model reduction, model calibration, and hardware in the loop simulation (HILS). A common characteristic for many of these methods and algorithms is that they rely on a *model* of the physical system at hand. However, the model *format* that they support may differ significantly. It is therefore not uncommon that the same system has to be modeled several times in order to take advantage of different design tools. This situation is troublesome, since it is time consuming to build several models of the same system, and it also rises issues about *consistency* of the models. Managing this type of heterogeneity of methods and algorithms is a key challenge that needs to be addressed.

Core Technologies

In this thesis, a few important techniques are explored in order to address some of the issues discussed in this section. In particular, these techniques are used extensively in Chapters 4-7 in the context of dynamic optimization of Modelica models.

High-level languages. In order to effectively, and efficiently, manage the increasing complexity, as described above, high-level description languages are necessary. Such languages should preferably be able

to handle large, complex, and heterogeneous systems. The issue of extending existing languages, in order to enable modeling of not only systems, but also modeling of specific design problems is important in this context.

Compilers. In order to transform high-level descriptions into a format suitable for algorithms, compilers are necessary. Modern compiler construction tools enable high-level encoding of the semantics of computer languages, which makes compiler development less time consuming, more flexible and in addition, the resulting compiler specification can be interpreted as an executable specification. The core data structure of compilers, the syntax tree, also forms the basis for symbolic and structural computations such as equation sorting and automatic differentiation. In addition, compilers are used for code generation, which is an important technique in this context.

Modular extensibility. It is a reasonable assumption that new languages and design methodologies will emerge. It is therefore important to develop design-software frameworks that are extensible, in order to meet demands for new functionality. In addition, it is desirable to make extensions in a modular fashion, both at the language level and at the compiler implementation level. A well known modularization principle in compiler construction is the organization of the compiler into front-ends and back-ends. A compiler front-end translates source code written in a particular language, into a canonical code representation format[1]. A compiler back-end then translates the intermediate code into, for example, machine code for a particular hardware architecture. The main point is that this design decouples front-ends from back-ends, which means that it is sufficient to develop one front-end for each language and one back-end for each architecture, instead of one compiler for each language-architecture combination. This methodology is applicable also in the case of design-software frameworks, but in this case the intermediate data format should describe a canonical representation of a design problem, or model. In addition, back-ends are developed in order to interface with different algorithms.

Code generation. As argued above, there is a large number of algorithms that are useful in the design process, although the programming APIs of these algorithms may be incompatible and in some cases also cumbersome to use. In order to avoid large programming

[1] One famous example of intermediate code representation is P-code, or pseudo code, which was introduced at UCSD in 1978.

efforts associated with encoding a particular problem for a particular algorithm, it is desirable to use automatic code generation based on high-level descriptions. The user is then relieved from the burden of managing algorithm APIs directly, but can rather focus on formulating the design problem, instead of encoding it. Another target of code generation may be embedded processors, in which case the high-level description language is used to specify, for example, a control system.

1.4 Modeling and Simulation

A key enabling technology for managing complexity is modeling and simulation. Mathematical models are extensively used in many engineering fields, as a standard design methodology. Building models, and the ability to simulate their behavior, enables assessments of performance, safety and operation to be done before a process is constructed. As a consequence, the design can be thoroughly tested, evaluated and optimized before proceeding to the implementation phase. Modeling and simulation is also used to model processes that have already been constructed. In this case, operations such as bottle-neck analysis, what-if scenario analysis and process optimization are common. Modeling and simulation can also, in some cases, replace experiments, which are not feasible to perform for safety or economical reasons. In this section, the history of modeling and simulation will be briefly reviewed. For a more thorough presentation of the subject, see [Åström *et al.*, 1998].

Analog Simulation

The field of modeling and simulation originates from *analog simulation*, which was used as a method for simulating dynamic systems, primarily from the 1920s to 1970s. Initial methods were based on mechanical systems, see for example [Bush, 1931]. The predominant technique, however, during the period was simulation by means of analog electric circuits. A dynamical system was then described in terms of integration, addition, subtraction and multiplication by constants, which are all operations that can be performed using basic electric components, such as operational amplifiers. Setting up such simulation experiments is tedious. Not only is it necessary to rearrange the description of the dynamic system so that it can be mapped onto electrical circuits, scaling is also necessary. The latter need stems from the fact that analog circuits have limited range. Another common problem, which is still highly relevant in many modern modeling frameworks, is that of algebraic loops. In order to simulate, by

means of analog simulation, a system containing algebraic loops, these must be eliminated, which in effect means that the algebraic variables must be eliminated in the mathematical description.

Numerical Integration

The advent of digital computers in the 1950s and 1960s dramatically changed the area of modeling and simulation. In particular, *numerical integration* became a standard technique for simulation of dynamic systems. Since then, the field of numerical integration has been an active area of research. Many integration algorithms have been proposed, including the family of Runge-Kutta schemes and multi-step methods such as the backward difference formula (BDF). Particular attention has been given to the issue of *numerical stability* of integration algorithms. It has been shown, for example, that implicit solvers often have superior stability properties, as compared to explicit solvers. In particular, this is important for stiff system. On the other hand, implicit solvers require solution of a system of equations at each time step which is computationally demanding. The choice of an appropriate integration algorithm is therefore dependent on the properties of the system to be integrated, see [Gustafsson, 1994] for a discussion. *Ordinary differential equations* on the form

$$\frac{dx}{dt} = f(x) \tag{1.1}$$

has been the subject of much of the research on numerical integration. In addition the more general class of *differential algebraic equations* on the form

$$g(\dot{x}, x, y) = 0 \tag{1.2}$$

where $\dot{x}$ denotes the time derivative of x, has received increasing attention. The variables x here denotes variables which appears differentiated with respect to time, whereas y denotes algebraic variables. Differential algebraic systems arise naturally in physical modeling, and in particular this formulation admits algebraic loops to be included. Closely related to DAEs is the notion of *index*. Roughly, the index of a DAE defines how many times a particular set of equations in the DAE have to be differentiated in order to solve for the differentiated variables. For a monograph on properties of DAEs and their numerical solution, see [Brenan *et al.*, 1996]

Another important extension of numerical integration algorithm is event-handling, which arises in mixed continuous-discrete, or hybrid, systems. The task of the numerical integration algorithm is then extended to include both integration of a continuous time system in between events, and detection of event instants, see [Barton and Lee, 2002].

Modeling and Simulation Frameworks

The advent of digital computers and numerical integration algorithms triggered the development of modeling and simulation languages and software. An early initiative was CSSL, [Strauss, 1967], which served as a unifying framework for modeling and simulation. CSSL formed a de facto standard for simulation of continuous time systems for a long time, and was used as a basis for, for example, the commercial tool ACSL, [Mitchell and Gauthier, 1976]. CSSL provided the user with a macro language and a set of built-in operators for expressing integration, time delay and hysteresis etc.

Another paradigm was used in Simnon, [Elmqvist, 1975], where continuous and discrete systems can be modeled in state space form. Simnon allows *connections* between subsystems, and variables within each subsystem are categorized as inputs, outputs or states. Simnon also has some other nice features, such as variable sampling rate of discrete blocks, automatic sorting of equations in order to find the correct evaluation order, and in addition, parameters may be changed without the need to recompile the model.

Graphical Block-Based Modeling Graphical block-based modeling has been used extensively in several engineering domains during the last two decades, partly due to the success of Matlab/Simulink. This paradigm has many common features with analog simulation using electrical components, and is based on block diagrams composed of integrators, gains, summation blocks, function generators etc. A particular feature of block-based models is that the causality of each connection is fixed, which means that the user must decide the direction of the data flow. While this is natural when modeling, for example, control systems, it is not convenient for modeling physical systems. In effect, the block-based modeling paradigm has inherited some of the inherent limitation of analog simulation. Most importantly, algebraic loops cannot be easily handled, which means that the user must solve for the right hand side of the differential equation.

Physical Modeling The above mentioned modeling and simulation frameworks can be viewed as evolutions of analog simulation, that followed the introduction of digital computers. In order to better match the needs associated with *physical modeling*, several other paradigms have emerged. One of the most significant drawbacks with the previously described methods, for example block-based modeling, is that when algebraic loops are eliminated, the structure of the physical system may be destroyed. Typically, a physical system is composed from a number of subcomponents, which interact through *acausal* connections. By keeping this

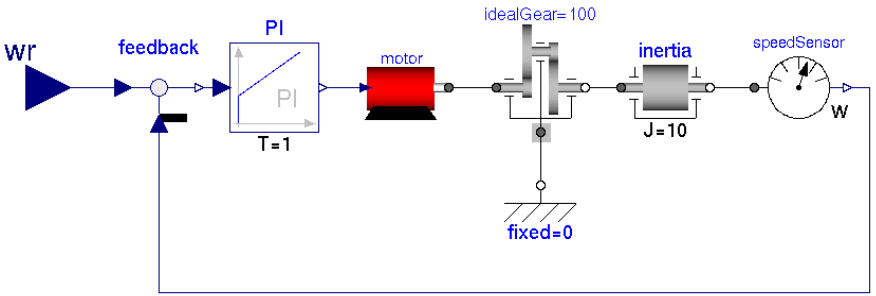


Figure 1.1 A Modelica model of a DC-motor connected to a load, controlled by a PI controller.

structure intact in the modeling framework, the mapping between the model and the physical plant is preserved.

An early description language for physical modeling is bond graphs, [Karnopp and Rosenberg, 1968]. Using bond graphs, subsystems are defined by nodes, and the edges between sub-systems (nodes) represent power flow. Bond graphs support acausal connections, and offer mechanisms to express connections involving *flow* variables (the sum of all variable values in a connection equals zero) and *potential* variables (all variables in a connection have the same value). These concepts are equivalent to Kirchhoff's laws of voltage and current, but are applicable also to other domains, such as mechanical connections. In the latter case, the torque/force variables represents the flow variables, and the angle/position variables represents the potential variables.

A modeling paradigm that has won widespread use during the last two decades is object-orientation. Object-orientation originates from the field of computer science, where it had been used for 20 years prior to its introduction in the field of physical modeling. In the context of physical modeling, object-orientation is used primarily as a *structuring concept*. One of the underlying ideas in this paradigm is to enable *reuse* of models. The fundamental concept in object-orientation, the class, is typically used to create libraries of physical counterparts, such as pumps, pipes and valves, see for example [Nilsson, 1993]. In this way, the procedure of building a model resembles the procedure of composing the actual system from physical components – having selected the sub-components from the library of model classes, the structure of the system could be encoded by specifying acausal connections between the components. Using this methodology, a model class could be reused in several contexts, without the need to explicitly solve for the dynamic variables. In addition, object-oriented modeling

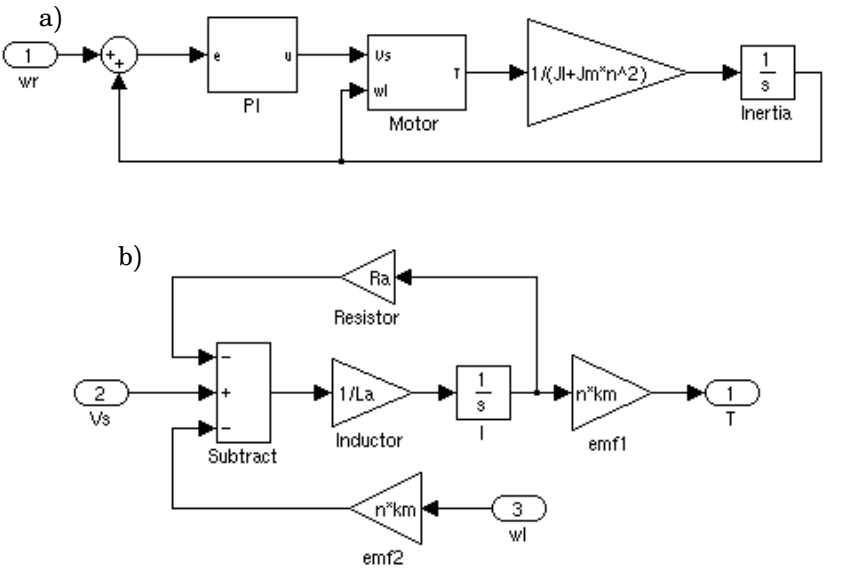


Figure 1.2 A Simulink model of a DC-motor connected to a load, controlled by a PI controller. Diagram a) shows the control loop, whereas diagram b) shows the motor model.

offers *inheritance* of classes, which is useful to specialize models by adding additional variables and equations to a base class. These concepts have been applied in several languages and tools, including Omola, [Andersson, 1994], Dymola, [Elmqvist, 1978], VHDL-AMS, [IEEE, 1997] and Modelica, [Elmqvist *et al.*, 1998].

EXAMPLE 1.1
To illustrate the difference between block-based modeling and and object-oriented component-based modeling, consider a DC-motor connected to a load, controlled by a PI controller. The example, including the dynamic equations can be found in [Åström *et al.*, 1998]. Figure 1.1 shows the component-based Modelica model of the system, and Figure 1.2 shows the corresponding system modeled in Simulink. Notice how the physical structure of the system is preserved in the Modelica model, but not in the Simulink model. □

While the object-oriented modeling paradigm is very appealing for the user, the resulting models are usually hierarchical, and cannot immediately be interfaced with numerical integration algorithms. In this respect,

models constructed using a block-based approach are more easily simulated, since algebraic loops have already been eliminated, and explicit functions for the state derivatives can be obtained. In contrast, component-based models with acausal connections typically yield DAEs, potentially with high index, which further complicates simulation. Therefore, a sophisticated machinery is required in order to translate such models into a format suitable for numerical integration. The foundations of such a machinery involves structural analysis of the DAE, equation sorting, symbolic solution of equations whenever possible, and tearing which is a technique for reducing the complexity of the block structure of the resulting DAE. These techniques were available in an early implementation of Dymola, [Elmqvist, 1978]. Since then, a technique called index-reduction has been introduced, which enables efficient translation and integration also of higher-index systems. The impact of this type of software on the design process is significant. In particular, it has shifted the focus of the user from *encoding* of the problem to *formulation* of the problem.

Hybrid Modeling An important development in modeling and simulation is support for mixed continuous and discrete time behavior. This extension is orthogonal to existing modeling paradigms in the sense that it can be incorporated both in an object-oriented component-based framework, as well as in the block-based framework. Since many physical systems exhibit hybrid behavior, this development has proven extremely useful, and greatly increased the scope of systems which can be effectively modeled. Examples of phenomena which are intrinsically hybrid are friction and switched equipment, for example on/of valves and gear-boxes. The introduction of support for hybrid systems in languages and tools followed from the development of numerical integration algorithms which are capable of efficiently and accurately simulate systems with state-events. Typically, such an algorithm monitors a set of *zero-crossing functions*, and detects events by locking on time-instants when one of these functions changes sign. At an event-instant, discrete actions may take place. For example, states may be re-initialized, or the set of equations governing the dynamics may be changed.

Future Challenges

There has been significant developments in the field of modeling and simulation during the last fifty years. Largely this development has been driven by the availability of increasingly powerful computers, which have enabled simulation of large and complex systems. Also, concepts from the field of computer science, such as languages and compilers, have been introduced which has led to further improved productivity in the engineering design process. However, important challenges lie ahead. Supported by

the arguments discussed in this section, it is clear that existing tools and languages need to be developed further in order to meet the increasing need to manage large-scale, complex and heterogeneous models. Certainly, there are strong initiatives in this direction, notably the development of the Modelica language [The Modelica Association, 2007b]. Furthermore, there is a wide range of design algorithms in various application fields, ranging from control systems design to numerical optimization, that are currently difficult to apply to large-scale models encoded in high-level languages. A major challenge is therefore to enable use of such algorithms based on existing and future high-level model descriptions, in order to further increase productivity in the design process. For these reasons, the field of modeling and simulation is likely to continue its evolution, into *modeling and model-based design*, where simulation remains an important usage of models, but where other design methodologies are also supported. Development of software that implements unifying model-based design framework is therefore one of the major challenges in this field.

1.5 Manual Control

Even if the bulk of the research in control has been devoted to fully automatic systems, there are many situations where humans are an essential part of the feedback loop. Typical examples are flight control systems, where the pilot is a key element and process control where operators are essential. Modern cars with advanced safety systems for distance and lane keeping are other examples where humans closely interact with the underlying control system.

Flight control is an area where the effect of humans has been investigated for a long time. Pioneering research on the behavior of humans in tracking tasks was done by Tustin in [Tustin, 1947], which was followed by intensive research for several decades, [McRuer and Krendel, 1959; Wilde and Westcott, 1962; Hall, 1963; McRuer *et al.*, 1965; Kleinman *et al.*, 1970; Baron *et al.*, 1970; Miall *et al.*, 1993; Sheridan and Lunteren, 1997]. These studies had a major impact on the design of autopilots. One interesting results is that the human adapts. For example, if the lag in the process increases, the human will introduce more derivative action. In [Gaines, 1969] it was argued that the adaptation could also be interpreted as a consequence of non-linear behavior.

There has also been research on failures of control systems. The interest in this area increased significantly after the Three Mile Island and the Chernobyl incidents, see [Rausmussen, 1983] and [Moray *et al.*, 1990] and references therein.

In this thesis, a special case of interaction between manual and automatic control is considered, where the process is unstable and the controls are bounded. If precautions are not taken, the system may be driven into saturation by manual control, with lost stability as a result. Such a control system can be decomposed into two main functions. The first, and most important function, is to ensure stability, regardless of the commands given by the user/pilot. Secondly, the control system should offer high performance in terms of maneuverability for the user/pilot. Clearly, these objectives are contradictory, and design of a control system which ensures stability but which does not unnecessarily restrict the performance is therefore a major challenge.

Typical examples representing this class of systems are high performance air crafts, exothermic chemical reactors and nuclear reactors. The inherent difficulty with these problems have been discussed in [Stein, 2003]. In particular, there has been major problems related to control of unstable (in some flight conditions) air crafts such as the Saab Gripen [Rundqwist *et al.*, 1997] and the YF-11 [Patcher and Miller, 1998]. Air crafts are challenging examples because of the extreme requirements of maneuverability, and that the unstable modes are too fast for the pilot to stabilize manually.

The characteristic features of the systems described above can be captured by significantly simpler systems, however. In particular, the inverted pendulum will be used to demonstrate the ideas presented in Chapter 10. While an inverted pendulum system may differ significantly from a fighter aircraft, it still shares the key characteristic features of being unstable and having bounded controls.

An Example

The idea explored in Chapter 10 will be illustrated here by an example. Although simple, this example reveals much of the structure of the problem, and provides important insight that will be used in the following. Consider the first order system

$$\dot{x} = x + \text{sat}(u). \tag{1.3}$$

The system is unstable, and the control signal is limited by the standard saturation function

$$\text{sat}_a(u) = \begin{cases} -a & u < -a \\ u & -a \leq u \leq a \\ a & u > a \end{cases} \tag{1.4}$$

If a is not given explicitly, the default choice of $a = 1$ is assumed. A characteristic feature of the system (1.3) is that there are initial states,

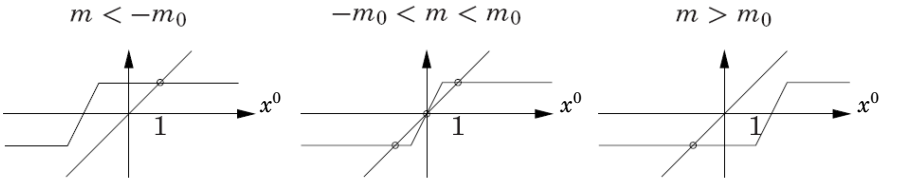


Figure 1.3 Different values of m results in three different equilibria configurations.

$x(0)$, from which the system cannot be brought back to the origin. This feature is typical for unstable systems subject to bounded feedback control. In this case, it must be required that $|x(0)| < 1$. If this is not the case, it is not possible to recover the state of the system to the origin.

Introduce the set $V_s = \{x : |x| < 1\}$, consisting of all points in the state space from which the state may be transferred to the origin by feasible controls. This set is called the backwards reachability set. Since the state space can be divided into the reachability set and its complement, it is clear that it is impossible to design a controller that achieves global stability. A realistic, but more restricted goal is then to find a controller that guarantees that the state remains in V_s, if the initial state is in V_s. This stability concept is referred to as semi-global stability, and implies that the state of the system is confined to an *a priori* known subset of the state space. This subset may coincide with the reachability set or it might be smaller.

Now, for $x \in V_s$, the system is stabilized by the control law $u = -lx$, provided that $l > 1$. It is worth noting that when x is close to the equilibrium points $x = \pm 1$, the solution departs slowly towards the origin. This means that recovery to the origin is slow if x is driven close to the borders of V_s. This phenomenon is called sticking, see [Patcher and Miller, 1998]. Sticking is typical for systems subject to manual control, although it is not present in all such systems.

To investigate the effect of manual control we consider the control law

$$u = -lx + m, \tag{1.5}$$

where m represents the manual control signal. It is clear that m may drive the system into saturation, which in turn may cause the system to leave V_s if no precautions are taken. The equilibria of the system controlled by the control law (1.5) are given by

$$x^0 = \text{sat } (lx^0 - m). \tag{1.6}$$

Assuming that m is constant, then depending on the magnitude of m, there are three cases, as illustrated in Figure 1.3. In the cases of $m < -m_0$

or $m > m_0$, $m_0 = l - 1$, there is one unstable equilibrium. It is then clear that the system is unstable for such (constant) values of m. In the case when $-m_0 \leq m \leq m_0$ there are two unstable equilibria and one stable equilibrium. In this case, trajectories for which $x(0) \in V_s$ will converge to the stable equilibrium. The key to preserving stability of the system, i.e., enforce that $x \in V_s$, is to limit the influence of the manual control term m. This is actually quite natural, since maintaining stability is the primary task of the controller, whereas manual control is secondary and should be allowed only when there is no risk of jeopardizing stability. Limiting of m may simply be achieved by an additional saturation function, with appropriate limit, acting on m, that ensures that the state does not leave V_s. In practice, it is convenient to introduce a safety margin, δ, ensuring that the state cannot get arbitrarily close to the border of V_s. This makes sense for two reasons. Firstly, for robustness reasons it is reasonable not to get too close to the border of V_s. Secondly, the sticking effect makes it undesirable to force the state close to the border of V_s. A sensible control law is then

$$u = -lx + \text{sat}_{m_0-\delta}\ (m). \tag{1.7}$$

This choice guarantees that $x \in V_s$, regardless of m. It is worth noticing that the stability preserving property of the controller is independent of the choice of m, which may not be a constant but rather a function of time and/or state variables. The situation is further illustrated in Figure 1.4, where the solid lines show the equilibrium point corresponding to constant values of m.

It remains to discuss what choices of m give desirable tracking properties. Although stability is enforced by the control structure (1.7), nothing is said about convergence of the state to a desired set point. Introducing

$$m = l_r r, \tag{1.8}$$

with $l_r = l - 1$, gives perfect steady state tracking of constant values of r. Notice, however, that zero-error tracking, in stationarity, can be achieved only if

$$-\frac{m_0 - \delta}{l - 1} \leq r \leq \frac{m_0 - \delta}{l - 1}$$

holds.

In Figure 1.5, the response of the system (1.3) controlled by the control law (1.7) with the manual control term defined by equation (1.8) is shown. In the simulation, the parameter values $l = 5$, yielding $m_0 = 4$ and $\delta = 0.4$ is used. Notice the slow response to the negative reference change, which is typical for systems exhibiting sticking.

To summarize the insight gained from this motivational example, three main points were made:

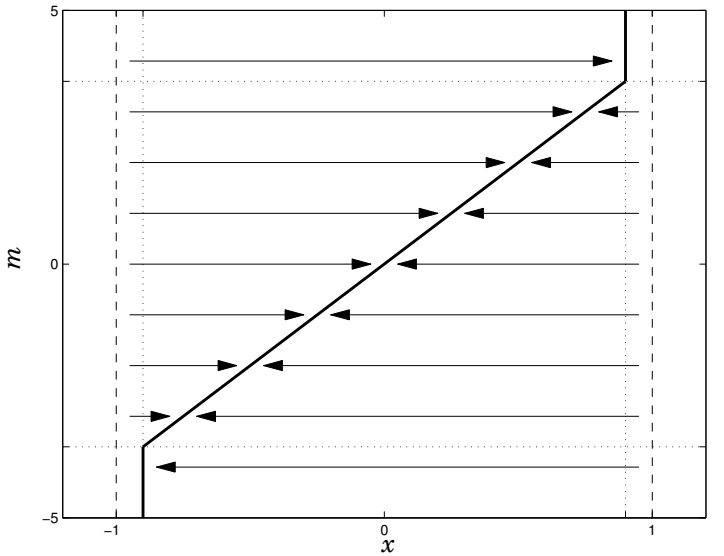


Figure 1.4 For constant values of m, the solid lines correspond to the equilibrium value of the state. The dashed lines indicate the border of V_s, whereas the vertical and horizontal dotted lines indicate maximum and minimum equilibrium values given the control law (1.7) and inner saturation limits $\pm(m_0 - \delta)$ respectively.

- The notion of reachability sets, which define a set of points in the state space from which recovery to the origin is possible with bounded controls, is essential.
- The introduction of manual control in a stabilizing feedback control law may easily drive the system into saturation with escape of the state as a consequence.
- If the influence of the manual control term is limited, stability can be preserved.

Reachability Sets

There is an extensive literature on stabilizing a dynamical system subject to input and/or state constraints. For linear systems, the problem is well understood. For asymptotically stable systems, the stabilization problem is trivial, since the state of the system will converge to the origin if the control is set to zero. Of course, in order for this to be a feasible strategy when the controls are bounded, the value zero must be feasible given the bounds. For unstable systems without poles strictly in the right complex

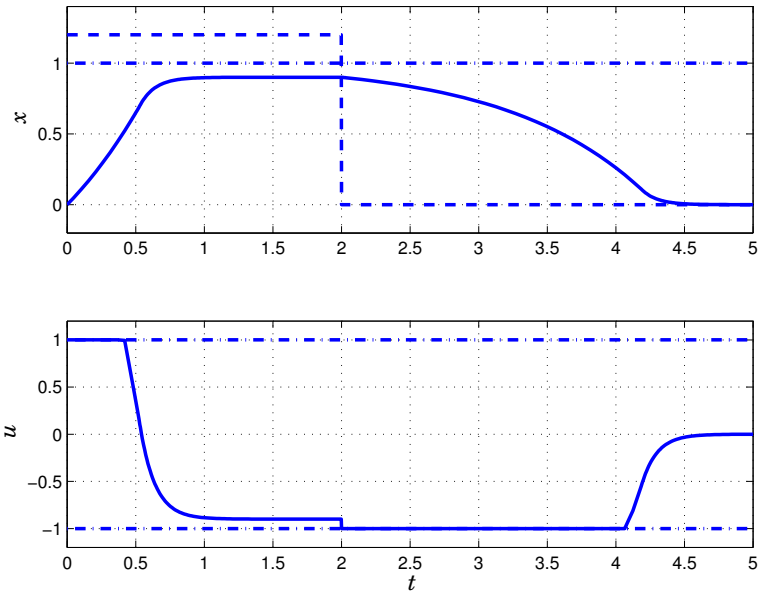


Figure 1.5 State trajectory and control signal in solid, state reference in dashed and the positive boundary of V_s in dash dotted.

half plane, which include systems with poles on the imaginary axis, there exist controllers that achieve global stability. The result was proven for a chain of integrators in [Teel, 1992] and for the general case in [Sussmann *et al.*, 1994]. For linear exponentially unstable systems, the situation is more involved. A key concept for control of unstable systems subject to bounded inputs is the notion of *reachability sets*.

It can be shown that the reachability set of a linear exponentially unstable system, subject to bounded controls, is bounded in the directions of the unstable modes. Consequently, only semi-global stability may be achieved. An elegant result for calculation of reachability sets for exponentially unstable systems as well as a method of semi-global stabilization are given in [Hu *et al.*, 2001].

The problem of calculating reachability sets can be cast as an optimization problem, where this set is given as the solution of a Hamilton-Jacobi-Bellman (HJB) partial differential equation. This approach also accommodates non-linear systems, but on the other hand, the resulting HJB equation may be very difficult to solve in practice. There is, however,

software available for this type of problems, see [Mitchell and Templeton, 2005].

Another branch of the theory deals with the problem of *anti-windup*. In this setting, a local *performance controller* is designed without taking the saturation non-linearity into account. The problem is then to find an anti-windup modification of the controller that leaves the behavior of the local controller unaffected when there is no saturation, and limits the effects of saturation if it occurs, see for example [Rönnbäck, 1993]. In [Teel and Kapoor, 1997], the problem was given a rigorous definition and solved for the case of stable linear systems. In [Teel, 1999] the anti-windup problem for exponentially unstable linear systems is addressed.

The notion of reachability sets will be used in Chapter 10, where analytical expressions for the reachability set of a planar inverted pendulum system are derived.

1.6 Organization of the Thesis

The thesis consists of five parts. Part I, Background, contains references to related work. In Chapter 2, an overview of the Modelica language is given, and in Chapter 3 results and methods in the field of optimal control are reviewed. In Part II, Languages and Software Tools, the work on the JModelica compiler and Optimica is presented. Chapter 4 contains motivations and objectives for the JModelica compiler, as well as benchmark results. Chapters 5 and 6 give details of the implementation of the JModelica compiler, and Chapter 7 presents the Optimica language extension and gives some examples of optimization problems formulated in Optimica. In Part III, Case Studies, two case studies are presented. In Chapter 8, the work on DryLib and related applications are presented, and in Chapter 9, the work on start-up optimization of a plate reactor is presented. Part IV, Control of Pendula, contains material related to manual control of inverted pendula. Chapter 10 presents theoretical results for a control structure enabling safe manual control of an inverted pendulum, and in Chapter 11, the experimental work on a pendulum on two wheels is presented. The thesis ends with Part V, Conclusions.

I

Background

I

2

Modelica

The Modelica language is widely used in industry for modeling and simulation of physical systems [The Modelica Association, 2007b]. Example application areas include industrial robotics, automotive applications and power plants. Typical for models of such systems is that they are of large scale, commonly up to 100.000 equations. There are also extensive standard libraries for e.g., mechanical, electrical, and thermal models.

The first version of Modelica, 1.0, (see [The Modelica Association, 1997]) was published in September 1997. The effort was targeted at creating a new general-purpose modeling language, applicable to a wide range of application domains. While several other modeling languages were available, many of those where domain-specific, which made simulation of complex multi-domain systems difficult. Based on experiences from designing other modeling languages, notably Dymola, [Elmqvist, 1978], and Omola, [Andersson, 1994], the fundamental concepts of *object-orientation* and *declarative programming* were adopted. The latest version of the Modelica specification, 3.0, see [The Modelica Association, 2007a], was released in September 2007. The work presented in this thesis, however, is based on the previous specification, version 2.2 [The Modelica Association, 2005], which was released in February 2005.

The Modelica language has evolved from the simulation community, with roots in analog simulation dating back to the 1940s. Several modeling formalisms have been proposed, including Bond Graphs, [Karnopp and Rosenberg, 1968], which explores the energy flow between systems, VHDL-AMS, [IEEE, 1997], which is used to model electronic hardware, and block diagram modeling, which is the dominating paradigm in Matlab/Simulink. For an overview of the evolution of the field of continuous time modeling and simulation, see [Åström *et al.*, 1998]. As a modern modeling language, Modelica has been inspired by several modeling paradigms, with the objective of offering a unified modeling environment.

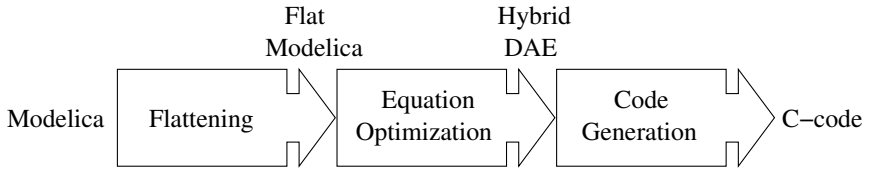


Figure 2.1 The Modelica translation process.

2.1 Simulation of Large Dynamic Systems

Mathematical modeling of complex physical systems requires appropriate high level languages. In particular, it is essential that a modeling language for such systems offers abstractions for *structuring* of models. A particularly successful paradigm has been that of object-oriented modeling, [Cellier, 1991]. In the Modelica context, the structural concepts of object-orientation, such as classes, components and inheritance, are emphasized, rather than dynamic creation of objects and message passing.

Modelica is designed with multi-domain modeling in mind. Accordingly, the language is particularly useful for applications which involve modeling of physical phenomena from different domains. For example, in automotive applications, it is desirable to have sub-models for the combustion, the mechanical systems, the electronics and the interaction with the road. In this type of applications, Modelica serves as a unifying language in which all these sub-systems can be modeled.

The primary objective of formulating a model of a complex physical system is most often *simulation*. That is, prediction of the response of a model, given a specified input stimuli, and the state of the model. If the model is sufficiently accurate, the simulation results may be used to draw conclusions about the behavior of the true physical system.

Now, numerical simulation of large dynamical systems requires sophisticated numerical algorithms. The interfaces of such algorithms are usually cumbersome, and does not correspond well to the engineering need for a high-level description language. Therefore, symbolic algorithms have been developed which transform a high-level Modelica description into executable code, [Elmqvist *et al.*, 2004]. In fact, it is fair to say that the success of Modelica relies on the availability of efficient computer tools.

```
class BouncingBall //A model of a bouncing ball
  parameter Real g = 9.81; //Acceleration due to gravity
  parameter Real e = 0.9; //Elasticity coefficient
  Real pos(start=1); //Position of the ball
  Real vel(start=0); //Velocity of the ball
equation
  der(pos) = vel;   // Newtons second law
  der(vel) = -g;
  when pos <= 0 then
    reinit(vel,-e*pre(vel)); // set velocity after bounce
  end when;
end BouncingBall;
```

Listing 2.1 A class modeling bouncing balls

2.2 The Modelica Translation Process

There is a fundamental difference between Modelica and traditional programming languages regarding the outcome of the compilation process. For many traditional programming languages such as Pascal or C, the ultimate objective of a compiler is to output a file containing executable machine code. Potentially, this involves compilation to an intermediate format. For Modelica, however, the objective of the compilation process is to output code (usually C-code) to be compiled and linked with a numerical simulation package. Therefore, concepts such as program counter, stack and heap are not applicable or relevant. Rather, a Modelica model consists of a set of *class instances*, often referred to as *components*. These components are allocated statically, in the sense that they are specified prior to simulation. Dynamic creation of new components during simulation is not allowed in Modelica.

The process of translating Modelica source code into a format suitable for numerical simulation/optimization algorithms can be divided into a number of steps, see Figure 2.1. In the first step, the Modelica code is flattened. This means that all component and inheritance structures are eliminated. The resulting model contains essentially a set of variables and a set of equations. The only property of the flat model that indicates its hierarchical origin is that a variable name is usually expressed as a qualified name, indicating the path of the corresponding variable. In the next step, the equations are sorted using graph-theoretical methods such as the BLT transformation, [Tarjan, 1972]. Sorting of equations are done in order to explore the structure of the model. The equations are then analyzed further and manipulated so that they can be more efficiently used with numerical software. The output of this step is referred to as a

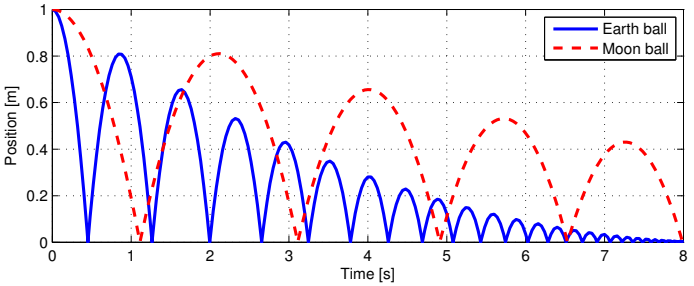


Figure 2.2 Position as a function of time for the `BouncingBall` example.

hybrid differential algebraic equation (DAE). The hybrid DAE also represents a generic mathematical description of the original Modelica model, and may be used for different purposes. The most common application is to generate C code, which is compiled and linked with an algorithm for numerical integration. The behavior of the system can then be simulated by executing the resulting application.

EXAMPLE 2.1
Consider the example in Listing 2.1. The model describes the dynamics of a bouncing ball. The free motion of the ball is governed by Newton's second law, and the actual bounce is modeled using a `when`-clause, in which the velocity of the ball is reversed. The behavior is parametrized by `g`, which is the acceleration due to gravity, and `e`, which is the elasticity coefficient.

Now, consider the following model which contains two components of the `BouncingBall` class in Listing 2.1:

```
class BBex
  BouncingBall earthBall;
  BouncingBall moonBall(g=1.62);
end BBex;
```

The component `moonBall` corresponds to a ball bouncing on the moon, and accordingly, the acceleration due to gravity is changed in a modification. In Figure 2.2 the positions of the two balls as a function of time are shown. □

2.3 The Modelica Language

Modelica is "a unified object-oriented language for physical systems modeling", [The Modelica Association, 2005]. As such, its most important fea-

tures are, [Fritzson, 2004]:

- Modelica supports equation-based acausal modeling, as opposed to assignment statements. Using equations, the modeler can state relations on their most natural form, without the need to solve for a particular variable. As a consequence, the data-flow direction is not determined *a priori*, but rather by the context of a particular component.
- Modelica can be used to express models from different domains, enabling modeling of heterogeneous systems.
- Modelica is an object-oriented language. This feature enables the modeler to use powerful structuring concepts such as classes, components, inheritance and generics.
- Modelica has strong support for component-based models, including means to connect components. This feature enables modelers to create modular models, as well as interfaces through which they can be connected.

These properties make Modelica particularly well suited for modeling of large and complex systems. For example, a well known limitation of block-based modeling is the need to solve for a particular set of variables. Even though the equations for each component in a composite model are simple and straight forward to derive, the modeler has to transform, usually by hand, the original model component equations into the standard ODE representation. For many physical systems, this transformation is often global, in the sense that all model components have to be considered simultaneously. In addition, the original structure of the model is often destroyed in the transformation. Modelica overcomes this difficulty by allowing acausal connection of model components. This approach leaves to the tool to transform the model equations into a format suitable for, for example, numerical integration.

The structuring constructs of Modelica, such as classes, inheritance, generics and packages promote model reuse and development of *model libraries*. This, in turn, enable domain experts to encapsulate knowledge in an accessible and structured way. As a consequence, the Modelica users can roughly be categorized into two groups. The first group contains the domain experts who implement libraries and who encode physical relations at the equation level. The second group contains users who primarily build custom simulation models based on existing libraries, but rarely deal with the specifics of the actual library implementation. In this way, Modelica provides a platform for a wide range of users and serves as a unifying language between users with different skills and across domain

borders. For a comprehensive description of the Modelica language and its usage, see [Fritzson, 2004].

Classes and Components

The class concept is fundamental in Modelica. Apart from the built-in classes "Real", "Integer", "Boolean" and "String", classes can also be defined by the user. A Modelica class may contain local class definitions, component declarations (these two entities are referred to as *elements*), equations and algorithms. A component declaration corresponds to an instance of a class, which can be either a user-defined class or a built-in class, such as Real. In the latter case, the component declaration is sometimes referred to as a variable declaration. The *variability* of a variable can be specified. For example, a variable of type Real can be specified to be a parameter, which means that it is constant during simulation. Elements can also be specified to be either public or protected. In the latter case, such elements are only accessible from within the class itself. In addition, Modelica supports multiple inheritance.

Consider Listing 2.2. The class `C` contains four elements: one local class definition, `A`, and three component declarations `z`, `v` and `a`. The behavior of the class *C* is defined by three equations (notice that the binding expression in the declaration of `v` is an equation), which relate its variables. The function `der` represents the time derivative operator $\frac{d}{dt}$, which is used to express dynamic behavior.

An important concept in Modelica is that of specialized classes[1]. Specialized classes can be used to indicate that a class has certain properties, e.g. in a **function** class, all public variables must have prefix either **input** or **output** and it may only contain *algorithms*, see Listing 2.3. A **record** on the other hand may neither contain equations nor algorithms. A **package** may only contain class definitions and constants and a **model** cannot appear in connection statements. Further, the specialized class **block**, is intended for models which have connections with fixed causality. Typically, control systems are implemented as blocks. Finally, the specialized class **connector** is used for connection interfaces, and may contain variables but no equations.

Equations and Algorithms

Equations and algorithms are used to define behavior. Physical phenomena are often modeled by mathematical equations. Equations in turn defines relationships between physical variables, such as pressure, temperature, current or voltage. In addition, many physical phenomena are

[1]The term *specialized class* was introduced in the specification of Modelica 3.0. Previously, the same concept was referred to as *restricted class*.

```
class C
  class A
    Real y[2];
  end A;
  Real z;
  Real v[2]={1,1};
  A a;
equation
  der(z)=-z;
  a.y=v;
end C;
```

Listing 2.2 A class.

```
function pow
  input Real x;
  input Integer p;
  output Real y;

algorithm
  y:=x;
  for i in 1:p-1 loop
    y:=y*x;
  end for;
end pow;
```

Listing 2.3 A function.

described by differential equations, where the variables as well as their derivatives with respect to time or space appear. Typical origins of equations are the laws of nature, e.g. the law of conservation of energy or Ohms law. By stating equations declaratively, the need to *solve* for a certain variable determined by the model context, and possibly simulation environment, is eliminated. Consider e.g., Ohms law, valid for an ideal resistor: $v = Ri$, where v is the difference in potential between the terminals, R is the resistance and i is the current. This equation can be stated in three different ways. Apart from the standard form, $i = v/R$ and $R = v/i$ also have the same mathematical meaning. This example illustrates what is a well known problem when formulating simulation models: it is often necessary to have several versions of the same model depending on how it is connected to its environment. Modelica solves this problem by enabling the user to state equations on their natural form and then leave it to the tool to transform the model into simulation code.

Most realistic systems exhibit discontinuous behavior. For example, the gear box of a car have a limited number of discrete gears. Also, equipment that is controlled by switching it on and off results in models with discrete behavior. Systems containing both continuous and discrete dynamics are referred to as *hybrid systems*. Since many engineering problems are hybrid in this sense, Modelica offers support for expressing hybrid models. There are two constructs available in Modelica for introducing hybrid behavior. `if`-clauses are used to express conditional equations, i.e., based on one or many conditions, a corresponding set of equations are *active* in the model, see Listing 2.4. It is also possible to express instantaneous events, i.e., *when* a specified condition evaluates to true, some actions should be taken. Examples of an action that is typically triggered by a `when`-clause are re-initialization of state variables, see Listing 2.1.

While physical phenomena are conveniently expressed by equations,

```
class PWL
  parameter Real a[4]={-3,-2,-1,0};
  parameter Real xlim[3]={-1,0,1};
  Real x;
equation
  if x<=xlim[1] then
    der(x)=a[1]*x;
  elseif x>xlim[1] and x<=xlim[2] then
    der(x)=a[2]*x;
  elseif x>xlim[2] and x<=xlim[3] then
    der(x)=a[3]*x;
  else
    der(x)=a[4]*x;
end PWL;
```

Listing 2.4 A class illustrating conditional equations.

there are other types of behavior that is expressed in a more natural way using *algorithms*. One such example is discrete-time control systems. Since it is often desirable to model not only the actual physical system, but also the associated control system, the ability to express algorithms in Modelica is important. Algorithms in Modelica can be used to express sequences of assignment statements, conditional statements and iteration. It is straight forward to map a Modelica algorithm to C code, and algorithms are therefore not explained in further detail here.

Connection of Components

Physical systems can often be decomposed into distinct subsystems, which are connected. By using a top-down approach, models for the subsystems can be combined to form more complex composite models. This methodology is strongly supported in Modelica. Specifically, the specialized class `connector` and the built-in function `connect` can be used to formulate structured models composed of interacting components.

A connector class serves as an interface between components. If two components both have a connector of the same type, they may be connected. The interface defined by a connector class consists of a set of variables, which are either of *potential* type or *flow* type. The semantics of a connection operation is the following: When a connection is formed, the potential variables of all connected connector components are set equal, while the sum of the flow variables is set to zero. During translation of a Modelica model, equations are generated from connection statements.

Consider Listing 2.5, where a number of classes describing a few basic electrical components are shown. The connector class `Pin` represents

```
connector Pin
  Real v;
  flow Real i;
end PositivePin;

partial model OnePort
  Real v;
  Real i;
  Pin p;
  Pin n;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end OnePort;

model Resistor
  extends OnePort;
  parameter Real R=1;
equation
  R*i = v;
end Resistor;
```

Listing 2.5 Modelica classes describing electrical components.

```
model ConstantVoltage
  extends OnePort;
  parameter Real V=1;
equation
  v = V;
end ConstantVoltage;

model Ground
  Pin p;
equation
  p.v = 0;
end Ground;

model Circuit
  ConstantVoltage cv;
  Ground g;
  Resistor r;
 equation
  connect(cv.p,r.p);
  connect(cv.n,g.p);
  connect(cv.n,r.n);
end Circuit;
```

Listing 2.6 Additional models defining electrical components.

an interface between electrical components. The potential variable `v` represents electrical potential and the flow variable `i` represents current. Using the `Pin` class, a partial (equivalent to abstract) model, `OnePort`, for a generic electrical component with two terminals can be formulated. This model is incomplete in the sense that it lacks an equation defining the behavior of the component. A defining equation is given in the specialized model `Resistor`, in this case Ohms law. It is straight forward to formulate models corresponding to, e.g., capacitors and inductors based on the partial model `OnePort`. Models corresponding to a voltage source and a ground element are also shown in Listing 2.6. Using these classes, it is straight forward to create a simple circuit, `Circuit` consisting of a voltage source, a resistor and a ground.

This example illustrates several important features of Modelica, such as hierarchical modeling, inheritance, and the connection mechanism.

Types

Modelica has a structural type system, [Abadi and Cardelli, 1996]. In essence, this means that a *class* does not define, uniquely, a *type*. Instead,

```
model B
  parameter Real s=-0.5;
  connector C
    flow Real p;
    Real q;
  end C;
protected
  Real x(start=1);
equation
  der(x)=s*x;
end B;
```

Listing 2.7 A Modelica class.

```
model classtype //Class type of B
  public parameter Real objtype s;
  public connector classtype
    flow Real objtype p;
    nonflow Real objtype q;
  end C;
  protected Real objtype x;

model objtype //Object type of B
  parameter Real objtype s;
end
```

Listing 2.8 Corresponding type interfaces.

the type of a class is either a built-in type, such as Real or Integer, or it is defined by all named elements (local classes and components) of the class. Based on this (informal) definition, relations such as subtype, supertype and type equivalence are defined. As a consequence, inheritance is not the only way to introduce a subtype-supertype relationship between classes, as in e.g. Java. Instead, two classes unrelated by inheritance may well be equivalent or have a subtype-supertype relation. This means that in Modelica, subclassing is not equivalent to subtyping.

The concept of types is not formally defined in the Modelica specification, [The Modelica Association, 2005][2]. In [Broman *et al.*, 2006], however, an interface for specifying Modelica types is proposed. In this work, the entities `classtype`, corresponding to the type of a class, and `objtype`, corresponding to the type of a component, are introduced. The type of an element is then defined by a set of `classtype` and `objtype` specifications. This type definition is recursive, in the sense that a `classtype` definition may contain other `classtype`'s as well as `objtype`'s. Notice that equations and algorithms are not part of the type definition, since they define the behavior and not the interface of a class. The use of `classtype` and `objtype` is illustrated in Listings 2.7 and 2.8. The `model` B contains one public Real parameter, `s`, one protected Real variable, `x`, one local class, `C` and one equation. The `classtype` interface of `B` contains elements corresponding to `s`, `x` and `C`, whereas the `objtype` interface contains only an element corresponding to the public declaration `s`.

[2] In the specification of Modelica 3.0, the concept of types has been developed and specified further.

```
model Capacitor
  extends OnePort;
  parameter Real C=1;
equation
  i=C*der(v);
end Capacitor;

model BaseCircuit
   replaceable Resistor e1 extends OnePort;
   Capacitor c(C=0.01);
end BaseCircuit;

model MyCircuit
   BaseCircuit circuit(c(C=0.001),redeclare Capacitor e1);
end MyCircuit;
```

Listing 2.9 An example of modifications and redeclarations.

Modifications and Parametrized Classes

Modelica supports changing of behavior of classes through *modifications*. Using the modification mechanism, it is possible to e.g., add or change binding expressions of variable declarations or replace local classes and components with more specialized elements. The latter feature enables the used of parametrized classes, commonly known as generics. This functionality can be used to apply a *top-down* approach to modeling. A course-grained model which contains simplified submodels can gradually be *refined* into a fine grained high-fidelity model. This concept is quite useful when designing prototype models, where it is initially important to gain insight into the qualitative behavior of a system.

Listing 2.9 shows the use of parametrized classes. The model `BaseCircuit` contains a resistor component, `e1`. The resistor is declared as replaceable, which is a requirement for parametrized classes and components. In addition, the declaration has a constraining clause; the keyword **extends** is here reused for another purpose than inheritance. A constraining clause states that the type of a replacing component must be a subtype of the type of the class given in the constraining clause. If there is no constraining clause, the type of a replacing component must be a subtype of the type of the default component declaration. In this case, the type of the default declaration is `Resistor`, but there is also a constraining type, `OnePort`, which has precedence.

In order to replace the replaceable component `e1`, a redeclaration modification is applied when a component of the class `BaseCircuit` is declared. For this purpose, the keyword `redeclare` is used. In the example, the class

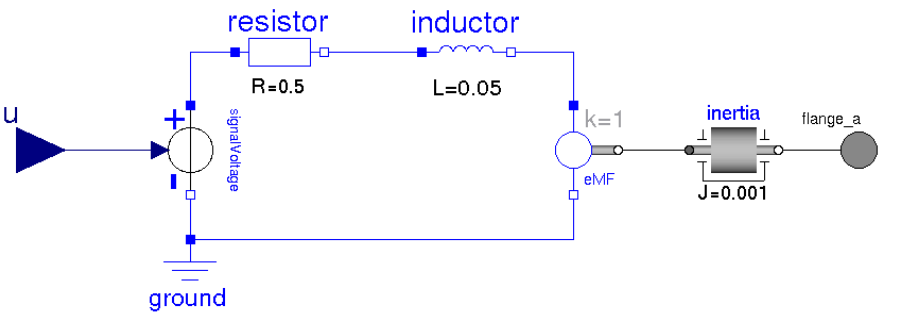


Figure 2.3 A DC motor model.

`Capacitor` is a subtype of `OnePort`, and therefore the redeclaration modification is valid. Listing 2.9 also illustrates modification of binding expressions and *merging* of modifiers, where outer modifiers overrides inner modifications. The default value of the capacitance in the model `Capacitor` is 1. However, this value is overridden in the component declaration of the capacitor `c` in the model `BaseCircuit`, using a modifier. This modifier, in turn, may well be overridden by modifications in a component declaration of `BaseCircuit`, as is the case in `MyCircuit`.

2.4 Graphical Annotations

Apart from the above mentioned features, Modelica supports the use of *annotations*, where complimentary information, for example tool specific information or graphics, can be given. The specification of annotations includes a standard for describing graphical objects, which may represent e.g., classes, components and connections. Typically, this feature is extensively used by library developers. By introducing a graphical notation corresponding to the classes of the library, non-expert users can use the library in a drag and drop manner, without the need to work directly with the actual code. Figure 2.3 shows an example of the graphical features of Modelica, illustrated by a simple model of a DC motor. The graphical annotations also have a textual representation, which is embedded in the model code. However, most tools offer a graphical modeling environment, and the graphical annotations are therefore seldom edited textually.

2.5 Modelica Tools

There are several software tools, commercial as well as free, offering support for the Modelica language. Although not all tools currently offer full Modelica support, the adoption of Modelica as a standard for dynamic models increases. Therefore, the level of support for Modelica, as well as the number of software tools supporting Modelica is also expected to increase. In this section, a brief overview of some tools supporting Modelica is given, for more information, see [The Modelica Association, 2007b].

Dymola

Dymola, developed by Dynasim, Sweden, [Dynasim AB, 2007], is a simulation environment offering full Modelica support. Dymola is considered the leading Modelica software, and is capable of translating and simulating very large models (>100.000 equations). Apart from graphical model editing, simulation and plotting, Dymola also offers support for model calibration, parameter optimization and hardware in the loop simulation (HILS).

OpenModelica

OpenModelica is an open source project managed by PELAB, University of Linköping, Sweden, [PELAB, 2007], and is released under the BSD license. The distribution contains a Modelica compiler which translates Modelica source code into C-code intended for simulation. OpenModelica is developed in the language MetaModelica, which is an extended subset of Modelica, specifically targeted at implementation of semantic specifications in compilers.

MathModelica System Designer Professional

MathModelica System Designer Professional (MSDP), developed by Mathcore, Sweden, [MathCore Engineering AB, 2007], is a commercial Modelica environment which is based on the OpenModelica compiler kernel. MSDP offers a graphical editor as well as a simulation and plotting environment. In addition, MSDP has a link to Mathematica, which enables further analysis and transformation of Modelica models.

Scilab/Scicos

Partial Modelica support is offered by Scilab, and the associated simulation environment Scicos, developed within INRIA, France, by the Scilab Consortium, [INRIA, 2007]. In Scicos it is possible to mix traditional causal block-style modeling with *implicit blocks*, containing Modelica code.

MosiLab

Mosilab is a Modelica software developed by Fraunhofer-Institut für Rechnerarchitektur und Softwaretechnik, Germany, [Nytsch-Geusen, 2007]. Apart from support for editing and simulation of Modelica models, Mosilab also offers an extension to also include support for statecharts. This extension is integrated with the Modelica language, and enables, for example, the *structure* of a model to change during simulation, which is generally not possible with standard Modelica.

SimulationX

The simulation software SimulationX, developed by ITI, Germany, [ITI GmbH, 2007], also offers support for Modelica. The module TypeDesigner supports definition of Modelica classes, which may then be used in the model creation module and for simulation.

3

Dynamic Optimization

During the last five decades, the theory of dynamic optimization has received much attention. In 1957, Bellman formulated the celebrated principle of optimality, and showed that *dynamic programming* was applicable to a broad range of applications, [Bellman, 1957]. Following this work, dynamic programming has been applied to various fields, notably inventory control, economics, statistics, and engineering. For a modern description, see [Bertsekas, 2000a; Bertsekas, 2000b]. Using dynamic programming, an optimal control law can be derived from the solution of a partial differential equation, the Hamilton-Jacobi-Bellman equation. However, although a very elegant theory, dynamic programming has proven difficult to apply to large optimal control problems. In particular, the presence of nonlinear dynamics and state or control variable constraints may lead to computationally intractable problems. However, recent initiatives have shown that it is possible to compute approximate solutions (with prespecified error bounds) of the dynamic programming problem, increasing the applicability of the technique to complex systems, see [Lincoln, 2003].

Another important contribution to the theory of optimal control is the *maximum principle*, which was presented by Pontryagin and co-workers in 1962, see [Pontryagin *et al.*, 1962]. Whereas dynamic programming provides a closed loop control law (the control law as a function of the system states), the maximum principle provides the necessary conditions for an open loop control law (the control law as a function of time) to be optimal[1]. The maximum principle is an extension of classical results in the area of calculus of variations, dating back to the 17th century. The necessary conditions for optimality, as specified by the maximum principle, constitute a two-point boundary value problem.

More recent advances in research on optimal control have resulted in a new family of methods – the direct methods. The direct methods have

[1]There are some exceptions, where the maximum principle yields a closed loop control law, for example in the case of linear systems and quadratic cost.

been successfully applied to large-scale systems, and they can also, in some formulations, accommodate state constraints. These properties are important in industrial applications. In contrast, both dynamic programming and the maximum principle are difficult to apply to large non-linear systems, in particular in the presence of state bounds.

This chapter is organized as follows. In Section 3.1, some results from the calculus of variations, and the maximum principle are reviewed. In Section 3.2, the two main direct methods, namely sequential and simultaneous methods, are described. Section 3.3 describes some important industrial applications of dynamic optimization. The chapter ends with a summary in Section 3.5.

3.1 Calculus of Variations and The Maximum Principle

Optimal control has its roots in the calculus of variations, a branch of mathematics concerned with solving optimization problems containing differential equations as constraints. The very first variational problem was formulated and solved by Isaac Newton in 1686. The problem was to find the minimum-drag nose shape in hypersonic flow, [Bryson and Ho, 1975, p. 52]. Another classical variational problem is the brachistochrone problem, formulated by John Bernoulli in 1696, where a bead is assumed to slide frictionless on a wire between two points in space, located in a constant gravity field. The brachistochrone problem is then to find the shape of the wire that minimizes the time it takes for the bead to slide from one point to the other.

The classical results of optimal control derived by means of calculus of variations will now be reviewed. Consider the simplified optimal control problem

$$\begin{aligned} &\min_u J = \min_u \left\{ \phi(x(t_f)) + \int_{t_0}^{t_f} L(x,u)\,dt \right\} \\ &\text{subject to} \\ &\qquad \dot{x} = f(x,u), \quad x(t_0) = x_0 \end{aligned} \tag{3.1}$$

where x and u are functions of time. Introducing the *Hamiltonian* $H(x,u,\lambda) = L(x,u) + \lambda^T f(x,u)$, the cost function can be augmented to

$$\begin{aligned} J &= \phi(x(t_f)) + \int_{t_0}^{t_f} L(x,u) + \lambda^T (f - \dot{x})\,dt \\ &= \phi(x(t_f)) - \left[\lambda^T x\right]_{t_0}^{t_f} + \int_{t_0}^{t_f} H + \dot{\lambda}^T x\,dt \end{aligned} \tag{3.2}$$

where the second equality is obtained by integrating by parts. The variables $\lambda(t)$ are the *adjoined* or *co-state* variables. The variation of J can now be written

$$\delta J = \left[\left(\frac{\partial \phi}{\partial x} - \lambda^T \right) \delta x \right]_{t=t_f} + \int_{t_0}^{t_f} \left[\left(\frac{\partial H}{\partial x} + \dot{\lambda}^T \right) \delta x + \frac{\partial H}{\partial u} \delta u \right] dt \quad (3.3)$$

Requiring that $\delta J = 0$ gives the necessary conditions for a control profile u to be optimal:

$$\begin{aligned} \dot{\lambda}^T &= -\frac{\partial H}{\partial x}, & \dot{x}^T &= \frac{\partial H}{\partial \lambda}, \quad \frac{\partial H}{\partial u} = 0 \\ \lambda(t_f)^T &= \left. \frac{\partial \phi}{\partial x} \right|_{t=t_f}, & x(t_0) &= x_0 \end{aligned} \quad (3.4)$$

These conditions form a *two-point boundary value problem* (TPBV), since the state variables are fixed at $t = t_0$, whereas λ is fixed at $t = t_f$. A sufficient condition for optimality is $\frac{\partial^2 H}{\partial u^2} > 0$.

In some cases, the TPBV problem can be solved analytically, in particular if the size of the dynamic system is moderate, or if it exhibits some special structure. For example, if the dynamics is linear and the cost function is quadratic, then the solution of the TPBV problem specified by the necessary conditions can be shown to be equivalent to solving a Riccati differential equation. In this specific case, the control variable u is given as a function of the state variables only, i.e., a closed loop control law is obtained.

The necessary conditions for optimality (3.4) were generalized by Pontryagin and co-workers in *the maximum principle*[2], [Pontryagin *et al.*, 1962]. The maximum principle is based on the observation that an optimal control profile u^* must fulfill the condition $H(x^*, u, \lambda^*) \geq H(x^*, u^*, \lambda^*)$, that is, the Hamiltonian is minimized over all feasible u. u^* and λ^* denote the optimal state and adjoint profiles. The necessary, and also sufficient, conditions for optimality as specified by the maximum principle are then

$$\begin{aligned} \dot{\lambda}^T &= -\frac{\partial H}{\partial x}, & \dot{x}^T &= \frac{\partial H}{\partial \lambda}, \quad u^* = \arg\min_{u \in U} H(x^*, u, \lambda^*) \\ \lambda(t_f)^T &= \left. \frac{\partial \phi}{\partial x} \right|_{t=t_f}, & x(t_0) &= x_0 \end{aligned} \quad (3.5)$$

The maximum principle can in some cases provide a solution to problems where the conditions derived by means of calculus of variations fail. One

[2]The maximum principle was given its name because in the formulation used by Pontryagin, a criteria was maximized, whereas in western literature, it is more common to minimize cost functions.

such example is when the equation(s) $\frac{\partial H}{\partial u} = 0$ does not provide any information about the optimal u. This is the case if the control variable occurs linearly in the dynamics and in the cost function. For example, in the case of minimum-time problems with linear dynamics and control bounds, a solution can be computed by means of the maximum principle, which results in a bang-bang control profile. For an example see [Bryson and Ho, 1975, p. 110].

There are several extensions of the necessary conditions for optimality, which allow for more general problem formulations, including minimum-time problems and problems with terminal and path constraints for the states. See for example [Stengel, 1994] for a comprehensive, and comprehensible, presentation.

Numerical Solution of Two-Point Boundary Value Problems

Numerical solution of TPBV problems is significantly more difficult than solving initial value problems. The reason for this is mainly that variables are constrained both at the beginning and at the end of the interval. Standard integration algorithms does not normally allow for this kind of general boundary value problems. Several other methods have been proposed, however.

Shooting methods Early attempts to solve TPBVs were based on algorithms referred to as *shooting algorithms*. By starting with an initial guess of $\lambda(t_0)$, and then integrate the systems for the state and adjoint variables forward in time, the terminal value of the adjoint variables, $\lambda(t_f)$ are obtained. If the values obtained by integration do not match those specified by the necessary conditions for optimality, new values $\lambda(t_0)$ are selected, and the procedure is repeated. One strategy for updating $\lambda(t_0)$ is then

$$\Delta\lambda(t_0) = \epsilon \left(\frac{\partial \phi}{\partial x}(t_f) - \lambda(t_f)^T \right) \left(\frac{\partial \lambda(t_f)}{\partial \lambda(t_0)} \right)^{-1}, \quad \epsilon \in [0, 1] \tag{3.6}$$

There are some problems associated with this method, however. Firstly, it is inherent from the definition of the differential equations governing the adjoint system, that if the system $\dot{x} = f(x, u)$ is stable, then the adjoint system is unstable. Secondly, it may be difficult to obtain an initial guess for $\lambda(t_0)$. Thirdly, computation of the quantity $\frac{\partial \lambda(t_f)}{\partial \lambda(t_0)}$ may be ill-conditioned. For these reasons, improved methods referred to as gradient methods were developed. For additional details see [Bryson and Ho, 1975].

Gradient methods In order to overcome the problem associated with the adjoint system being unstable, the adjoint variables can be integrated

backwards in time. Starting with an initial guess for the control variable $u(t)$, the system dynamics is integrated forward in time in order to obtain the state variable trajectories. The adjoint system is then integrated backwards in time. Notice that the state variable trajectories are usually needed in order to integrate the adjoint system, and therefore needs to be stored. Given the quantities obtained by the integration, the control profile may then be updated according to the steepest descent strategy:

$$\Delta u(t) = -\alpha(t)\frac{\partial H}{\partial u} = -\alpha(t)\left(\frac{\partial L}{\partial u} + \lambda^T \frac{\partial f}{\partial u}\right) \tag{3.7}$$

where $\alpha(t)$ is a scalar function which is adjusted to ensure that the Hamiltonian is decreased. The justification for this method is that the Hamiltonian should be minimized with respect to u, in order to fulfill the necessary conditions for optimality. The update scheme may be further improved, by implementing a generalized Newton scheme

$$\Delta u(t) = -W(t)\frac{\partial H}{\partial u}. \tag{3.8}$$

The choice $W(t) = \frac{\partial^2 H}{\partial u^2}$ gives the classical Newton method. For overviews of gradient methods, see for example [Stengel, 1994; Bryson and Ho, 1975].

Gradient methods improves convergence, as compared to the previously described shooting methods. Also, if second order gradients are computed, convergence can be further improved.

Collocation methods Another alternative for solving TPBV problems is *collocation methods*[3]. In this approach, the continuous time derivative operator is *transcribed* into a discretize time approximation. This operation results in an approximation of the original problem, where the differential equation is fulfilled at a finite number of points in time, referred to as collocation points. The TPBV problem is then transformed into a system of equations to solve. Collocation is also used as an element of simultaneous methods, which are described in Section 3.2. For an overview of collocation methods applied to TPBV problems, see [Ascher and Petzold, 1998].

3.2 Direct Methods

In the last two decades, a new family of methods have emerged, referred to as direct methods. Direct methods attempt to solve dynamic optimization problems by transcribing the original *infinite dimensional* dynamic

[3]The term collocation point refers to a point in time at which the differential constraint $\dot{x} = f(x,u)$ is (approximately) fulfilled.

problem into a *finite dimensional* static optimization problem. Thus, the necessary conditions for optimality derived in the previous section are not considered. The development of direct methods is motivated in [Betts, 2001], where three main arguments are given:

- Indirect methods, based on the conditions specified by the maximum principle, require the user to derive the quantities $\frac{\partial H}{\partial x}$, $\frac{\partial H}{\partial u}$ etc. This requires skilled users, and even then the derivation can be involved for complex models.
- An important element of many realistic optimal control problems is state path constraints. Although there exists extensions of the maximum principle to treat also such problems, these results are often difficult to apply in practice. A particular difficulty is that the activation sequence of the constraints must be known *a priori*.
- The adjoint variables, which play an important role when solving optimal control problems by means of an indirect method, are troublesome in two respects. Firstly, the adjoint variables are not physical quantities, which makes it difficult to obtain initial guesses. Secondly, the computation of the adjoint variables is often ill-conditioned, which may lead to numerical problems. The problem is further elaborated in [Bryson and Ho, 1975, p. 214].

There are two main branches within the family of direct methods. The *sequential methods* rely on state of the art numerical integrators, typically also capable of computing state sensitivities, and standard nonlinear programming (NLP) codes. The controls are then usually approximated by piece-wise polynomials, which render the controls to be parametrized by a finite number of parameters. These parameters are then optimized. *Simultaneous methods*, on the other hand, are based on *collocation*, and approximate *both* the state and the control variables by means of piecewise polynomials, see [Biegler *et al.*, 2002] for an overview. This strategy requires a fine-grained discretization of the states, in order to approximate the dynamic constraint with sufficient accuracy. Accordingly, the NLPs resulting from application of simultaneous methods are very large, but also sparse. In order to solve large-scale problems, the structure of the problem need to be explored.

In the following, the sequential and simultaneous methods will be described. The presentation will be based on the optimal control problem

$$\begin{aligned} &\min_{u(t)} \phi(x(t_f)) \\ &\text{subject to} \\ &\qquad \dot{x} = f(x,u), \quad x(0) = x_0 \end{aligned} \tag{3.9}$$

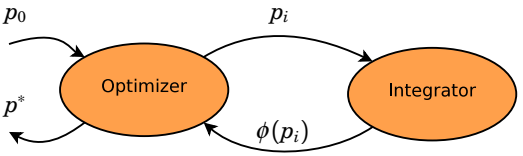


Figure 3.1 A sequential method is based on an integration algorithm and an NLP solver.

The Lagrange cost term is dropped here for the sake of brevity of the presentation. However, it is straightforward to extend the system with an additional state $\dot{x}_L = L(x,u)$, in order to incorporate the integral penalty term.

Sequential Methods

In a sequential method, the control variables are parametrized by a finite number of parameters, for example using a piece-wise polynomial approximation. Given fixed values of the parameters, the cost function of the optimization problem (3.9) can be evaluated simply by integrating the dynamic system. The parameters may then, in turn, be updated by an optimization algorithm, and the procedure is repeated, as illustrated in Figure 3.1. When the optimization algorithm terminates, the optimal parameter configuration is returned. Since the parameters determine the control profiles, which are then used to compute $x(t_f)$, the cost function can be written as $\phi(x(t_f)) = \phi(p)$. The infinite dimensional optimization problem is thus transformed into a finite dimensional approximation. For these reasons, sequential methods are also referred to as control parametrization methods. For a thorough description of single shooting algorithms, see [Vassiliadis, 1993].

Lagrange Polynomials A common choice of control parametrization is to use Lagrange polynomials. Lagrange polynomials are also commonly used in simultaneous methods, and will therefore be described here in some detail. The length of the control interval, ranging from t_0 to t_f is divided into N_e intervals, see Figure 3.2, and in each interval, the controls are written

$$u(t) = \sum_{j=1}^{N_c} u_{i,j} L_j^{(N_c)} \left(\frac{t - t_{i-1}}{h_i} \right) \quad t \in [t_{i-1}, t_i], \quad h_i = t_{i+1} - t_i \tag{3.10}$$

where $L_j^{(N_c)}$ are the Lagrange polynomials, t_i is the time at the beginning of interval i and h_i is the length of the interval. $u_{i,j}$ are the weights that

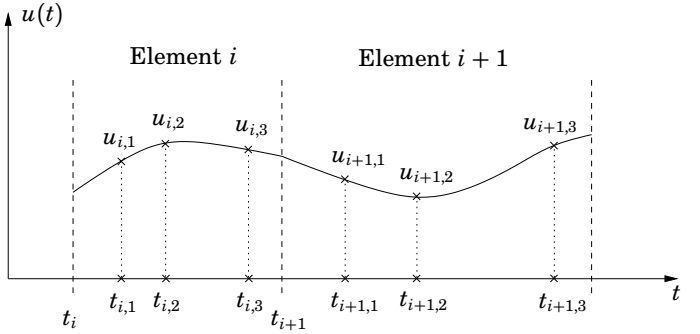


Figure 3.2 Piece-wise representation of control variables by means of Lagrange polynomials.

parametrize the control profiles. The Lagrange polynomials are given by

$$\begin{cases} L_j^{(N_c)}(\tau) = 1 & \text{if } N_c = 1 \\ L_j^{(N_c)}(\tau) = \prod_{k=1,k\neq j}^{N_c} \dfrac{\tau - \tau_k}{\tau_j - \tau_k} & \text{if } N_c \geq 2 \end{cases} \tag{3.11}$$

where the points $\tau_1 \ldots \tau_{N_c} \in [0,1]$ are used to define the polynomials. Notice that N_c points result in N_c polynomials of order $N_c - 1$. Lagrange polynomials are particularly attractive, since they possess the property

$$L_j^{(N_c)}(\tau_k) = \begin{cases} 1, & \text{if } \quad j = k \\ 0, & \text{if } \quad j \neq k \end{cases} \tag{3.12}$$

Accordingly, the values of $u(t)$ at the points $t_{i,j}$ are given by

$$u(t_{i,j}) = \sum_{k=1}^{N_c} u_{i,k} L_k^{(N_c)} \left(\frac{t_{i,j} - t_{i-1}}{h_i} \right) = \sum_{k=1}^{N_c} u_{i,k} L_j^{(N_c)}(\tau_k) = u_{i,j} \tag{3.13}$$

This property is useful since it enables path inequality constraints to be enforced, approximately, by simply introducing bounds for $u_{i,j}$ in the optimization. Some caution is required, however, since the values of the Lagrange polynomials in between the points are not considered.

Obtaining Gradients

Typically, the convergence of an optimization algorithm can be improved by providing it with gradients of the cost function with respect to the

parameters. While finite differences is a simple method for obtaining gradients, it is not well suited for in this particular application due to scaling problems and limited accuracy, [Rosen and Luus, 1991]. Taking the full derivative of the cost function $\phi(x(t_f)) = \phi(p)$, we obtain

$$\left.\frac{d\phi}{dp}\right|_{t_f} = \left.\frac{\partial\phi}{\partial x}^T \frac{\partial x}{\partial p}\right|_{t_f}. \tag{3.14}$$

While $\frac{\partial\phi}{\partial x}$ is usually straightforward to compute, the quantity $\frac{\partial x}{\partial p} = x_p(t)$, referred to as the *state sensitivity* with respect to the parameter p, needs attention. A common approach for computing state sensitivities is derived by differentiating the differential equation $\dot{x} = f(x,u)$ with respect to p:

$$\frac{d}{dp}\frac{dx}{dt} = \frac{d}{dp}f(x,u) \Rightarrow \frac{d}{dt}\left(\frac{\partial x}{\partial p}\right) = \frac{\partial f}{\partial x}\frac{\partial x}{\partial p} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial p} \tag{3.15}$$

which gives the *sensitivity equations*

$$\dot{x}_p(t) = \frac{\partial f}{\partial x}x_p(t) + \frac{\partial f}{\partial u}\frac{\partial u}{\partial p}. \tag{3.16}$$

If Lagrange polynomials are used to parametrize the controls, the parameters p correspond to the parameters $u_{i,j}$. Also, the quantity $\frac{\partial u}{\partial p}$ can be computed by differentiation of the corresponding Lagrange polynomials. This method for computing derivatives results in a new set of differential equations to solve. If the number of states of the system is n_x and the number of parameters is n_p, then $n_x \times n_p$ additional equations must be integrated. This operation is computationally expensive, although the efficiency of the integration can be increased by exploring the structure of the sensitivity equations. There is also software available which supports integration of the sensitivity equations, for example DASPK, [Maly and Petzold, 1996].

Another method for computing gradients is based on the adjoint equations. The gradient of the objective function can be computed by the following expression:

$$\frac{d\phi}{dp} = \int_0^{t_f} \lambda^T \frac{\partial f}{\partial u}\frac{\partial u}{\partial p}\,dt, \tag{3.17}$$

where the adjoined variables are given by

$$\dot{\lambda}^T = -\lambda^T \frac{\partial f}{\partial x}, \quad \lambda^T(t_f) = \frac{\partial\phi}{\partial x} \tag{3.18}$$

The strategy when using the adjoint method for computation of gradients is then to first integrate the system dynamics, forward in time, and

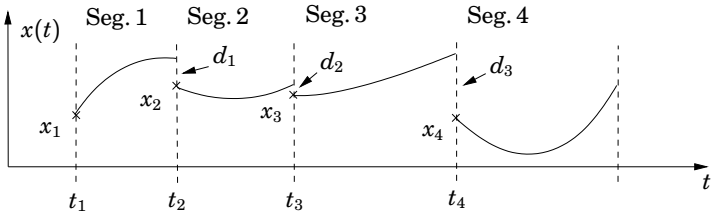


Figure 3.3 In a multiple shooting method, the control horizon is divided into a number of segments, which are integrated independently.

then integrate the adjoint system backwards in time. Notice that this requires the state trajectories resulting from the forward integration to be stored, since they are needed in the backwards integration. The advantage of using the adjoint method, as compared to integrating the sensitivity equations, is that fewer additional equations are introduced. Also for the adjoint method, efficient software is available, for example DASPK-ADJOINT, [Cao *et al.*, 2003; Li and Petzold, 2002].

Multiple shooting A popular extension of the single shooting algorithms is *multiple shooting*. In a multiple shooting algorithm, the optimization interval $[t_0, t_f]$ is divided into a number of *segments*, see Figure 3.3. New optimization variables corresponding to the initial conditions for the states in each segment, are then introduced. This enables the dynamics, as well as the sensitivity equations, to be integrated *independently* in each segment. In order to enforce continuity of the state profiles, equality constraints are introduced in the optimization problem which ensure that the *defects*, $d_i = x(t_{i+1}^+) - x(t_{i+1}^-)$ are equal to zero.

Multiple shooting addresses some of the problems associated with single shooting algorithms. For example, the computation of gradients may be very sensitive, in that small variations in the parameters may give rise to large changes in the cost function. This is true in particular if the integration interval is long. If the integration is performed over shorter intervals, this type of sensitivity is reduced, and the numerical stability properties of the algorithm are improved. Another advantage of multiple shooting algorithms is that state inequality constraints can be more easily accommodated. Since the initial states in each segment are optimization variables, algebraic inequality constraints can be enforced for the states variables at the segment junctions. Notice, again, that precaution is needed, since the values of the state variables in between the segment junctions are not considered.

Simultaneous Methods

A key element of a simultaneous method is the scheme used to discretize the differential equation. For this purpose, a collocation scheme is commonly used. To illustrate this concept, consider the optimal control problem (3.9). Introducing the forward Euler approximation of the continuous time derivative

$$\dot{x} \approx \frac{x_{k+1} - x_k}{h} \tag{3.19}$$

where $h = N_e/t_f$ is the interval length and N_e is the number of discretization intervals. The differential equation is then approximated by

$$x_{k+1} = x_k + h f(x_k, u_k). \tag{3.20}$$

Most commonly, this equation is then used iteratively to obtain an approximate solution of the differential equation. In the context of simultaneous methods, however, these equations are rather included in a static optimization problem as an equality constraint. The transcribed problem can then be written

$$\begin{aligned} &\min_{u_k} \phi(x_{N_e}), \quad k = 0 \ldots N_e - 1 \\ &\text{subject to} \\ &c(\bar{x}, \bar{u}) = \begin{pmatrix} x_0 + h f(x_0, u_0) - x_1 \\ \vdots \\ x_{N_e-1} + h f(x_{N_e-1}, u_{N_e-1}) - x_{N_e} \end{pmatrix} = 0 \end{aligned} \tag{3.21}$$

where the unknowns are $\bar{x} = (x_1,^T \ldots, x_{N_e}^T)^T$ and $\bar{u} = (u_0^T, \ldots, u_{N_e-1}^T)^T$.

Having transcribed the original continuous time problem (3.9) into a discrete time approximation (3.21), it is instructive to examine the connections between the first-order optimality conditions for the two problems. Following the presentation in [Betts, 2001], we introduce the Lagrangian $L = \phi(x_{N_e}) + \lambda^T c(\bar{x}, \bar{u})$. The optimality conditions for the problem (3.21) can then be written as

$$\begin{aligned} \frac{\partial H}{\partial x_i} &= \lambda_{i-1}^T - \lambda_i^T h \frac{\partial f(x_i, u_i)}{\partial x_i} - \lambda_i^T = 0, \quad i = 1..N_e - 1 \\ \frac{\partial H}{\partial x_{N_e}} &= \frac{\partial \phi(x_{N_e})}{\partial x_{N_e}} - \lambda_{N_e-1}^T = 0 \\ \frac{\partial H}{\partial u_i} &= \lambda_i^T h \frac{\partial f(x_i, u_i)}{\partial u_i} = 0, \quad i = 0..N_e - 1 \end{aligned} \tag{3.22}$$

which can be rearranged into

$$\begin{aligned} &\frac{\lambda_i^T - \lambda_{i-1}^T}{h} = -\lambda_i^T \frac{\partial f(x_i, u_i)}{\partial x_i}, \quad \lambda_{N_e-1}^T = \frac{\partial \phi(x_{N_e})}{\partial x_{N_e}} \\ &\lambda_i^T \frac{\partial f(x_i, u_i)}{\partial u_i} = 0 \end{aligned} \tag{3.23}$$

In the limit, when $h \to 0$, these relations reduce to the optimality conditions for the original problem (3.9). The convergence properties for a more realistic case, where a *monomial-basis* formulation (see [Bader and Ascher, 1987]) is used instead of the forward Euler approximation, is analyzed in [Kameswaran and Biegler, 2006].

In order to increase the accuracy of the approximation of the differential equation, the number of intervals, N_e can be increased. This, in turn, leads to very large NLPs to solve. However, NLPs resulting for collocation are typically highly structured, in that the Jacobian of the equality constraint $c(\bar{x}, \bar{u})$ is very sparse, see [Betts, 2001] for a detailed treatment. Exploring sparsity is therefore essential in order to efficiently solve this type of problems.

Collocation using Lagrange polynomials Whereas the forward Euler approximation is straightforward to implement, there are several integration schemes which have superior numerical properties. One such example is orthogonal collocation over finite elements with Radau points and Lagrange polynomials. It can be shown that such a collocation scheme is equivalent to a fully implicit Runge-Kutta scheme, see Appendix A. Accordingly, the stability properties of this scheme are inherited, see for example [Petzold, 1986].

As described above, the purpose of the transcription procedure is to translate the infinite dimensional dynamic constraint into a finite dimensional constraint, which can be incorporated into the final static non-linear program.

Consider, again, the differential equation

$$\dot{x} = f(x, u), \quad x(0) = x_0, \tag{3.24}$$

where x are the state variables and u are the control variables. The optimization horizon is divided into N_e finite elements, and within each element, N_c collocation points, $\tau_j \in [0, 1]$, are defined. The choice of Radau points implies that $\tau_{N_c} = 1$. Introducing the element lengths h_i, the time instants of the collocation points may be expressed as $t_{ij} = t_i + h_i \tau_j$, where t_i denotes the start time of element i. Introduce the polynomial state vari-

able approximations

$$x_{N_c+1}(t) = \sum_{k=0}^{N_c} x_{ik} L_k^{(N_c+1)} \left(\frac{t - t_{i-1}}{h_i} \right) \quad t \in [t_{i-1}, t_i], \tag{3.25}$$

where $L_k^{(N_c+1)}$ denotes Lagrange interpolation polynomials of order N_c and x_{ij} are parameters. In order to obtain $N_c + 1$ Lagrange polynomials, the point $\tau_0 = 0$ is added. The control variables are discretized using the approximation

$$u_{N_c}(t) = \sum_{k=1}^{N_c} u_{ik} L_k^{(N_c)} \left(\frac{t - t_{i-1}}{h_i} \right) \quad t \in [t_{i-1}, t_i], \tag{3.26}$$

where the Lagrange polynomials $L_k^{(N_c)}$ have been introduced, and u_{ij} are parameters. The collocation equations may now be written

$$\sum_{k=0}^{N_c} x_{ik} \dot{L}_k^{(N_c+1)}(\tau_j) = h_i f(x_{ij}, u_{ij}) \tag{3.27}$$

for all $i = 1..N_e$ and $j = 1..N_c$. In order to enforce continuity of the state variables between elements, the constraints

$$x_{i-1,N_c} = x_{i,0} \tag{3.28}$$

must be enforced. This is also the reason why the state variables are approximated by Lagrange polynomials with a higher order than the control variables. The variables x and u and the original dynamic constraint (3.24) are now replaced by the parameters x_{ij} and u_{ij} and the equality constraints (3.27) and (3.28) in the final transcribed algebraic non-linear program.

Another popular choice of collocation scheme is the monomial-basis representation, [Bader and Ascher, 1987]. Using this method, the derivatives, $\dot{x}$, are approximated by Lagrange polynomials of specified order, which implies that the state variable approximations are based on the integral of the interpolation polynomials. The monomial-basis representation corresponds precisely to a Runge-Kutta scheme. Notice that in the case when the states, x, rather than their derivatives, are approximated by Lagrange polynomials, some calculations are needed to show that this scheme is indeed equivalent to a Runge-Kutta method, see Appendix A.

Comparison Between Sequential and Simultaneous Methods

There seems to be no general agreement on whether the simultaneous methods or the sequential methods are preferable for dynamic optimization of large-scale optimization problems. Multiple-shooting, which can be viewed as an algorithm bridging the gap between the two families of direct methods[4], is quite popular in non-linear model predictive control (NMPC) applications. Also, the simultaneous methods are gaining increasing industrial use, see for example [Pettersson *et al.*, 2005].

In comparing single shooting and collocation methods, some advantages and disadvantages can be distinguished. Single shooting is appealing due to its conceptual simplicity, and because of the size of the resulting NLP, which is usually manageable. In each iteration in a single shooting algorithm, the dynamic constraint is fulfilled with high accuracy, which means that the optimization procedure can be terminated prematurely (for example in order to avoid lack of convergence). The resulting control profiles may be then be suboptimal, but may still be an improvement over the initial guess. On the down-side, single shooting methods are computationally expensive, in particular if the sensitivity equations are integrated. Further, unstable systems and state path constraints are difficult to manage with single shooting methods.

Simultaneous methods, on the other hand, handle state path constraints more easily, and are also better suited for unstable systems. In fact, the ability to efficiently handle state path constraints is one of the major benefits of the simultaneous methods. Further, with the availability of NLP solvers capable of exploiting the sparse structure of the constraint Jacobian and Hessian resulting from collocation, simultaneous methods have proven to be computationally efficient also for large-scale systems. On the other hand, first and second order derivatives, as well as sparsity information, might be necessary for convergence. Also, the simultaneous methods are sensitive to poorly scaled problems, in which case the resulting NLP might be ill-conditioned which can result in, slow, or even lost convergence.

Multiple shooting methods inherit properties from both single shooting and collocation methods. If a large number of shooting intervals are used, the numerical stability properties are improved. Also, path constraints can be enforced at interval junctions, which is an improvement compared to single shooting. However, the computational demand for multiple shooting algorithms is still large. If derivatives are computed by means of integration of the sensitivity equations, additional equations must be integrated in order to obtain derivatives with respect to the initial state variables in

[4] A collocation scheme can be viewed as an extreme case of multiple shooting, where one shooting segment corresponds to one step in the integration algorithm.

each shooting interval.

Both sequential and simultaneous methods rely on the availability of high accuracy derivatives of the functions involved in the formulation of the optimal control problem. For example, $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial u}$ are needed to integrate the sensitivity equations. The same quantities are needed to compute the constraint Jacobian resulting from a collocation scheme. While manual differentiation of complex functions is cumbersome and error-prone, there is efficient software available for the purpose. Two examples are ADOL-C, [Walther and Griewank, 2007] which compute high accuracy derivatives of functions encoded in C, and ADIFOR, [Hovland and Carle, 2007], which performs the same task for FORTRAN codes. The availability of such software greatly increases the applicability of the direct methods, since it relieves the user from the task of computing and encoding, manually, derivatives.

3.3 Applications

Dynamic optimization is becoming a standard design tool, which is applied industrially in a large number of applications. In this section, a few of the most important applications will be briefly described.

Model Predictive Control

The key feature that distinguishes model predictive control (MPC) from most other control strategies is the receding horizon principle. An MPC controller solves, at each sampling instant, a finite horizon optimal control problem. Only the first value of the resulting optimal control variable solution is then applied to the plant, and the rest of the solution is discarded. The same procedure is then repeated at each sampling instant, and the prediction horizon is shifted forward one step. Thereby the name receding horizon control.

Historically, there has been two major selling points for MPC; it works well for MIMO plants, and it takes constraints into account *explicitly*. Both these issues arise frequently in many practical applications, and must be dealt with in order for a control design to be successful. MPC has been particularly successful in the area of process control, which is also the field from where MPC originates. Traditionally, MPC has been mainly applied to plants with slow dynamics, where the computational delay is small compared to typical sampling intervals. However, recent reports of MPC applications include plants with fast dynamics, ranging from air plane control to engine control. For a review of industrial use of MPC, including a historical review of the evolution of MPC, see [Qin and Badgwell, 2003]

During the last decade, there has been significant research efforts to resolve the theoretical issues of MPC. Notably, the problem of formulating a stabilizing MPC scheme has received much attention. As a result, several techniques to ensure stability have been presented, see [Mayne *et al.*, 2000] for a review. The theory for MPC based on linear systems is well developed, and strong results ensuring robust stability exists, see [Maciejowski, 2002] for an overview. Also, the optimization problem resulting from linear MPC is a Linear Inequality Constrained Quadratic Programming (LICQP) problem, which is a convex optimization problem, and efficient solution algorithms exist. In particular, existence of a unique global minimum is guaranteed. For non-linear system, there exist MPC formulations that guarantee stability under mild conditions. However, the resulting optimization problem is, in general, non-convex and usually no guarantee of finding a global minimum exits. Non-linear MPC remains a very active field of research and recent results have shown that optimality is not a necessary condition for stability, see [Scokaert *et al.*, 1999].

State Estimation

Receding horizon estimation (see for example [Rao *et al.*, 2003]) can be viewed as the dual of model predictive control. In this case, however, the problem is to compute estimates of the current state variables, given past measurement data, and a model of the dynamic system that generated the data. Receding horizon estimation is particularly appealing in the presence of state and control constraints, since these elements do not naturally fit within the framework of, for example the extended Kalman filter. It is interesting to note that while model predictive control has received significantly more attention, both in research and in industry, the problem of designing a state estimator is often more challenging than designing the actual controller. Also in this respect, receding horizon estimation is appealing, since the resulting dynamic optimization problem can be solved using the same type of methods which have been developed for NMPC.

Off-line Optimization

In some cases, the computational complexity associated with large-scale plants prohibits on-line optimization schemes. In such cases, it may still be feasible to calculate, *off-line*, optimal transition trajectories. This strategy is applicable, for example, in the process industry, where a common operation is to perform *grade changes*, which transfer the state of a system from one operating point to another. The different operating points, or grades, may then correspond to different product qualities. Application of optimal trajectories which have been calculated off-line typically need to be supported by a feedback system, in order to compensate for disturbances and modeling errors. In Chapter 9, a case study dealing with

off-line computation of optimal start-up trajectories for a plate reactor, and their implementation using a mid-ranging feedback system, is presented.

Parameter Optimization

Component-based dynamic models constructed using first-principles often contain a large number of physical parameters. While some parameter values, for example densities of metals, can be looked up in tables, other parameter values, for example heat transfer coefficients, may be difficult to obtain accurate values for. As a result, the result of a simulation experiment may not match the output produced by the corresponding physical system. In order to calibrate a model to better match measurement data, an identification problem may be formulated, where the deviation between model and plant output is minimized. Parameters for which accurate values can not be obtained are then optimized.

3.4 Tools for Dynamic Optimization

There are several commercial and free software tools that support dynamic optimization. In this section, some of these tools are briefly described.

Dymola

Dymola, see [Dynasim AB, 2007], is primarily a tool of modeling and simulation of Modelica models. Since version 6.0, Dymola also supports dynamic optimization. Multi-objective and multi-case problems can be formulated and solved by means of a sequential method. The resulting nonlinear programs can be solved by means of, for example, an SQP algorithm or a pattern matching algorithm. Dymola also provides a module that is designed for calibration of Modelica models, given measurement data. The focus of the optimization capabilities of Dymola is primarily parameter optimization, although optimal control problems can also be solved, if the controls are parametrized by the user.

HQP

HQP (Huge Quadratic Programming), see [Franke, 2007], is a software package that contains a numerical solver for large-scale non-linear programs, and interfaces to, for example, model representations available in the S-function format and in Omuses [The Omuses Team, 2007]. Dynamic optimization problems can be solved by means of a multiple shooting algorithm. An application of HQP is reported in [Franke *et al.*, 2003]. HQP is released under the license GNU Library General Public License, Version 2.

gPROMS

gPROMS, see [Process Systems Enterprise, 2007], is a modeling, simulation and optimization software that is particularly targeting chemical engineering applications. The optimization module within gPROMS supports both a single shooting and a multiple shooting algorithm.

Jacobian

Jacobian, see [Numerica Technolgy, 2007], is a modeling, simulation and optimization environment that has evolved from research at MIT. In particular, Jacobian features sensitivity analysis and optimization of a class of hybrid DAEs.

GESOP

GESOP, [ASTOS Solutions GmbH, 2006], provides interfaces to model and optimization descriptions encoded in Fortran, C, or ADA, as well as a graphical user interface. The graphical user interface enables the user to specify, with a high level of detail, the properties of the mesh, as well as variable bounds. GESOP contains both single shooting and multiple shooting algorithms, as well as a collocation method, for solving dynamic optimization problems. GESOP also provides an interface to the dynamic optimization package SOCS [Boeing, 2007], which is a package for solution of large-scale dynamic optimization problems.

MUSCOD-II

MUSCOD-II, [Diehl *et al.*, 2001], is a software supporting dynamic optimization by means of a multiple shooting method. Dynamic models may be provided in C, Fortran, or as gPROMS models. MUSCOD-II also supports solution of multi-stage problems, where the dynamics of the system model is governed by different sets of differential-algebraic equations during the optimization interval.

AMPL and IPOPT

AMPL [Fourer *et al.*, 2003] is a language, as well as a tool, for formulation of *static* optimization problems. The AMPL language offers constructs for encoding of cost functions and constraints, and also provides a convenient separation between the actual optimization problem and data.

AMPL, by itself, does not offer numerical algorithms for solving optimization problems. Rather, AMPL provides a generic interface that offers evaluation of the cost function and the constraints. In addition, the AMPL solver interface offers evaluation of the first and second order derivatives of the cost function and the constraints as well as sparsity patterns of the constraint Jacobian and Hessian. The first and second order derivatives

are computed internally within AMPL by means of automatic differentiation, which is a fast and accurate method for obtaining derivatives. A large number of optimization algorithms have been interfaced with AMPL. In particular, the solver IPOPT has been used to solve the large-scale optimization problems resulting from the automatic transcription procedure in the Optimica compiler, see Chapter 7. IPOPT is an interior-point barrier function algorithm, see [Wächter and Biegler, 2006] for details.

3.5 Summary

Historically, optimal control originates from the calculus of variations, dating back to the 17th century. Significant contributions were given by, amongst others, Newton and Bernoulli. The field was revitalized in the 1950s and 1960s. Two important contributions were then given by Bellman (dynamic programming) and Pontryagin, (the maximum principle). During the last two decades, a new family of numerical methods for dynamic optimization have emerged, referred to as direct methods. This development has been driven by the industrial need to solve large-scale optimization problems and it has also been supported by the rapidly increasing computational power of modern computers. Two important classes of direct methods are the sequential methods, which rely on state of the art numerical integrators and standard NLP codes, and the simultaneous methods which rely on collocation methods and specialized NLP solvers capable of exploring sparsity.

II

Languages and Software Tools

II

4

JModelica – A Modelica Compiler

4.1 Introduction

Current state-of-the-art software for simulation of Modelica models is very efficient for its main purpose—simulation. Certainly, simulation is the single most important usage[1] of Modelica models today, and will most likely be so, at least in the near future. However, as the body of Modelica libraries, commercial as well as public, grows, they also represent an increasing value in terms of expert knowledge being encoded in Modelica models. In order to increase the return of investment it is then of primary interest to enable flexible usage of models, in contexts other than simulation. There are several emerging usages of Modelica models, including:

Model restrictions. Modelica allows for very complex model behavior. In the general case, a DAE resulting from compilation of a Modelica model is non-linear and hybrid, and contains algorithms and function calls which may invoke execution of external C or Fortran code. On the other hand, many design algorithms, as will be described in the following, impose restrictions on model structure. For example, for linear systems, many strong results as well as efficient algorithms are available. Similarly for DAEs without hybrid elements. Since the choice of algorithms usually affects performance and accuracy, it is important to explore the structure of the model. It is

[1] The term *usage* will be used in the following to refer to ways of using a Modelica model, for example, for simulation or for optimization. The word *application*, on the other hand, will be used in this text to refer to the area of application of a model, such as electrical circuits, power trains or thermo-fluid systems.

generally the case that the efficiency of algorithms decreases as the complexity of the model structure increases. Restricting the choice of algorithms by always assuming the most general case in terms of model structure is therefore quite conservative. It could be argued that most Modelica models actually do use many of the facilities in Modelica which makes it difficult to explore model structure. This is true, but only partially. Models are most often developed with one or more particular usages in mind. If the single intended usage is simulation, there are few restrictions that need to be observed. On the other hand, if the intended usage of a model is optimization, it might be desirable to reduce the complexity of the model in order to enable use of more efficient algorithms and also restrict the use of elements which are known to be difficult to handle in the context of optimization. Assisting the user in choosing an appropriate algorithm, by offering means to classify the model structure, is therefore an important issue.

Model reduction. Modelica models are often quite complex, sometimes to a level where they are difficult to understand and analyze. A large number of parameters distributed over a large model may also result in models which are difficult to calibrate, simply because the effect of changing the value of a parameter is difficult to predict. One approach to overcome this difficulty is model reduction. The problem is then to produce a simpler model which approximates the original model. The simplified model may then be more suitable for analysis, while still capturing the important features of the original model.

Static and dynamic optimization. Optimization is becoming a standard tool to solve design problems in many engineering disciplines. A significant trend is that models of physical systems are used as constraints in optimization problems.

While there is a large body of work in the area of optimal control and estimation, it is not immediately applicable to general Modelica models, since many results and algorithms impose restrictions on the structure of models in order to be applicable. This observation stems simply from the fact that the class of models that can be efficiently simulated is much larger than the class of models for which efficient optimization algorithms have been devised. Never the less, a Modelica model may indeed have a structure which fits that of a particular algorithm, either because the model inherently has a particular property (for example, RLC network models, consisting of resistors, capacitors and inductors, are linear if the network component models are linear), or because the model has been adapted to fit

into a particular model class. As an example of the latter case, consider models containing interpolation tables. Linear interpolation of table data gives rise to models which are not continuously differentiable. If an algorithm which requires twice continuously differentiable models is to be used, it is a standard procedure to replace a linear interpolation scheme with, for example, a spline interpolation scheme, which fulfills the differentiability condition.

A distinguishing feature of many optimization algorithms is that they require additional information derived from the model, apart from the model itself. A standard requirement is gradient information, i.e., first, and in some cases, also second order derivative information. In order to manage large systems, some algorithms can also exploit sparsity information to further improve performance. For a discussion on dynamic optimization and related algorithms, see Chapter 3.

Control design. Models are often used in control design. The most common example is perhaps to derive a linearized model from a nonlinear system, and apply tools from the theory of linear control systems. Linearization of Modelica models is also often supported by current tools. However, more sophisticated control strategies are becoming more popular, which also increases the need for a flexible model API in order to explore the structure of models. One such example is Non-linear Model Predictive Control (NMPC) (see for example [Mayne *et al.*, 2000]. An NMPC controller solves, at each sampling instant, an optimal control problem based on predictions of the future model outputs. In order to solve the resulting optimization problem efficiently, the same requirements as for dynamic optimization applies.

Parameter estimation and calibration. Having constructed a model of a physical system, the problem of *calibrating* the model to the system often arises. Calibration of a model aims to reduce the mismatch between the model and the plant behavior, and is commonly performed by tuning model parameters until the simulated response of the model is close to that of the true system. However, this procedure can be quite tedious, especially if the model behavior is complex and the number of parameters to tune is large. It is therefore important to enable efficient calibration of Modelica models. This can be achieved, for example, by applying parameter optimization methods, or grey-box identification techniques [Bohlin and Isaksson, 2003; Gerdin *et al.*, 2007].

Embedded code generation. A current trend, that is particularly visi-

ble in the automotive industry, is to merge physical modeling, control system design and code generation targeted at embedded CPUs. The benefit from this approach is that the user is enabled to construct models and embedded control systems using high-level languages and tools, whereas the actual executable code is automatically generated from a high-level description. This area is quite challenging since the complex interplay between physical systems, control algorithms, and real-time control systems must be explored in order to predict the behavior of the composite system. As a consequence, in order to simulate, with a high level of detail, the behavior of the composite system, it is essential to model not only the physical plant and the control system, but also the real-time behavior of the embedded CPU, see for example [Henriksson *et al.*, 2003] for an overview. Another important usage in this area is hardware in the loop simulation (HILS), where the actual control system is evaluated against the model, which is then executing in a simulation environment.

Currently, it is often difficult to apply this wide range of available and emerging methods to models developed in Modelica. There are two main aspects of this problem that need attention.

Firstly, new usages call for new high-level language constructs. For example, in the context of optimization, there are several concepts that are not represented in the Modelica language, such as cost functions and constraints. This is not to be considered as a deficiency of Modelica, but rather a consequence of Modelica being intended for modeling of physical systems, not optimization problems. High-level language extensions are therefore important in order to provide, for the particular usage domain, suitable constructs possessing expressive power which is on par with that of Modelica. It is desirable that such extensions are done in a modular way, so that the actual *model* description is separated from, potentially several, *complementing* descriptions, relating to the model. The issue of language extensions will be further elaborated on in Chapter 7.

Secondly, algorithms for the usages listed above may require an extended model API, as compared to simulation. A typical simulation-based interface in a Modelica tool essentially offers evaluation of the right hand side of the DAE, evaluation of root-functions for detecting events, and possibly the Jacobian of the right hand side with respect to the states. This interface may, however be insufficient for other algorithms. For example, in model reduction, a common operation is to apply a coordinate transformation to the DAE and then truncate some states. Further, some optimization algorithms require the Jacobian of the right hand side also with respect to the model parameters and inputs. Second order derivatives and sparsity patterns may also be useful in order to improve performance

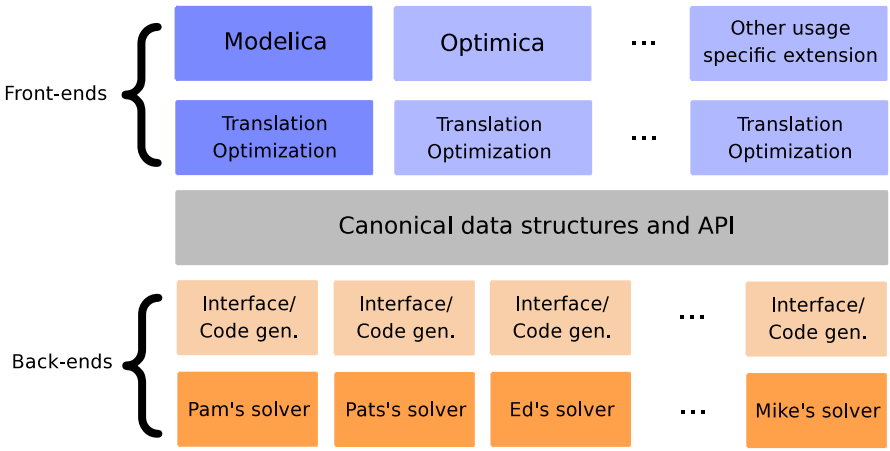


Figure 4.1 A software architecture enabling extensible compiler construction.

in terms of convergence of some optimization algorithms.

These two aspects have consequences for the design of extensible Modelica compiler software. In terms of front-ends, it is desirable to enable flexible implementation of front-ends, so that the core language, which is Modelica, can easily be extended. It is also desirable to offer a generic API, to which numerical or symbolic algorithms can be interfaced. Such interfacing algorithms, potentially complemented by supporting code generation modules, are referred to as back-ends. The division of a compiler into front-ends and back-ends is illustrated in Figure 4.1. This design is commonly used also in conventional compilers, in which case the front-ends may support different programming languages, and the back-ends may generate machine-code for different hardware architectures.

In the context of physical modeling and model-based design, the task of developing a framework according to the principle illustrated in Figure 4.1 is a major challenge. In particular, the problem of designing the intermediate representation layer, canonical data structures and API, requires attention. This involves mathematical definition of the class of models which can be expressed using the supported high-level languages. Such a definition exists, for example, for the Modelica language. The major

challenge lies, instead, in the specification of intermediate formats for potential extensions, and their relations to other extensions as well as the mathematical model description format.

As a fist step towards creating a flexible Modelica-based modeling environment that addresses the issues of language extensions and flexible model API, a new compiler, entitled JModelica, is under development. For this development the compiler construction tool JastAdd [Hedin and Magnusson, 2003; Ekman, 2006], is used. This tool uses several declarative features such as reference attribute grammars and rewriting, in order to support building extensible compilers.

In this chapter, an overview of the JModelica project and its current status will be given. In Section 4.2, the objectives of the JModelica project are stated. Section 4.3 motivates the choice of the development platform, JastAdd, and in Section 4.4, the main features of JastAdd are described. In Section 4.5, the role of a JastAdd description as an executable language specification is discussed. In Section 4.6, PicoModelica, which is a subset of Modelica, is introduced. PicoModelica will be used to explain design concepts in the JModelica compiler in Chapters 5 and 6. The current status of the JModelica project and some test results will be presented in Sections 4.7 and 4.8. The chapter ends with summary and conclusions in Section 4.10.

4.2 Objectives

The objective of the JModelica project is to develop a Modelica environment which facilitates experimental language design and language extensions, and provides a flexible model API in order to enable Modelica models to be used by a wide range of algorithms. In terms of software, this objective can be divided into two parts. Firstly, in order to accommodate extensible compiler design at the language level, the compiler *front-end* should be such that modular extensions of the core Modelica compiler is possible. Secondly, in order to enable flexible interfacing of different numerical algorithms, the compiler should offer a generic model API. Such an API may then be used to construct compiler *back-ends*, which interface with algorithms, for e.g., simulation, optimization or model reduction.

4.3 Development Platform—JastAdd

One of the main objectives of the JModelica project is to develop a Modelica compiler which is *modular* and *extensible*. The core of the software

is the JModelica compiler, which supports flattening of Modelica models. Flattening is the procedure of transforming a hierarchical Modelica model into a flat representation corresponding to a hybrid differential-algebraic equation (DAE) and will be treated in Chapter 6. It is desirable that extensions of the compiler, i.e., addition of new language constructs, can be made in a modular fashion such that the code of the core compiler remains intact. This approach is quite powerful, in that it allows for the core compiler supporting pure Modelica to be developed independently of, potentially several, core compiler extensions. The choice of JastAdd as implementation platform is therefore natural, since it is explicitly designed with modular extensible compiler construction in mind.

An interesting alternative to JastAdd is MetaModelica [Pop and Fritzson, 2006], which is used for implementation of the OpenModelica compiler. Like JastAdd, MetaModelica is a language designed for implementation of semantic behavior in compilers. MetaModelica is primarily based on concepts from functional programming, and uses pattern matching techniques to encode operations on the syntax tree. This paradigm differs from the object-oriented approach in that data structures and transformation functions are separated. A particularly interesting idea underlying the MetaModelica language is *meta-modeling*. In essence, meta-modeling enables modeling of *transformations* of models within the same framework as ordinary modeling. Fully implemented, meta-modeling would then allow for example for libraries to be developed which do not contain actual models but rather implement transformations of models.

At an early stage of the JModelica project, JastAdd was used to develop a front-end for a small subset of Modelica. The experiences from this early attempt were promising, given the level of functionality in the compiler that was obtained, and that the high-level constructs available in JastAdd enabled rapid development. These early experiences inspired further development based on the JastAdd platform. A particular target of JastAdd is *extensible* compiler construction, which matches the objectives of the JModelica project. In this respect JastAdd has proven to be a successful choice, supported by the fact that the Optimica extension presented in Chapter 7, has been implemented fully modularized. The JastAdd platform has also been used to develop a full-scale Java compiler (JastAddJ), [Ekman and Hedin, 2007]. The JastAddJ compiler is executing at reasonable speed (within a factor of 3 slower than some hand-coded compilers), while still being modularly extensible. The work on the JastAddJ compiler provides further support for the hypothesis that JastAdd is a feasible choice for the development of a Modelica compiler.

4.4 JastAdd

The JastAdd compiler construction system combines a number of features including object-orientation, inter-type declarations, reference attribute grammars, circular attributes, and context-dependent rewrites. These concepts will be presented in detailed in this section. The goal is to allow high-level executable specifications that support building extensible compilers. Current state-of-the art hand-coded compilers do not support the fine-grained extensibility that is needed for such extensions, but usually have support only for simple modularization into separate compiler phases, e.g., using the Visitor design pattern [Gamma *et al.*, 1995]. JastAdd is implemented in Java, generates Java code, and is available under an open-source license [Ekman *et al.*, 2006]. This section gives a summary of the main features of importance for the following chapters.

Lexing and Parsing with JastAdd

JastAdd is dependent on supporting software for transforming the source code into a stream of tokens (lexical analysis), and for parsing and subsequent construction of the AST. For these purposes, JFlex [Gerwin Klein, 2007] and Beaver [Beaver Project, 2007], respectively, have been used. JFlex is a lexical analyzer generator, which takes as its input a specification of tokens and produces a Java class which can be used to serve a parser with a stream of tokens. Beaver is an LALR(1)[2] parser generator which reads a syntactic language specification expressed as a context-free grammar, and produces a Java class containing methods for constructing an abstract syntax tree (AST) given a stream of tokens. The parser specification is given in terms of a *concrete syntax*, which describes the syntactic rules of the language and is in Beaver given on Extended Backus-Naur Form (EBNF). The *abstract syntax*, on the other hand, represents the *structure* of the program, disregarding the details of the syntax.

Abstract Syntax Trees

The core data structure in JastAdd is an abstract syntax tree. Representing computer programs as tree data structures is a standard technique, which is commonly used in compilers. Each node in the AST corresponds, roughly, to a language element in the source. When an AST has been constructed, it can then be used for analysis, for example finding declarations corresponding to identifiers, or for transformations. JastAdd uses an object-oriented representation of the AST, where each node is represented by a Java object. The child relations are implemented as object references.

[2]The acronym LALR(1) is short for *lookahead, left-to-right parse, leftmost-derivation, 1-symbol lookahead*, see for example [Appel, 2002]

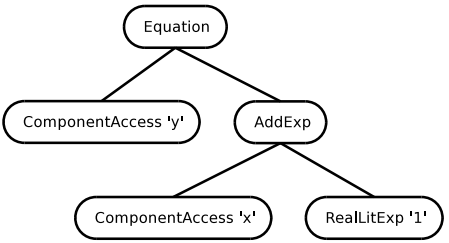


Figure 4.2 An AST representing the equation in Example 4.1.

EXAMPLE 4.1
Consider the Modelica equation

```
y=x+1
```

This equation would typically be represented by the AST depicted in Figure 4.2. Each node is labeled with the name of its corresponding type, and, in the case of a terminal, also a literal value. □

Object-oriented Abstract Grammar

The core of a language implementation is the specification of an abstract grammar. JastAdd uses an object-oriented abstract grammar specification, from which a Java class hierarchy, referred to as the source AST class hierarchy, is generated. The AST classes contain references to the parent and children nodes, as well as constructors and a traversal API. As an example of an abstract grammar specification, consider an excerpt from the PicoModelica (PicoModelica will be introduced in Section 4.6) grammar:

```
Root ::= ClassDecl* ...;
abstract ClassDecl ::= Name:IdDecl;
Model: ClassDecl ::= ExtendsClause*
                     ClassDecl*
                     ComponentDecl*
                     Equation*
                     /InstRoot/;
RealClass: ClassDecl;
ExtendsClause ::= Super:Access
                  [Modification];
IdDecl ::= <ID:String>;
```

In a JastAdd grammar specification, a *non-terminal* is declared by its name, followed by the symbol `::=`, after which the children of the non-terminal are listed. A specification of a non-terminal is called a *produc-*

tion. Nodes in the AST, as specified by the productions, are classified as being either non-terminals, in which case their names occur in the left-hand side of a production, or *terminals*, in which case they only occur in the right-hand side of productions. For example, `Root` is a non-terminal and `<ID:String>` is a terminal. Children of a node can be named, which is expressed by specifying the child name and the name of the node type separated by a colon. For example, `A ::= myB:B` means that the node type `A` has a child named `myB` of type `B`. Inheritance, in the object-oriented sense, between node types is expressed by adding a colon followed by the name of the supertype to the declaration. For example, `A : B ::=` indicates that the node type `A` inherits from `B`. If a node has multiple children of the same type, this can be indicated by an asterisk, and a child is marked as optional if it is surrounded by square brackets.

The root node of a PicoModelica AST has the type `Root`, and consists of zero or more `ClassDecl` nodes, as indicated by the asterisk. The non-terminal `ClassDecl` is abstract, which means that it cannot be instantiated. It has a child of type `IdDecl` which is entitled `Name`. The Java class corresponding to `ClassDecl` then has a constructor `ClassDecl(IdDecl Name)`, as well as setters and getters for its child node. `IdDecl` is a non-terminal, which contains a `String` literal named `ID`.

The non-terminal `Model`, corresponding to a `model` class declaration in PicoModelica, is composed of zero or more `ExtendsClause` nodes (representing superclasses), zero or more `ClassDecl` nodes (representing local classes), zero or more `ComponentDecl` nodes (corresponding to component declarations), and zero or more `Equation` nodes. In addition, `Model` is a subclass of `ClassDecl`. `Model` also has a child node, `InstRoot`, which is a *non-terminal attribute* (NTA), [Vogt *et al.*, 1989], which is indicated by the child name being surrounded by slashes, see the description below for details on NTAs. `RealClass` corresponds to the primitive type `Real`, and is also a subclass of `ClassDecl`. `ExtendsClause` corresponds to an extends clause in PicoModelica, and consists of an `Access` and an optional `Modification`, as indicated by the brackets.

Having defined the abstract grammar, JastAdd reads this specification, along with aspects containing semantic behavior, and finally generates a Java class hierarchy. The procedure of merging the content of aspects into the AST classes defined by the abstract grammar is usually referred to as weaving, and is illustrated in Figure 4.3.

Aspects with Inter-type Declarations

The abstract grammar specification results in a Java class hierarchy, where each class contains default constructors and a traversal API. A common approach to compiler construction is to either edit the generated classes directly, or to use some modularization scheme like the Visi-

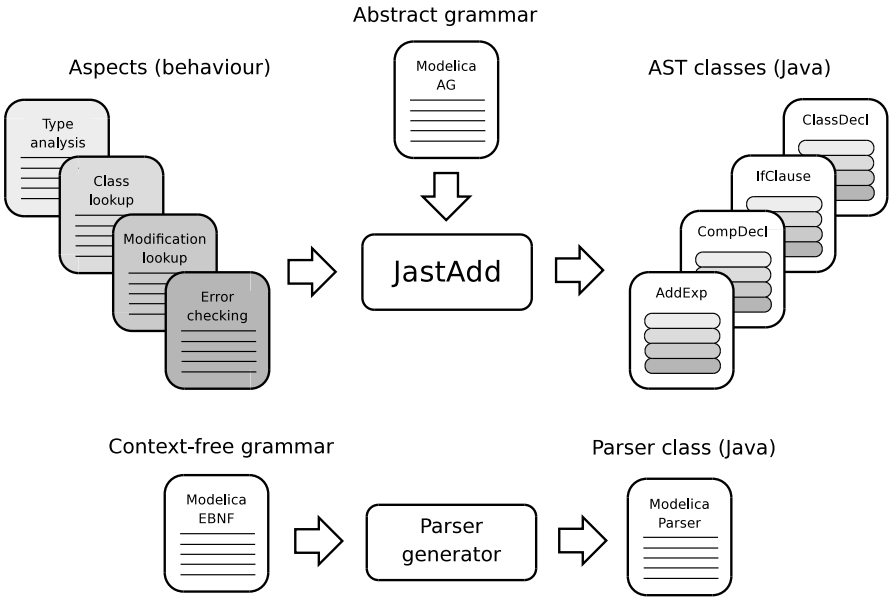


Figure 4.3 Schematic figure describing the functionality of JastAdd.

tor pattern. JastAdd, however, offers significantly increased flexibility, by supporting *inter-type declarations* which are organized into *aspects*. The term inter-type declaration refers to a declaration of, e.g., a method, a field or in the context of JastAdd, an attribute, outside the textual scope of a class. The use of aspects and inter-type declarations enable cross-cutting behavior to be modularized in a natural way. For example, the implementation of name analysis in a compiler typically involves addition of code to a large number of classes. In this case, the fundamental entity of object-orientated languages, the class, does not offer effective means for modularization. This deficiency can then be overcome by the introduction of aspect-orientation, see Figure 4.3. The inputs of the weaving procedure are the abstract grammar specification, which defines the AST classes, and the code defined in the aspects. The weaving procedure ensures that the semantic behavior, i.e., the inter-type declarations, are translated to Java and inserted into the correct classes. The result of the weaving is output as standard Java classes. These automatically generated classes are typically not edited further. Rather, changes are made in the aspects and the weaving procedure is repeated.

This approach to modularization is advantageous as compared to direct editing of automatically generated AST classes. Consider, for example, the

situation where the abstract syntax needs to be changed, which results in new AST classes being generated. In this case, manually inserted code potentially needs to be copied, again manually, into these classes. While the Visitor pattern enables modularization of behavior, it does not allow for new fields to be added to classes. In some situations, this feature is convenient, and the approach with inter-type declarations is therefore advantageous also in this respect.

Inter-type declarations in JastAdd use a syntax which is similar to that of AspectJ [Kiczales *et al.*, 2001]. Consider the example aspect `MyAspect`:

```
aspect MyAspect {
  // Add the boolean field visited to ASTNode
  boolean ASTNode.visited = false;

  // Add the method visit to ASTNode
  public void ASTNode.visit() {
    visited = true;
    for (int i=0;i<getNumChild();i++)
      getChild(i).visit();
  }
}
```

The first declaration adds a boolean field `visited` to the class `ASTNode` (`ASTNode` is the superclass of all AST classes generated by JastAdd). Next, the method `visit()` is introduced for the class `ASTNode`. This method uses the generic traversal API to retrieve all the children of the node and calls the `visit()` method for each of them.

Reference Attributed Grammars

The semantic behavior, corresponding to the rules of the language for which a compiler is developed, is primarily defined in JastAdd through *attributes*, whose values are defined by equations. An equation defines the value of one attribute in terms of the values of other attributes in the AST. Again, attributes and equations are inter-type declarations, which are defined in aspects. The equations used to define attributes are sometimes referred to as being *declarative*. This means that the order of execution is not determined a-priori, but rather that the equations are evaluated when needed as determined automatically by JastAdd. This meaning of declarative is different from the meaning of the term in the context of Modelica. In Modelica, an equation is declarative in the sense that it specifies an equality relation that *holds*, whereas in the JastAdd case the causality of the assignment defined by the equation is fixed. The attribute declarations are translated by JastAdd to ordinary Java methods, that are then inserted in the AST classes. Like in ordinary Knuth-style attribute grammars, see [Knuth, 1968], there are two types of attributes:

Synthesized attributes. In order to propagate information *up-wards* in the AST, synthesized attributes are used. Computation of a synthesized attribute can typically use information stored in the child nodes of the node for which the attribute is defined.

Inherited attributes. In order to propagate information *down-wards* in the AST, inherited attributes are used. The actual computation of the inherited[3] attribute value is performed in an *ancestor* node of the node for which the attribute is defined.

Apart from inherited and synthesized attributes, *equations* are key elements of AGs. Equations are used to define the actual value of attributes, and may also depend on evaluation of other attributes. For synthesized attributes, an equation can be specified in the attribute declaration, in which case it is called a default equation. Equations can also be specified for subclasses of the class for which the attribute is defined. In such cases the equation given in a subclass overrides a default equations. Equations for inherited attributes, on the other hand, are declared in ancestor nodes of the node for which the attribute is defined.

In contrast to Knuth-style AGs, attributes in JastAdd can be *reference-valued*. This means that an attribute may be a reference to another node, arbitrarily far away in the AST, and other data (attributes) can be accessed via the reference attribute [Hedin, 2000]. Typically, this is used for representing name bindings, e.g., from variable use to declaration, from class to superclass, etc. The attribute declarations and equations are specified using inter-type declarations in aspect modules, using the keywords `syn`, `inh`, and `eq`, which means synthesized attribute, inherited attribute and equation, respectively.

The use of synthesized and inherited attributes and equations will now be illustrated by means of some examples. The examples are extracted from the implementation of the name analysis framework in the PicoModelica compiler, see Chapter 5. In the end of this section, the evaluation of the example attributes will also be described. Consider the declarations:

```
① syn lazy ComponentDecl Access.myDecl() = null;
```

The declaration of a synthesized attribute is similar to a regular Java method declaration. However, some differences apply. Firstly, computation of synthesized attributes may not have side-effects. Secondly, synthesized attributes in JastAdd may be cached, if the keyword `lazy` is used. If an

[3]The term inherited used in the context of AGs is unrelated to inheritance in the object-oriented sense. The origin of the term is that in the original formulation of attribute grammars, inherited attributes are defined in direct parent nodes. Consequently, information propagated by attribute evaluations is *inherited* from parents to child nodes.

attribute is cached, its value is computed only once and then stored. Subsequent attribute evaluations then returns the stored value. The syntax for declaring a synthesized attribute is shown in the above listing, ①. The keywords `syn` and `lazy` indicate that the attribute is synthesized and that it should be cached. The return type of the attribute is `ComponentDecl`. The attribute is defined for the AST node `Access`, and has the name `myDecl`, which is expressed by the type name followed by a dot and then the attribute name. The `myDecl` attribute returns a reference to the AST node representing the component declaration corresponding the the identifier stored in the `Access` node. Finally, the attribute declaration has a default equation, which is optional, and defines the default value of the attribute. In this case, the default value is `null`. The default equation is valid for the node itself, but is also inherited (in the object-oriented sense) by its subclasses. Equations can also be overridden in analogy to Java methods. For example, consider the equations for the `myDecl` attribute:

```
syn lazy ComponentDecl Access.myDecl() = null;
② eq ComponentAccess.myDecl() = lookupDecl(getID());
③ eq Dot.myDecl() = getRight().myDecl();
```

which are valid for the classes `ComponentAccess`, ②, and `Dot`, ③. The syntax of equations for synthesized attributes is similar to that of attribute declarations, but uses the keyword `eq`. The return type is not specified for equations, since this is implicitly given by the attribute declaration. There are two syntactic alternatives for specifying equations. In the first alternative, which is shown in the example above, the value of the attribute is defined by a Java expression, preceded by the symbol `'='`. The second alternative is to define the attribute computation as a Java block, i.e., a number of Java statements enclosed by curly brackets. Such a method body must then contain a return statement which returns the value of the attribute. Notice that attribute computations may be dependent on other attributes, as in the case of the equation for `ComponentAccess`, ②, where the attribute `lookupDecl` is evaluated. Also, in the equation for `Dot`, ③, an attribute for a child node of `Dot`, `Right`, is evaluated. Accordingly, information is propagated upwards in the AST. Evaluation of attributes will also be illustrated in Example 4.2.

Let us now consider definition of inherited attributes:

```
④ inh ComponentDecl Access.lookupDecl(String name);
⑤ eq Model.getEquation().lookupDecl(String name) = memberDecl(name);
⑥ eq Root.getClassDecl().lookupDecl(String name) = null;
```

The syntax for declaring inherited attributes is similar to that of synthesized attributes. The only differences are that inherited attributes are declared using the keyword `inh`, and that they cannot have default equations.

Inherited attributes can also be cached. In the above example, the inherited attribute `lookupDecl(String name)` is declared for the type `Access`, ④. This attribute is *parametrized*, and the evaluation of the attribute may be dependent on the parameter. If a parametrized attribute is cached, then the evaluation result is stored for each parameter. The value of an inherited attribute is defined by an equation in an *ancestor* node. In the example above, this means that the value of the attribute `lookupDecl` is defined by ancestors of `Access`. In this case, two equations are declared for `Model`, ⑤, and `Root`, ⑥. If there is more than one ancestor with such an equation, it is the one closest to the `Access` that applies. An equation for an inherited attribute applies to a subtree of the node in which it is defined. Syntactically, this is indicated by specifying the child node that is the root of the corresponding subtree. In the example above, the equation declared in `Model` is then valid for the subtree which has an `Equation` node as root. This is indicated by the reference to `getEquation` in the equation declaration. In this particular case, `Model` delegates the computation of `lookupDecl` to the synthesized attribute `memberDecl(String name)`:

```
⑦ syn lazy ComponentDecl ClassDecl.memberDecl(String name) = null;
⑧ eq Model.memberDecl(String name)  {
  // Search all ComponentDecls and look for a matching name
  // Return matching declaration or else null.
}
```

The synthesized, parametrized, attribute `memberDecl(String name)` is declared for `ClassDecl`, with the default value `null`, ⑦. The default equation is overridden in the node type `Model`, by an equation definition, ⑧. If the `Model` contains a component declaration with a name matching the argument, then the declaration is returned. If not, `null` is returned.

Inherited and synthesized attributes are translated by JastAdd into Java methods located in their corresponding AST classes. To invoke evaluation of an attribute, the corresponding method is simply called. If the attribute depends on other attributes, such a call usually triggers a calling sequence where parts of the AST is traversed in order to locate nodes carrying equations applicable to the attribute which is being evaluated. For example, when `lookupDecl` is evaluated, the AST is searched upwards from the particular `Access`, until a node associated with a defining equation is encountered. For additional details on the evaluation framework for attributes in JastAdd, see [Ekman, 2006].

Evaluation of the attributes described above will now be described in the following example, where the calling sequence resulting from a particular evaluation is illustrated:

EXAMPLE 4.2
Consider the AST in Figure 4.4, which corresponds to a simple Modelica model:

```
model M
  Real x;
equation
  x=1;
end M;
```

We consider the case when the attribute `myDecl` is evaluated for the `ComponentAccess` (a) corresponding to the identifier `x` in the equation. `myDecl` is defined by equation ②, which accesses the inherited attribute `lookupDecl`. This results in evaluation of the attribute `lookupDecl`, declared in ④, with the parameter "x". The first ancestor of the `ComponentAccess`, which is of type `Equation` (b), does not have an equation defining the attribute `lookupDecl`. The next ancestor, however, is of type `Model` (c), which contains an equation defining the value of `lookupDecl`, ⑤. The attribute `memberDecl` is then evaluated, ⑤. The `memberDecl` attribute is defined by equation ⑧, and results in a search for a `ComponentDecl` with a name matching the argument. In this case, the `ComponentDecl` (d) matches, since its `IdDecl` has the name "x", and a reference to it is returned. This value is then returned to the `ComponentAccess` node (a). Since `myDecl` is a cached (`lazy`) attribute, its value (the reference to the `ComponentDecl`) is cached in the AST, (e).

Notice how synthesized attributes propagate information upwards in the AST, whereas inherited attributes propagate information downwards, as is indicated by the arrows. This example is intended to illustrate the evaluation of attributes. For a thorough discussion on name lookup, see Chapter 5. □

Using inherited and synthesized reference attributes, the original AST is decorated with node edges represented by reference attributes. This results in the AST being, conceptually and practically, a graph in which several new edges have been introduced. Such edges result from evaluation of reference attributes, and connect nodes related by semantic rules, although they may be located at very different locations in the AST. In addition, the result of an attribute evaluation can be cached. In Example 4.2, the reference attribute `myDecl` introduces a connection between a `ComponentAccess` and its corresponding `ComponentDecl`. This link is represented in Figure 4.4 by a dashed arrow (e). To summarize, inherited and synthesized reference attributes provides a convenient means to introduce connections between related nodes, although they might be located far apart in the AST.

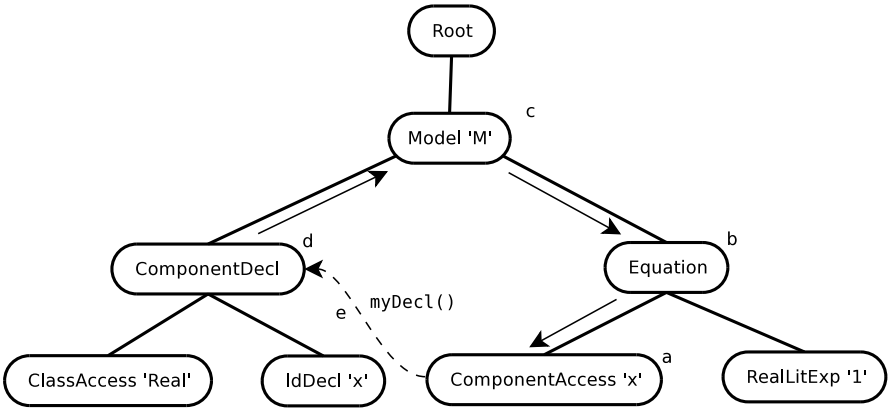


Figure 4.4 A simple AST illustrating how inherited attributes are evaluated during run-time.

Parametrized Attributes

An equation for an attribute is defined as a member of a class, and can be seen as having `this` as an implicit parameter, giving access to all other attributes in the same node (and possibly others via reference attributes or AST traversal). JastAdd supports also explicitly parametrized attributes, giving the equation access to more, possibly non-local, information. Typical uses include name lookup, as demonstrated in Example 4.2 and type comparisons.

Non-terminal Attributes

Non-terminal attributes, NTAs, [Vogt *et al.*, 1989], are typically used in cases when a child node cannot be constructed at parse time. Rather NTAs are defined by equations, and may be dependent on the structure of the AST, i.e., on the context. This mechanism is powerful, since it allows for the AST to be modified, and in particular for new nodes to be added, during the execution of the compiler, even after parsing. Consider the following example:

EXAMPLE 4.3
A common operation in a Modelica compiler is to *flatten* a model. As is described in Chapter 6, this procedure involves construction of a new data structure, the instance AST. The instance AST represents a particular model instance, and is used as an intermediate data structure, from which a flat model representation can be derived. It is convenient to insert the instance AST as a subtree of the corresponding `Model`. This is expressed

in the grammar specification:

```
Model: ClassDecl ::= ExtendsClause*
                     ClassDecl*
                     ComponentDecl*
                     Equation*
                     /InstRoot/;
```

where the child `InstRoot` is declared as an NTA. The actual computation of the subtree is defined by an equation:

```
syn lazy InstRoot Model.getInstRoot() = ...;
```

□

A more elaborate example would be the definition of Modelica library class declarations, stored on disk, as an NTA. In this case, when the NTA is accessed, the library files will be read, parsed and the corresponding AST will be constructed.

Rewrites

JastAdd supports conditional *rewrites* that can use attributes to define context-dependent modifications of the AST. Rewrites are typically used for modifying the AST from the initial form constructed by the parser, to a form more suitable for compilation [Ekman and Hedin, 2004]. A typical application of rewrites is when the node type cannot be determined at parse time. Consider the following example:

EXAMPLE 4.4
The *kind* of an access in a qualified name is usually not possible to determine at parse time. The term kind here refers to whether a particular access references a component declaration, or a class declaration. For example, if the name `A.B` occurs in an equation in Modelica, it is clear that the access `B` must correspond to a component declaration, whereas the meaning of `A` is ambiguous in the sense that it might be bound to either a class declaration or a component declaration. This ambiguity can be resolved by introducing the following rewrite:

```
rewrite AmbiguousAccess {
  when((lookupDecl(getID())!=null))
    to Access { return new ComponentAccess(getID()); }
  when((lookupClass(getID())!=null))
    to Access { return new ClassAccess(getID()); }
}
```

The rewrite is applicable to `AmbiguousAccess` nodes, and transforms such a node to a `ComponentAccess` or a `ClassAccess` depending on the conditions specified by the `when` clause. If no condition evaluates to true, then

no transformation is performed. The rewrite conditions are automatically checked whenever a node with associated rewrites are accessed. As a consequence, the order of execution does not have to be explicitly specified, but is rather determined automatically by JastAdd, based on the order in which nodes are accessed and attributes evaluated. □

Circular Attributes

JastAdd also supports circular attributes that are evaluated using fix-point iteration [Farrow, 1986; Magnusson and Hedin, 2003]. Circular dependencies of attributes arise when the evaluation of an attribute renders the same attribute to be evaluated twice for a particular node. If such situations occur, the JastAdd keyword `circular` can be used to invoke a fix-point iteration. A typical situation where circular attributes are useful is when to detect circular inheritance structures in object-oriented languages. In the case of Modelica, there are also examples of legal programs which contains circular dependencies. For example, import-statements and extends-statements are circular in the sense that it is valid to inherit from a class that becomes visible through an import class, and vice versa.

Combining Declarative and Imperative Code

While most of the compilation is best defined by the declarative attributes, there will usually be a need for generating some output based on the attributed AST. The attributes constitute an API that can be used by imperative code, i.e., ordinary Java methods. In some situations ordinary method declarations containing imperative code are more suitable than attributes, for example if a method has side-effects. In such cases, ordinary methods can simply be added using inter-type declarations.

Graphical Notation

In [Ekman and Hedin, 2007], a graphical notation for JastAdd code was introduced. Building on UML, [Object Management Group, 2007], a few additional elements have been added. Attribute declarations are displayed as class methods, carrying one of the prefixes `inh` or `syn`. Equations are shown using a similar syntax, with the prefix `eq`, but for brevity without the return type. In addition, the right hand side of equation declarations are shown in note boxes. For an example illustrating some of the attribute and equation declarations above, see Figure 4.5.

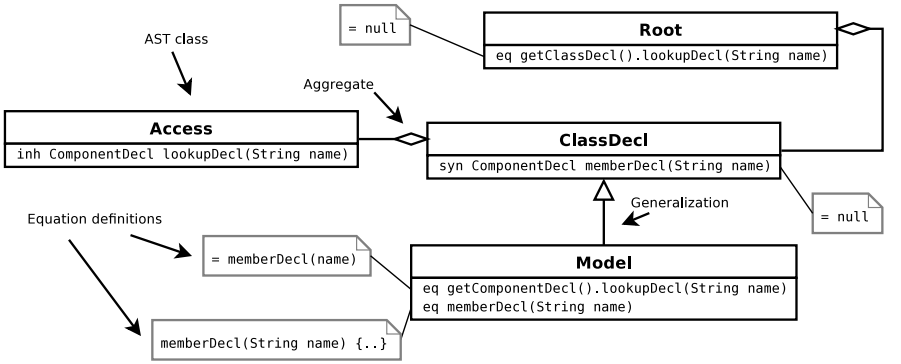


Figure 4.5 AST classes, attributes and equations in extended UML notation.

4.5 An Executable Specification

A JastAdd implementation of a compiler may be viewed as an executable language specification. The semantic behavior of the compiler is encoded as attribute declarations and equations, in combination with imperative code. Ideally, the attributes and equations of the JastAdd description are mapped to rules in the corresponding language specification. In line with this arguments, the JastAdd compiler implementation is then a formal and executable specification of the language.

As such, it is not intended, however, to replace a textual and possibly more informal specification text. A JastAdd description is a formal specification in the sense that a particular interpretation of the language specification has been encoded in an unambiguous manner, using JastAdd syntax. However, the mapping between the specification text of the language and the JastAdd description might not be obvious. In particular, the organization of the compiler implementation might not coincide with that of the specification text. Also, the specification text might not be declarative, as a JastAdd specification. The mapping between the textual specification and a JastAdd specification might then not be straightforward. In such cases, the procedure of translating a textual specification into a JastAdd equivalent involves many design choices, where the specification text has to be cast as declarative JastAdd constructions. Once it has been constructed, one of the main benefits of an executable JastAdd specification, in contrast to a traditional hand-coded compiler, is that it is significantly easier to analyze the semantic behavior of the compiler, due to its declarative nature.

The idea of creating executable specifications, in the context of Model-

ica, has been explored previously in [Kågedal and Fritzson, 1998], where the semantic description language RML was used to encode the semantics of the Modelica language. This implementation formed the basis of the OpenModelica project, [PELAB, 2007]. The success of the RML description of Modelica as an executable specification, seems to have been limited, however. As pointed out in [Broman and Fritzson, 2007], a likely reason for this is that the large size, and accordingly the level of detail, made it hard to get an overview of the specification. The same argument applies to JastAdd descriptions, since it is often necessary to study such a description to an extent that global understanding of the behavior is acquired. Another issue that deserves attention is consistency of specifications, if there is more than one. A specification written in a natural language is still needed, also in the presence of an executable specification. It is then important to clarify the relation between the specifications, and if possible, ensure that they are consistent.

Now, these objections are not to say that executable specifications expressed in, for example, JastAdd are not valuable. Certainly, attempts to formalize the semantic behavior of Modelica add to the understanding of the language, which, in turn, promote discussion and further development of the language itself. Semantic behavior of computer languages is characterized by a high level of abstraction. High-level description languages for describing semantics are therefore important, since they provide general concepts which make it easier to talk about semantics and to relate and distinguish different semantic options in the language design process.

To summarize, it is not clear that semantic description languages, such as JastAdd or RML, immediately provide means to formulate comprehensible formal and executable specifications. Rather, they should be seen as complements to textual specifications. While such descriptions may be unambiguous, they are also usually large and complex, much like the languages they describe. In order to improve this situation, further research is needed. In particular, the relationship between the a language specification text and the corresponding executable specification should explored, so that their interconnections are clear and explicit. For example, a JastAdd specification is more easily derived from a textual specification, if the latter explains the language declaratively.

4.6 PicoModelica

Since Modelica is a large and complex language, a compiler intended to support the full language is an equally complex program. Also, it is often desirable to introduce optimizations in the implementation which gives improved performance, but which might decrease the clarity of the design.

Design decisions, or design changes, made for a full compiler implementation also tend to involve much coding work simply due to the large code base. Therefore, a compiler for a subset of Modelica, hereafter referred to as PicoModelica (PM), has been introduced.

The purpose of PM is twofold. Firstly, it offers a convenient environment for testing and evaluating design principles, simply because the number of source code lines of the PM compiler is in the range of hundreds, while the number of lines of a full compiler implementation rather is counted in the range of ten thousands. Secondly, the compact nature of PM enables the design concepts to be presented without too many complicating details, which are due either to the need for managing all cases in the full language, or to performance optimizations. One of the objectives of this thesis is to present design concepts for a Modelica compiler implemented in JastAdd. This presentation, which is given in Chapters 5 and 6, is done in relation to PicoModelica, for the reasons of brevity and clarity. However, the same concepts are applicable, for the corresponding constructs, in a full-scale compiler.

Obviously, PicoModelica must include enough constructs of the full language to be a useful subject of study. On the other hand, the language should be small enough not to introduce too much complicating details. This trade-off is balanced by the inclusion of the following constructs:

- Class declarations
- Component declarations
- Inheritance
- Value modifications
- Replaceable components
- Simple equations

Certainly, several important constructs of the Modelica language are disregarded in PicoModelica, including dynamic name-lookup (inner/outer constructs), conditional components, arrays, and connect statements. Still, important concepts such as name and class lookup, type analysis, merging of modifications, management of parametrized classes and flattening may well be illustrated. The syntax, concrete as well as abstract, of PM can be found in Appendix B.

4.7 The JModelica Compiler

The design principles developed in the context of the PM compiler have been used to implement a prototype of a full-scale Modelica compiler, enti-

tled the JModelica compiler. The JModelica compiler is capable of parsing full Modelica 2.2, and supports flattening of a larger subset of Modelica than the PM compiler. Full support for parsing of the Modelica language is advantageous, even though not all constructs are supported by the flattening algorithm. For example, if a model contains constructs that are not currently supported by the flattening algorithm, corresponding error messages can easily be generated. Another advantage is that parsing of full Modelica enables use of a large number of classes in the Modelica standard library that contain only the supported constructs.

In addition to the capabilities of the PM compiler described in the previous section, the JModelica compiler supports flattening of the following constructs[4]:

- Short class declarations (for example `type PositiveReal = Real(min=0)`)
- Connect statements (including generation of connection equations)
- Standard library access
- Equations
- Built-in types and functions

Certainly, the JModelica compiler lacks essential functionality in order to be considered a full-scale Modelica compiler. For example, it lacks support for important language constructs such as functions, algorithms and dynamic name lookup (inner/outer declarations). Also, in order to flatten Modelica models, an evaluation framework for expressions and algorithms is needed. In the JModelica compiler, evaluation is currently supported only for scalar expressions. Support for arrays and type checking of arrays is also rudimentary. An important part of a Modelica compiler is the machinery for performing structural and symbolic computations. This includes Tarjan's algorithm and the BLT transformation, index reduction, tearing of equation systems and symbolic solution of equations when possible. These mechanism are also absent in the JModelica compiler.

Despite its limited functionality, however, the JModelica compiler, and in particular, the modular extension Optimica, have been successfully used to solve non-trivial dynamic optimization problems. Notably, Optimica (see Chapter 7), has been used to formulate and solve the start-up problem for a plate reactor, see Chapter 9. Also, the Optimica compiler has been used in two master's thesis projects dealing with lap-time optimization for racing cars and parameter optimization with application to vehicle models, respectively, see [Danielsson, 2007] and [Hultgren and Jonasson,

[4] As of October 2007.

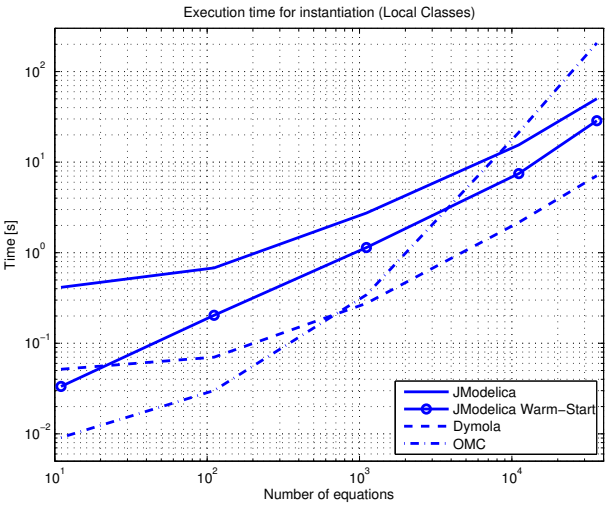


Figure 4.6 Benchmark results.

2007] for details. The Optimica compiler has also been used in the PhD course "Optimization-Based Methods and Tools in Control", that was given at the Department of Automatic Control, Lund University in September 2007.

4.8 Benchmarks

The correctness and performance of the flattening algorithm of the JModelica compiler have been evaluated and compared to two other Modelica tools: the OpenModelica 1.4.3 compiler and Dymola 6.0b. Three model structures have been constructed to be used for benchmarking. The first model structure consists of a network of electrical components, and the second model structure consists of hierarchically nested classes. The third model structure also consists of nested classes, but in addition, attributes of primitive variables are modified. Models with increasing complexity were automatically generated in order to compare the execution time for small as well as larger models. Correctness was verified by automatic comparison of the resulting flat descriptions from the three tools. The flat descriptions produced by the tools were equivalent in all cases. The execution times for the flattening procedure of the three tools are compared in Figures 4.6, 4.7 and 4.8. As can be seen, Dymola outperforms

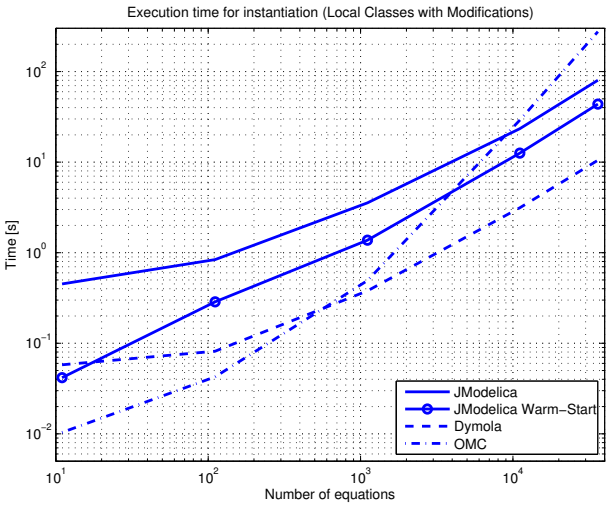


Figure 4.7 Benchmark results.

JModelica and OpenModelica in all cases. This should be no surprise, since Dymola is a highly optimized commercial product. For models of small and moderate size, the OpenModelica compiler is faster than JModelica. However, as model complexity increases, the JModelica compiler is faster than the OpenModelica compiler. Since it is well known that Java programs often execute slowly initially, execution times for the JModelica compiler were recorded also when the program was warm-started. That is, two models were parsed and flattened in sequence as a single Java-program. By measuring the execution time of the second flattening, the time associated with the start-up of the Java virtual machine and the initial dynamic Java compilation is not included in the result. As can be seen in Figures 4.6-4.8, the execution time of the JModelica compiler is then significantly decreased for the small models.

It should be noticed that these benchmarks only test a small number of language constructs. Accordingly, the results cannot immediately be used to draw conclusions about the overall performance of the tested tools. However, the results indicate that the JModelica compiler executes at a reasonable speed, when compared to other Modelica tools.

All benchmarks were performed on an Intel Core 2 Duo E6420 system, equipped with 4Gb of memory, running Fedora Core 7. The JModelica compiler was run on the Java HotSpot Server VM version 1.5.0 with a 2.5Gb sized heap.

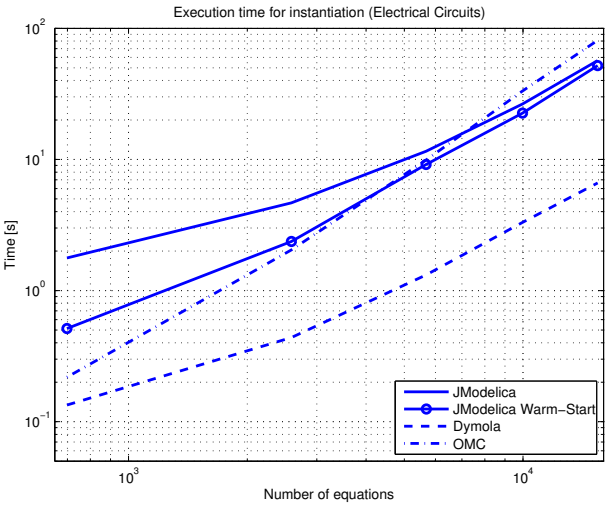


Figure 4.8 Benchmark results.

4.9 Extensibility of the JModelica Compiler

One of the primary objectives of the JModelica compiler is to develop a compiler that is easy to extend in a modular fashion. This approach enables rapid development of support for experimental language features without the need to modify the core compiler functionality, which is rather left intact. An additional benefit is that if the functionality of the original compiler is extended, this new functionality is immediately available also in a modular compiler extension. The extensibility feature will be illustrated in this section by means of a small example, where an abstract grammar specification encoded in the MetaModelica format is automatically converted into the JastAdd format for expressing abstract grammars. A larger example of the extensibility capabilities of the JModelica compiler will be given in Chapter 7, in the context of Optimica.

Abstract Grammar Conversion from MetaModelica to JastAdd

Compilers commonly rely on specifications of abstract grammars. This is the case with both MetaModelica and JastAdd. However, the format for specifying abstract grammars differ. MetaModelica uses a slightly extended version of Modelica, where some new keywords and syntactic constructs have been added. JastAdd, on the other hand, uses a format which corresponds closely to productions in a context free grammar. Consider

```
package PicoModelica

  uniontype ClassDecl

    record Model
     list<ExtendsClause> super;
     list<ClassDecl> classDecl;
     list<ComponentDecl> componentDecl;
     list<Equation> eq;
    end Model;

    record RealClass
    end RealClass;

  end ClassDecl;

  uniontype AbstractComponentDecl

    record ComponentDecl
      Option<Replaceable> repl;
      Option<Parameter> par;
      Access className;
      IdDecl name;
      Option<Modification> modification;
    end ComponentDecl;

  end ComponentDecl;

end PicoModelica;
```

Listing 4.1 A subset of the PicoModelica abstract grammar expressed in MetaModelica format.

the MetaModelica grammar specification in Listing 4.1, which contains a subset of the PicoModelica abstract grammar. The MetaModelica abstract grammar format is explained in [Pop and Fritzson, 2006]. In essence, the MetaModelica abstract grammar specification consists of standard Modelica language elements, such as classes and component declarations. Some new elements are introduced, however:

- A new specialized class, `uniontype`
- Lists of components. The syntax for specifying a list of components, named a, of the class `A` is `list<A> a`.
- Optional components. The syntax for an optional component named

a of the class `A` is `Option<A>` a.

Notice that the MetaModelica language contains several additional new constructs, but they are not needed to encode the PicoModelica abstract grammar, and are therefore not discussed here.

The modular extension of the JModelica compiler to also support the new language constructs needed to transform a MetaModelica abstract grammar into an equivalent JastAdd specification, consists of three parts. Firstly, the parser needs to be extended in order for the compiler to be able to read the new syntactic constructs. For this purpose, JastAdd offers a preprocessor for the parser generator Beaver. Using this feature, new productions can be added modularly. The result after the preprocessing step is one Beaver specification, which contains all grammar rules, both those from the original Modelica parser specification, and those from the extension. Secondly, new AST classes, corresponding to the new language elements need to be specified. Notice that the new AST classes may use, for example, inherit from, AST classes defined in the original Modelica abstract grammar. Thirdly, the semantic behavior of the new constructs needs to be specified. This can be done by introducing new aspects, in which new attributes, or equations for existing attributes, are added. These aspects are then included in the weaving procedure, so that the generated Java AST classes contain the behavior both from the original and the extended JastAdd specifications.

The resulting compiler now supports both the subset of standard Modelica that is supported by the original JModelica compiler, and the new constructs that were introduced in the modular extension. The result from the automatic conversion of the MetaModelica abstract grammar specification in Listing 4.1 is shown in Listing 4.2. It is worth noticing that the modular extension described in this section required only a moderate coding effort, primarily due to JastAdd's purposeful support for modularization.

4.10 Summary and Conclusions

In this chapter, an overview of the work on the JModelica extensible Modelica compiler has been given. The development platform JastAdd has been introduced, and some of JastAdd's main features have been highlighted. In addition, the role of a JastAdd description as an executable specification has been discussed. A small subset of Modelica, PicoModelica, has been introduced. PicoModelica will be used in the following two chapters to demonstrate how semantic behavior that is fundamental in a Modelica compiler, namely name and type analysis, and flattening, can

```
abstract ClassDecl;
Model : ClassDecl ::= super:ExtendsClause*
                      classDecl:ClassDecl*
                      componentDecl:ComponentDecl*
                      eq:Equation*;
RealClass : ClassDecl ::= ;

abstract AbstractComponentDecl;
ComponentDecl : AbstractComponentDecl ::= [repl:Replaceable]
                                          [par:Parameter]
                                          className:Access
                                          name:IdDecl
                                          [modification:Modification];
```

Listing 4.2 The abstract grammar shown in Figure 4.1 converted to the JastAdd format.

be implemented using JastAdd. The functionality of the current version of the JModelica compiler have been described, as well as its limitations. The modular extensibility capabilities of the JModelica compiler has been demonstrated by means of a small example, and benchmark results where the JModelica compiler has been compared to OpenModelica and Dymola have been presented.

Given the experiences from the development of the JModelica compiler, it is clear that JastAdd is a viable development platform for a Modelica compiler. An interesting observation can be made regarding the development time of the PicoModelica and JModelica compilers. As of October 2007, approximately 6 man-months have been spent on implementation. Given that the compilers support several non-trivial features of the Modelica language, this observation provides support for the claim that JastAdd is a development framework that enables rapid compiler development. During the development of the PicoModelica and JModelica compilers, it has also been evident that JastAdd specifications may be reusable. In particular, some design schemes that were developed for the JastAddJ compiler [Ekman and Hedin, 2007], have been possible to apply in the context of a Modelica compiler. This point will be elaborated on further in the next chapter.

5

Modelica Name and Type Analysis with JastAdd

Two areas which are fundamental in compilers are name and type analysis. Name analysis is concerned with binding names, or identifiers, to their corresponding declarations. Type analysis, on the other hand, deals with computation of types for expressions and computation of relations defined for types, such as subtype and supertype. Name and type analysis are important in order to perform checks of the validity of a program, but also in order to access information about declarations and types when implementing other features of the compiler, such as code generation.

Modelica is challenging both with respect to name and type analysis. While the basic name analysis framework resembles that of other object-oriented languages, Modelica also contains special constructs, such as modifications, which must be given special attention. As a consequence, identifiers in Modelica are looked up in context-dependent scopes, which are different from the normal lexical scope. In addition, Modelica has a structural type system, which differs from the type system of, for example, Java. As a consequence, a class declaration does not uniquely define a type, which complicates computation of relations such as subtype and supertype. Again, the presence of modifications must be accounted for in the type analysis, which further increases the complexity of this operation.

In the JastAdd framework, it is natural to represent name bindings as references between AST nodes. These references, in turn, may be implemented as declarative reference attributes, which are evaluated upon access.

In order to bind names to declarations, the kind of a name access must first be determined. In some situations, the meaning of a name is ambiguous, in the sense that it may refer either to a class or component declaration. This is the topic of Sections 5.1 and 5.2. Binding of names to declarations is then discussed in Section 5.3.

The results presented in this chapter builds on the work of Ekman and Hedin, specifically [Ekman and Hedin, 2006; Ekman, 2006; Ekman and Hedin, 2007]. The main contribution given in this chapter is the application and extension of their framework for name and type analysis of Java programs to Modelica. Initial work on this topic was presented in [Åkesson *et al.*, 2007].

5.1 Ambiguous Names

It is usually not possible to determine the meaning (or kind, see Example 4.4) of a Modelica name access at parse time. This is because the meaning of a name is highly context-dependent. For example, in some situations, it is not possible to determine, at parse time, if an entry of a qualified name refers to a class declaration or a component declaration. In addition to common object-oriented mechanisms such as qualified names and inheritance, there are also some very Modelica-specific language features such as *modifications*, that affect name analysis. Modifications are challenging, in particular since they render names to be looked up not in the enclosing environment, but rather in the scope of another class.

The Modelica specification states that a name is looked up in the ordered set of parents which lexically enclose an element. If a class is lexically contained in another class, the former precedes the latter in this set. An unnamed parent encloses all top-level class declarations. If a name cannot be found in a scope introduced by a parent in the set, corresponding to a class declaration, lookup proceeds in the next scope in the set. Some restrictions apply, however. A reference to a component declaration in another scope is only allowed if the declaration has the prefix `constant`.

Name ambiguities arise in qualified names, such as `A.B`. For example, if the qualified name `A.B` occurs in an equation, there are, at least, two possible meanings of this name. Either, both `A` and `B` refer to component declarations, or, `A` is a class and `B` is a component. Both alternatives are valid, but are not possible to distinguish without considering the *context* of the equation, i.e., what class declarations and component declarations are visible from the particular equation. The qualified name `A.B` is thus, in this case, ambiguous.

In some cases, the syntactic context of a name can be used to determine the meaning of the name. For example, in a component declaration clause, say `A.B b`, the name `A.B` must refer to a class declaration. In this case, the name `A.B` is not ambiguous since its meaning is clear from looking at the component declaration clause. However, the name itself, `A.B`, does not reveal its meaning, but must be considered in a context. This reasoning is based on the assumption that the Modelica program is correct.

Consequently, there must be a mechanism for checking the correctness of the classifications made. This, however, is considered to be part of the error-checking functionality of the compiler, and is not considered here.

From these examples, it can be concluded that there are two ways of determining the meaning of a Modelica name. In some cases, the syntactic context of the name can be used to classify the name. If this is not possible, the set of all visible class declarations and component declarations must be considered in order to determine the meaning of the name.

5.2 Classification of Names

In this section, a strategy for classification of Modelica names will be discussed. This strategy is adopted from [Ekman and Hedin, 2006], where it is applied to classification of names in Java.

Consider the following abstract grammar for names in PicoModelica:

```
abstract Access : Exp ::= <ID:String>;
ParseAccess : Access;
Dot : Access ::= Left:Access Right:Access;
ClassAccess : Access;
ComponentAccess : Access;
AmbiguousAccess : Access;
```

For simplicity, all `Access` nodes in the AST which are built by the parser are objects of the class `ParseAccess`. The `ParseAccess` nodes are then classified and transformed into nodes of the correct AST class using the *rewrite* mechanism in JastAdd. Apart from the `ParseAccess` class, the grammar contains the classes `ClassAccess`, `ComponentAccess` and `AmbiguousAccess`. In a correct Modelica program, there are no nodes of the `AmbiguousAccess` class when all nodes have been classified. Occurrences of such nodes indicates that there were names that could not be bound to declarations. Qualified names are modeled using the AST node class `Dot`, which in turn has two children: `Left` and `Right`. The parser ensures that AST subtrees corresponding to qualified names are right skew-symmetric, that is, only the `Right` child of a `Dot` node can be a node of the `Dot` class. The convention of how to build subtrees corresponding to qualified names is mentioned here, since it has consequences for how attributes in the following presentation are defined.

The classification framework for Modelica names proceeds in two steps, as noted above, and will now be described.

Syntactic Classification

Some names may be classified simply by considering their immediate syntactic context. Such classification is computationally inexpensive, since it

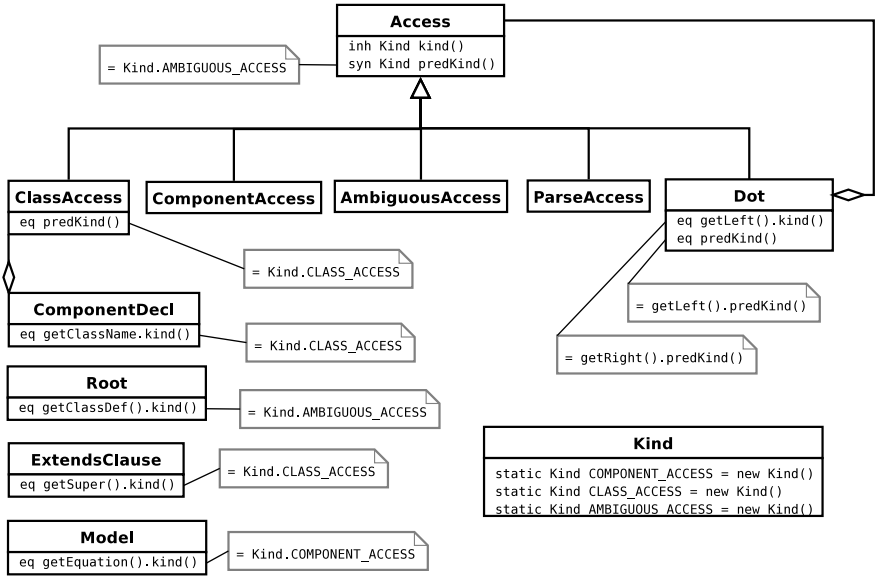


Figure 5.1 The framework for syntactic classification of names.

does not require a lookup of the name in order to check for visible declarations. Accordingly, it is desirable to use syntactic classification for as many names as possible. Following the presentation in [Ekman and Hedin, 2006], the inherited attribute `Access.kind()` is introduced. Notice that an inherited attribute is appropriate in this situation, since information about the syntactic context of a name needs to be propagated *downwards* in the AST. In PicoModelica, there are three kinds of syntactically classified names: class names, component names and ambiguous names. The purpose of the syntactic classification framework is to transform the unclassified `ParseAccess` nodes into nodes corresponding to either of these three kinds. The attribute `kind` is defined by equations in AST classes where the kind of a name can be determined without ambiguity, see Figure 5.1. For example, in the `ComponentDecl` class, it is clear that the child `ClassName` must be a class name, which in turn is encoded in a corresponding equation.

It is also possible to determine, in some situations, the kind of a name which is part of a qualified name. In Modelica, it is illegal to reference classes through component instances. This means that if a particular name in a qualified name has been classified as a class name, then all preceding names in the qualified name must also be class names. For

```
rewrite ParseAccess {
  to Access {
     if (kind()==Kind.COMPONENT_ACCESS)
        return new ComponentAccess(getID());
     else if (kind()==Kind.CLASS_ACCESS)
        return new ClassAccess(getID());
     else
        return new AmbiguousAccess(getID());
  }
}
```

Listing 5.1 The rewrite of ParseAccess nodes is based on syntactic classification given by the attribute kind.

example, consider the qualified name A.B.C. If C has been classified as a class name, then it follows that A and B must also be class names. This rule is encoded by the synthesized attribute predKind, see Figure 5.1.

It remains to define a rewrite which transforms ParseAccess nodes into their classified equivalent. However, given the attribute kind, this is straight forward, see Listing 5.1.

The syntactic name classification scheme will now be demonstrated by the following example:

EXAMPLE 5.1
Consider the following Modelica model:

```
model C
  model M
    Real x;
  end M;
  M m;
equation
  m.x=1;
end C;
```

Partial ASTs corresponding to this model are shown in Figure 5.2. The complete AST for the model is found in Appendix C. The execution trace following from accessing the ParseAccess 'm' node will now be described.

a) The node ParseAccess 'm' is accessed, which triggers the rewrite in Listing 5.1. This renders the inherited attribute kind() to be evaluated.

b) The first encountered node upwards in the AST is of type Dot, which has the associated equation:

```
eq Dot.getLeft().kind() = getRight().predKind()
```

c) The attribute predKind() is evaluated for the Right child of Dot.

d) When the node ParseAccess 'x' is accessed, the rewrite in Listing 5.1 is triggered. This renders the attribute kind() to be evaluated.

e) The call resulting from evaluation of kind() propagates upwards in the AST. The first encountered node is Dot, but since this class has no equation defining kind() for its Right child, the call is propagated upwards in the AST.

f) The node Equation does not define kind() either, and the call propagates further upwards.

g) For Model, the following equation is defined:

```
eq Model.getEquation().kind() = Kind.COMPONENT_ACCESS
```

and accordingly, the attribute kind() is evaluated to Kind.COMPONENT_ACCESS.

h) The node ParseAccess 'x' is rewritten to ComponentAccess 'x'

i) The attribute predKind() can now be evaluated using the equation:

```
eq ComponentAccess.predKind() = Kind.AMBIGUOUS_ACCESS
```

Notice that this equation is defined in the AST class Access, and is inherited to ComponentAccess.

j) Finally, ParseAccess 'm' is rewritten to AmbiguousAccess 'm' and the syntactic classification scheme terminates.

In this case, the qualified name m.x could not be completely classified in the syntactic classification framework, since the access corresponding to m could be either a class access or a component access. Classification of ambiguous names, as is necessary in this case, will be discussed in the next section. □

Resolution of Ambiguous Names

The syntactic classification of ParseAccess nodes results in some names which are unambiguously classified, but usually also in some ambiguous names. In such cases, the syntactic context of a particular name access is not sufficient to determine the meaning of the name. Instead, the name is looked up as described in the next section in order to find visible component or class declarations. If a matching declaration is found, the AmbiguousAccess node is rewritten into a corresponding node class,

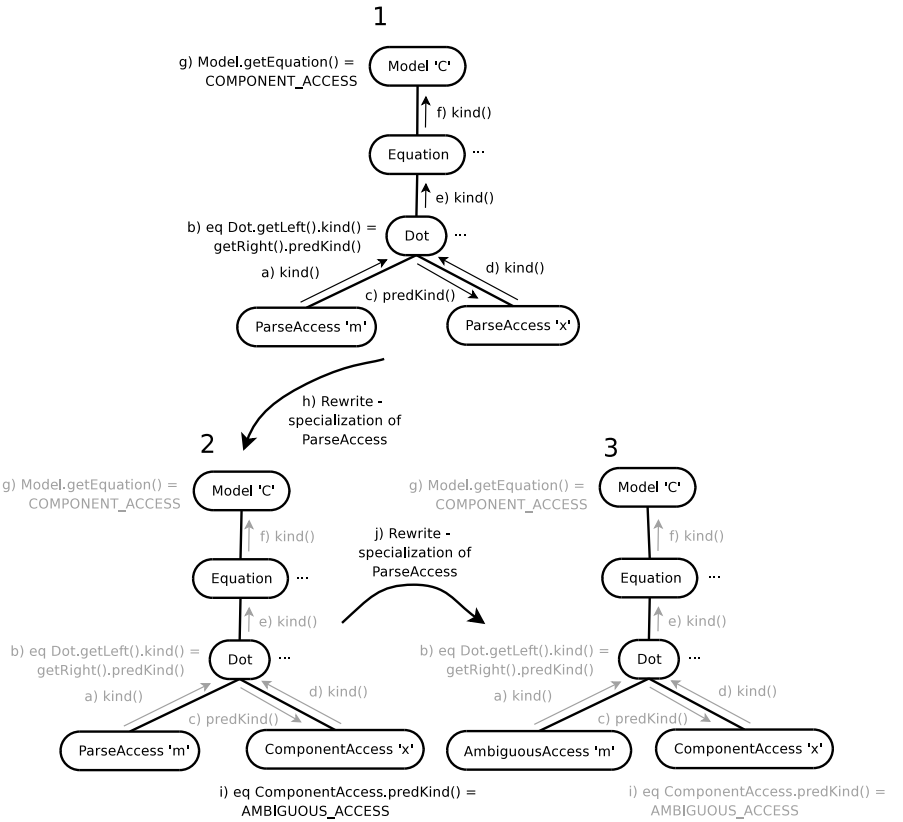


Figure 5.2 Syntactic classification of names.

i.e., `ClassAccess` or `ComponentAccess`, see Listing 5.2. The rewrite is performed only if the AST API method `duringNameClassification()` evaluates to `false`. This condition is necessary in order for the resolution of ambiguous names not to interfere with the syntactic classification scheme. If neither a class declaration nor a component declaration with a matching name is found, the node remains a `AmbiguousAccess` node, in which case an error message should be generated. Notice that the rewrite conditions are dependent on the lookup mechanism, which will be discussed in Section 5.3.

```
rewrite AmbiguousAccess {
  when (!duringNameClassification())
       to Access {
          if ((lookupDecl(getID())!=null))
             return = new ComponentAccess(getID());
          else if((lookupClass(getID())!=null))
             return = new ClassAccess(getID());
          return this;
       }
 }
```

Listing 5.2 Classification of ambiguous names.

5.3 Binding Names

The purpose of name analysis is to bind names to their corresponding class or component declarations. Such bindings are extensively used for different purposes in the compiler implementation. For example, when the type of an expression is computed, it is necessary to retrieve information about the type of individual identifiers (names) which occur in the expression. In Modelica, lookup of component names and class names is performed slightly differently, and the two lookup mechanisms will be treated separately in the following presentation. However, it is important to notice that they are mutually dependent.

To implement name lookup, the usual approach in attribute grammars is to use an inherited attribute *env* which contains all visible symbols at that point in the AST. This works fine for simple block-structured scopes, but becomes cumbersome when dealing with qualified names and inheritance. Instead, in the approach presented in [Ekman and Hedin, 2006], which is also used in the PicoModelica compiler, the lookup mechanism is implemented using parametrized reference attributes.

Component Names

Let us first consider ordinary qualified names like `a.b.c`. Here, `c` should be looked up in a scope decided by `b`, which in turn should be looked up in a scope decided by `a`. The scopes may potentially be complex due to inheritance from other classes. To implement such normal object-oriented name analysis we have applied the same basic design as in the JastAdd Java implementation [Ekman and Hedin, 2006], namely *delegating lookup attributes*.

The framework for lookup of component names is based on the parametrized, inherited attribute `lookupDecl(String name)`, which is defined for the class `Access`, see Figure 5.3. The argument `name` is the name to be

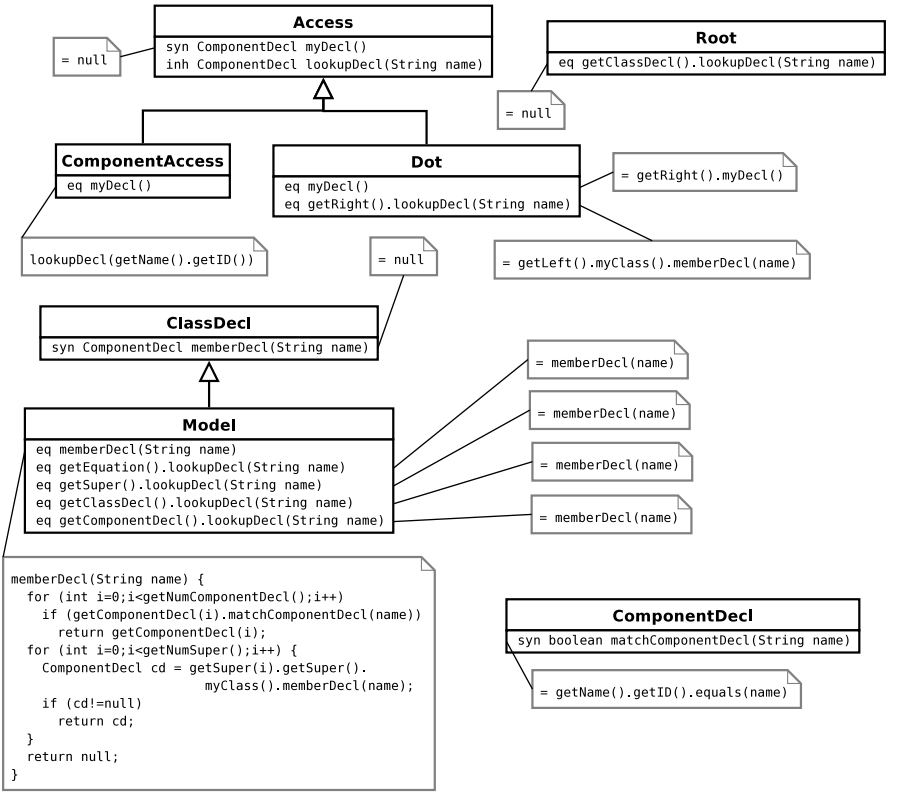


Figure 5.3 Framework for binding component names to their corresponding declarations.

looked up amongst the set of visible component declarations. Ignoring, for a moment, modifications, the set of visible component declarations is given by all declarations contained in the first entry in the ordered set of parents. That is, all component names hierarchically contained in a class, are looked up amongst the declarations of the mentioned class. PicoModelica does not support constants, which is the single case in Modelica in which it is legal to reference a component outside the containing class. Accordingly if a matching declaration is not found in the enclosing class, the name lookup fails. This semantics is expressed by defining equations for the `lookupDecl` attribute in the AST class `Model`, which corresponds to a composite class. The equations delegate lookup to the synthesized attribute `memberDecl(String name)`, which is defined for the `Model` class. The attribute `memberDecl`, in turn, is computed by searching the set of com-

ponent declarations contained in the Model node itself, and declarations contained in potential super classes. The search for matching declarations is supported by the attribute matchComponentDecl(String name) defined for ComponentDecl. If a matching declaration is found, it is returned, otherwise null is returned.

Lookup of qualified names requires an additional equation. Obviously, a name in a qualified name is to be looked up in the class which is bound to the name immediately to its left. To implement lookup for the right-hand side, the Dot node delegates to the class of the left-hand side, as follows:

```
eq Dot.getRight().lookupDecl(String name) =
  getLeft().myClass().memberDecl(name);
```

Here, Dot defines the attribute lookupDecl of its child getRight. The value is found by accessing the reference attribute myClass of the Left child and delegating to memberDecl. The attribute myClass defines the class declaration, which an Access node is bound to, and will be discussed in detail in the next section. Notice that the Dot class does not need to explicitly define the attribute lookupDecl of its left access, because there is an equation higher up in the AST which gives a default definition of this attribute for the whole subtree. In order for the lookupDecl attribute to always be defined, an equation is added to the Root class, which returns null, indicating that no declaration was found.

For convenience, an additional attribute myDecl, defined for Access, is introduced to provide easy access to the declaration of an Access node, see Figure 5.3. The component lookup mechanism will now be illustrated by means of an example.

EXAMPLE 5.2
Consider the model in Example 5.1. In this example, the attribute evaluations that proceed the syntactic classification in Example 5.1 will be explained. The corresponding (partial) ASTs are shown in Figure 5.4.

a) When the node ParseAccess 'm' has been rewritten to a node of the class AmbiguousAccess (see Example 5.1, **j)**), the rewrite condition of the rewrite in Listing 5.2 is checked. This, in turn, renders the inherited attribute lookupDecl to be evaluated, with the argument 'm'.

b) Since no equation defining the value of lookupDecl is present in Dot for the Left child, the call is propagated upwards in the AST.

c) Equation does not define lookupDecl either, and the call is propagated further upwards.

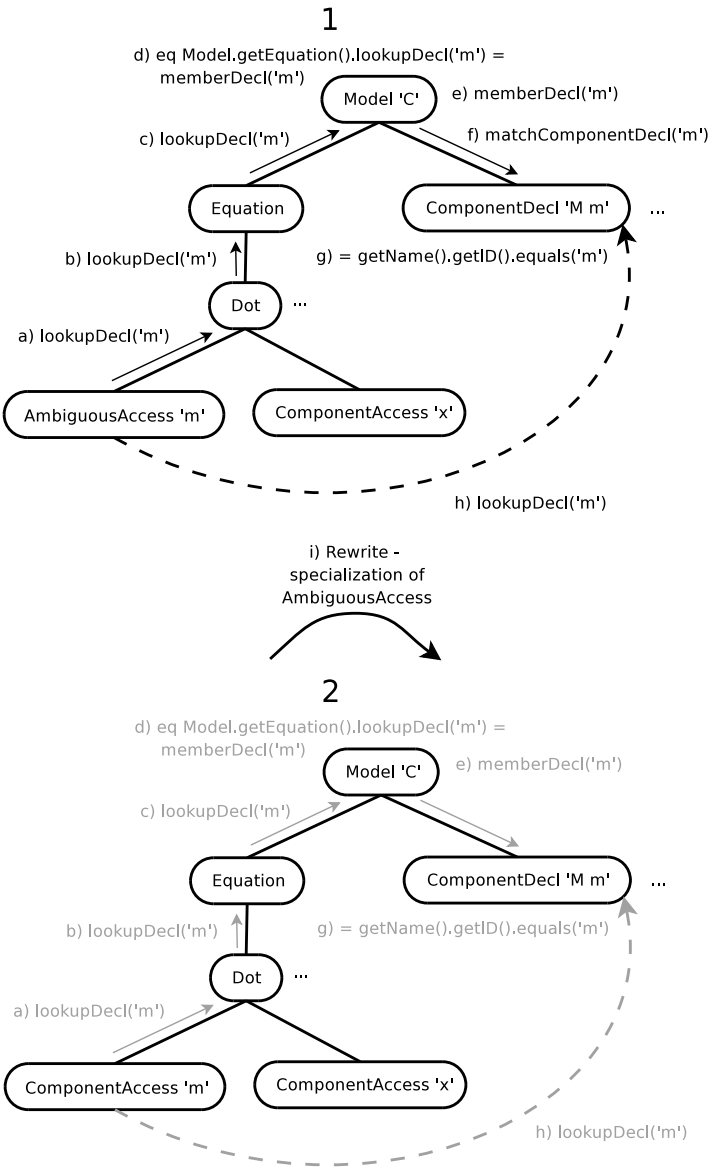


Figure 5.4 Lookup of component names.

d) An equation for `lookupDecl` is defined in `Model`:

```
eq Model.getEquation().lookupDecl(String name) = memberDecl(name);
```

e) The synthesized attribute `memberDecl` is evaluated for the `Model` node, with the argument `'m'`.

f) During the computation of `memberDecl`, all the components contained in the `Model` are retrieved, and for each declaration, the attribute `matchComponentDecl` is evaluated.

g) The default equation for the attribute `matchComponentDecl` is evaluated. In this case, the name which is being looked up matches the component name, and accordingly, the attribute evaluates to true.

h) A reference to the node `ComponentDecl 'M m'`, corresponding to the value of the attribute `lookupDecl` evaluated for the node `AmbiguousAccess 'm'`, is returned.

i) The first rewrite condition in the rewrite in Listing 5.2 evaluates to true and the node `AmbiguousAccess 'm'` is rewritten to `ComponentAccess 'm'`.

In this case, the evaluation of `lookupDecl` was triggered by the rewrite condition in the rewrite in Listing 5.2. Notice, however, that if the attribute is accessed once again, (and if the attribute is declared to be `lazy`), then the result is cached. In such case, the actual computation of the attribute is not repeated, but rather, the stored reference to the node `ComponentDecl 'M m'` is returned directly. □

Class Names

Class names are looked up similarly to component names. The lookup mechanism is based on the parametrized inherited attribute `lookupClass(String name)`, which is defined for the `Access` class, see Figure 5.5. The definition of the attribute is delegated to the class `Model`, which may contain local classes, and `Root` which is the AST root node and contains one or more class declarations. In the latter case, the equation for `lookupClass` is performed by searching the set of class declarations contained in the `Root` node for a matching declaration name. The actual search is supported by the synthesized attribute `matchClassDecl(String name)` which is defined for the class `ClassDecl`.

Lookup of class names in a `Model` node is slightly more elaborate than in the case of component names, due to the rules for lookup of names occurring in extends clauses. The Modelica specification states that *"The lookup of base-classes should be independent"*. This means that in a legal

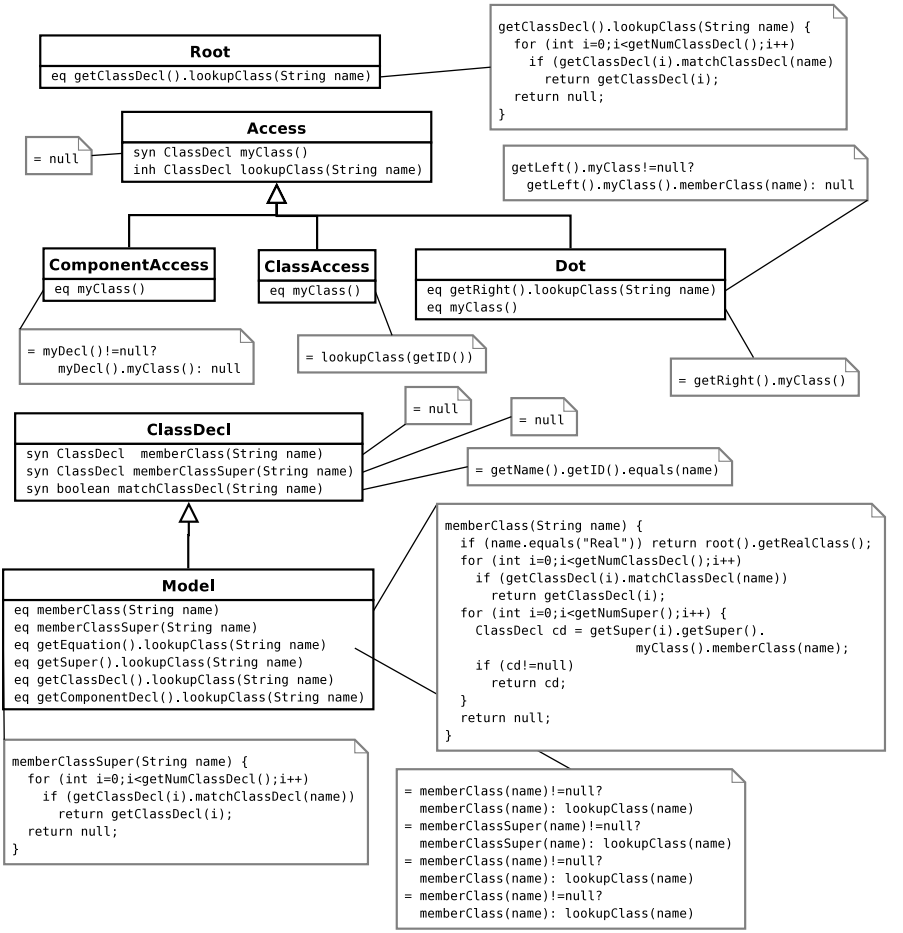


Figure 5.5 Framework for binding class names to their corresponding declarations.

Modelica program, it should be possible to look up the name of a super class *without* having to consider the content of other super classes contained in the same class. As a consequence, it is not valid to inherit from a local class which becomes visible through inheritance. Class names in component declaration clauses, however, are looked up amongst all visible class declarations, also those which become visible through extends clauses.

```
model C
  model P
    model A
      Real x=1;
    end A;
  end P;

  extends P;
  extends A; // This is illegal!
  A a;       // This is legal
end C;
```

Listing 5.3 Examples of legal and illegal use of class names which become visible through inheritance.

EXAMPLE 5.3
Consider Listing 5.3. While it is legal to inherit the local class `P`, it is illegal to inherit the class `A`, which becomes visible only after the clause `extends P` has been considered. Clearly, this violates the rule that lookup of class names in extends clauses must be independent. It is legal, however, to declare components from classes that become visible through inheritance, for example `A a`. □

To summarize, class names are looked up differently depending on their context, i.e., whether or not they are contained in an extends clause. The need to differentiate inherited attribute definitions with respect to context is natively supported in JastAdd. When an equation for an inherited attribute is defined, the particular child node for which the definition is valid must also be specified. Considering each child of a particular node to be a root in a subtree, it is thus possible to introduce attribute definitions which depend on the origin, i.e., in which subtree, of the attribute access that rendered evaluation.

Lookup of class names is delegated to either of the attributes `memberClass(String name)` or `memberClassSuper(String name)`, which are defined for the AST class `Model`. In the default case, when the class name does not reside in an extends clause, `memberClass` is used:

```
eq Model.getComponentDecl().lookupClass(String name) =
  memberClass(name)!=null? memberClass(name):
    lookupClass(name);
eq Model.getEquation().lookupClass(String name) = ...
eq Model.getClassDecl().lookupClass(String name) = ...
```

The actual implementation of `memberClass` is shown in Figure 5.5. Here, the set of all local classes contained in the `Model` node itself is searched,

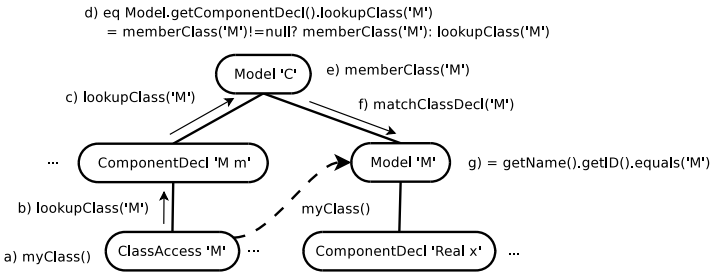


Figure 5.6 Lookup of class names.

as well as those of potential super classes. If the lookup of the class name originates from an extends clause, the computation is instead delegated to `memberClassSuper`. In this case, only the set of local classes of the `Model` node is searched for a matching declaration:

```
eq Model.getSuper().lookupClass(String name) =
  memberClassSuper(name)!=null? memberClassSuper(name):
    lookupClass(name);
```

If a class name is not found in a particular scope, lookup should proceed in the ordered set of parents. This is achieved in the JastAdd implementation by accessing the attribute `lookupClass`, which is broadcasted also to the `ClassDef` class, if no matching class declaration was found in a particular `Model` node. The computation of the `lookupClass` attribute is then propagated upwards in the AST, to the next enclosing `Model` node, which in turn corresponds to the next entry in the ordered set of parents. The class name lookup framework will now be now be illustrated by means of an example.

EXAMPLE 5.4
Consider, again, the model in Example 5.1. In this example, evaluation of the attribute `myClass` defined for `ClassAccess` will be illustrated. The corresponding (partial) AST is shown in Figure 5.6. It can be noticed that the original `ParseAccess 'M'` node, see Appendix C, has been rewritten to a `ClassAccess 'M'` node in the syntactic classification framework.

a) The synthesized attribute `myClass` in `ClassAccess 'M'` is accessed.

b) Evaluation of `myClass` renders the inherited attribute `lookupClass` to be accessed, with the argument `'M'`.

c) Since no equation for `lookupClass` is defined for `ComponentDecl`, the call is propagated upwards in the AST.

d) For `Model`, an equation for the attribute `lookupClass` is defined:

```
eq Model.getComponentDecl().lookupClass(String name) =
memberClass(name)!=null? memberClass(name): lookupClass(name)
```

This equation defines the value of `lookupDecl` if there is a local class with a matching name contained in the Model itself, otherwise, the class lookup is delegated to the next scope.

e) Evaluation of the equation for `lookupClass` in `Model` results in the synthesized attribute `memberClass` being evaluated.

f) During the computation of `memberClass`, in turn, the attribute `matchClassDecl` is evaluated for the `Model` node corresponding to the local class `M`.

g) The default equation for `matchClassDecl` returns true in this case, since the name of the class declaration matches the name being looked up. Accordingly, a reference to the corresponding `Model` node is returned. Notice that in this case the condition in the equation in step **d)** evaluates to true, and the lookup is not continued in the enclosing scope.

h) Evaluation of `myClass` terminates, and the value of the attribute is a reference to the `Model` node corresponding to `M`.

Notice that the computation of the attribute `lookupClass` is similar to that of `lookupDecl`, that was illustrated in Example 5.2. In the case of class name lookup, however, lookup continues in the enclosing scope if there is no matching class declaration contained in the `Model` itself. □

Names in Modifications

In the sections above, lookup of class and component names in an ordinary object-oriented setting was discussed. Modelica modifications, however, introduce additional rules for name lookup which must be considered. In particular, some names which occur in a modification clause should be looked up in the default scope, that is, amongst the declarations of the enclosing class, whereas other names should be looked up in a different scope. The latter scope is typically defined by a class declaration node located in another part of the AST. In PicoModelica, there are two kinds of modifications which contain names which should not be looked up in the default scope, namely component modifications and component redeclare modifications. The following examples illustrate the difficulties when binding names in modifications:

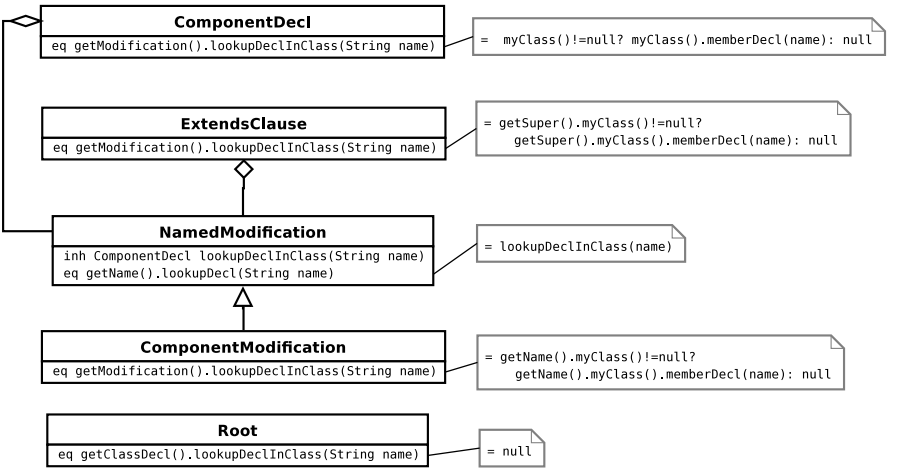


Figure 5.7 Extension of the name lookup framework to also support modifications.

EXAMPLE 5.5
Consider the following component declaration `A a(b=c)` which declares `a` to be a component of class `A`, but modified so that the variable `b` has the value `c` (rather than the value defined in the class `A`). In this case, `A` and `c` are looked up in the normal scope, i.e, the enclosing class. However, `b` should be looked up in the scope defined by the class `A`, and in order for the modification to be legal, there should be a component named `b` in `A`. □

EXAMPLE 5.6
A similar lookup mechanism is used in the case of a component redeclare construct, for example `A a(redeclare B b)`. In this case, the declaration `b` should be bound to a corresponding declaration contained in the class `A`. □

The AST classes for these constructs are defined in the abstract grammar by:

```
abstract NamedModification : Modification ::= Name:Access;
ComponentModification : NamedModification ::= Modification;
ComponentRedeclare : NamedModification ::= ComponentDecl;
```

The common superclass `NamedModification` is introduced in order to provide a common interface to the AST classes `ComponentModification` and `ComponentRedeclare`, which in turn simplifies the implementation of the

extended lookup framework. Now, the new lookup mechanism for names in modifications is implemented by delegating lookup to the new inherited attribute `lookupDeclInClass(String name)`, which is defined for the class `NamedModification`, see Figure 5.7. This attribute is used to define the attribute `lookupDecl` in the class `NamedModification` by introducing a new equation:

```
eq NamedModification.getName().lookupDecl(String name) =
  lookupDeclInClass(name)
```

This equation delegates the computation of `lookupDecl` to the attribute `lookupDeclInClass`, if the name to be looked up is contained in a `NamedModification` node. The attribute `lookupDeclInClass`, in turn, is defined in AST classes where the scope of the lookup can be determined. For example, a defining equation is added to the class `ComponentDecl`:

```
eq ComponentDecl.getModification().
  lookupDeclInClass(String name) =
    myClass()!=null? myClass().memberDecl(name): null
```

The attribute is computed by accessing the class of the component declaration, through the attribute `myClass`, and then invoke the ordinary lookup mechanism in the scope defined by this class. Similar equations are added to the classes `ExtendsClause` and `ComponentModification`, see Figure 5.7.

These examples illustrate how complex semantic rules can be encoded in a compact manner in JastAdd, and how different parts of the specification can be modularized. In this case the framework that handles lookup of names in modifications has been separated from the specification that handles ordinary object-oriented name lookup.

5.4 Type Analysis – Subtype Computation

Modelica has a structural type system in which relations such as type equivalence and subtype are determined by the structure of classes rather than through explicit declarations. Two unrelated classes may thus be type equivalent if they declare the same named elements, i.e., local classes and components. The concept of types in Modelica was briefly discussed in Section 2.3.

In this section, computation of the subtype relation for classes and components in PicoModelica is treated. The subtype relation is particularly important in a Modelica compiler, due to the complex type system and type rules. Subtype conditions are often checked when determining the validity of class or component redeclarations in parametrized classes. It is important to notice that the type system of Modelica differs from that

of, for example, Java. In Java, a class uniquely defines a type, whereas in Modelica, the interface, i.e., the named public elements of a class defines it type.

In this section, the concept of types in Modelica, and in particular different interpretations of the specification in this respect, are discussed. Also, a basic framework for computation of the subtype relation in the PicoModelica compiler is presented. The proposed implementation is based on the design principles for type analysis presented in [Ekman, 2006], but it has been adapted to the context of Modelica.

The Concept of Types

While the specification [The Modelica Association, 2005] does not formally define a type, definitions of the subtype relation are given for classes and components. It should be mentioned, however, that there is on-going work in this area, see for example [Broman *et al.*, 2006]. Also, the specification of Modelica 3.0 [The Modelica Association, 2007a], contains an improved and extended notion of types. As was noted previously, however, this work is based on the specification of Modelica 2.2. For classes, the subtype relation is defined as:

Definition 5.1—Subtyping of classes, [The Modelica Association, 2005]
For any classes S and C, S is a supertype of C and C is a subtype of S if they are equivalent or if:

- every public declaration element of S also exists in C (according to their names), and
- those element types in S are supertypes of the corresponding element types in C.

□

An element is defined in [The Modelica Association, 2005] as either a class declaration, a component clause or an extends clause. Clearly, this definition allows for classes to be in the subtype relation without being related by inheritance. For components, the subtype relation is defined as:

Definition 5.2—Subtyping of components, [The Modelica Association, 2005]
Component B is subtype of A if:

- Both are scalars or arrays with the same number of dimensions, and
- The type of B is subtype of the base type of A (base type for arrays), and

- For every dimension of an array
  - The size of A is indefinite, or
  - The value of expression (size of B) - (size of A) is constant equal to 0 (in the environment of B)

□

Neglecting the conditions applicable for arrays, which are not part of PicoModelica, the subtype condition for components reduces to *"The type of B is subtype of the base type of A"*. From a tool development point of view (and perhaps other points of views as well), this statement is troublesome, since the concept of type of a component is not further elaborated on, or defined. One possible interpretation is that the component `B b` is a subtype of the component `A a`, if the class `B` is a subtype of `A`, according to the above definition of subtyping of classes.

Now, this interpretation may be overly conservative, since it is not legal to access classes through components in Modelica. For example, the access `a.B` is illegal if `a` is a component and `B` is a local class contained in the class of `a`. Therefore, it may be reasonable to exclude local classes when checking the subtype condition for components. This approach is also taken in [Broman *et al.*, 2006][1]. This interpretation will be used in the implementation of the subtype computation presented here.

Computation of the Subtype Relation

The subtype test is defined in the compiler by the parametrized synthesized attribute `subtype(ClassDecl superType)` and the synthesized attribute `subtype(ComponentDecl superType)` which are defined for the AST classes `ClassDecl` and `ComponentDecl` respectively, see Figure 5.8. The attributes evaluate to `true` if the argument is a supertype of the node representing the `ClassDecl` or `ComponentDecl` for which the attribute is evaluated. The subtype test for classes and components are different, as described above. In order to support computation of both kinds of tests without repeating code, an additional attribute, `subtypePar(ClassDecl superType, boolean objType)`, is introduced for `ClassDecl`. The second argument here indicates whether the test is to be performed without considering local classes, i.e, taking only component declarations into account.

When computing the subtype relation it is convenient to have different rules for different kinds of classes. Since Java only supports dispatch on the receiver the *double dispatch*, [Ingalls, 1986], pattern has been used

[1] Dymola version 6.0b as well as OpenModelica version 1.4.2 also neglects local classes when checking the subtype condition in the case of component redeclarations.

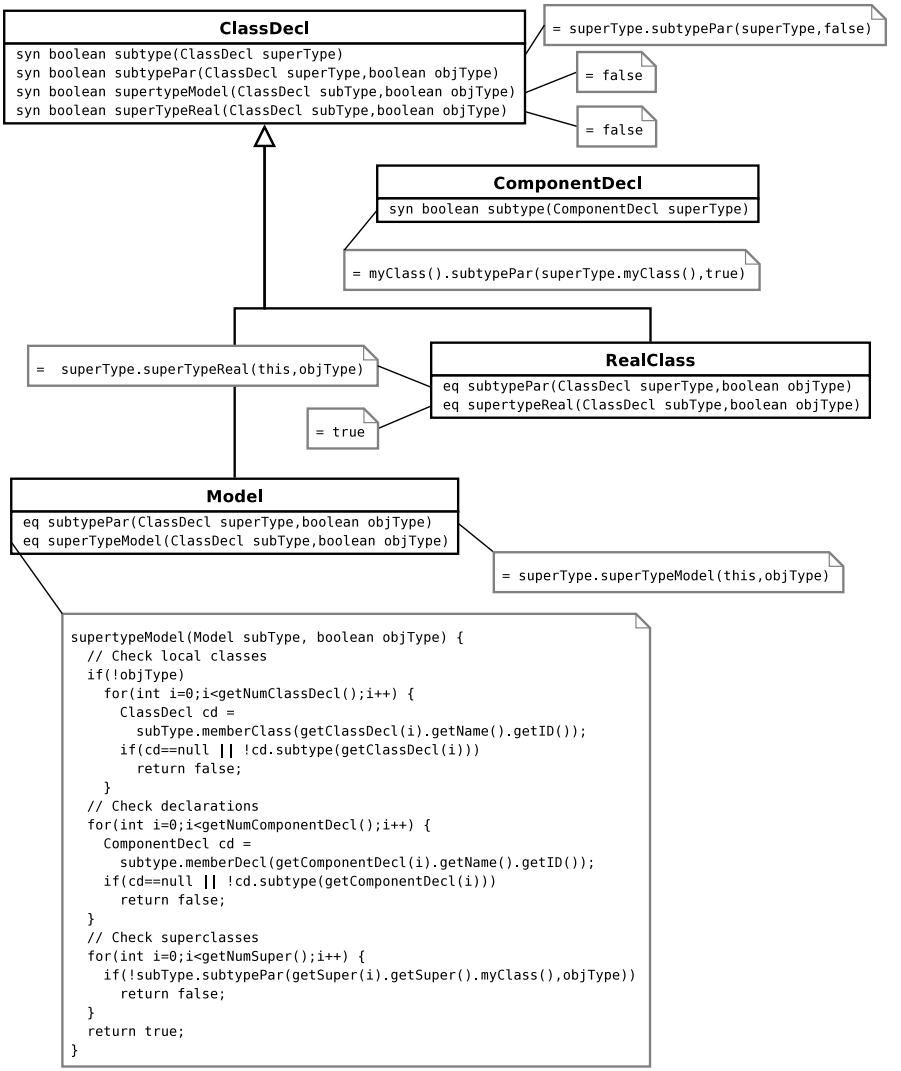


Figure 5.8 Framework for type analysis.

to implement a binary method where the target method is selected based on the type of both the receiver and the argument. This method for type analysis is used also in [Ekman, 2006]. Consider the equation for the attribute subtypePar defined for `Model`

```
eq Model.subtypePar(ClassDecl superType, boolean objType) =
```

```
  superType.supertypeModel(this, objType)
```

Using this strategy, the first dispatch is performed based on the type of the receiving object, in this case `Model`. The second dispatch is performed based on the type of the argument, in this case the type of `superType`. The actual computation of the subtype relation is thus delegated to the attribute `supertypeModel`. Notice that the double dispatch pattern renders the actual computation to be performed for the supertype relation. `RealClass` also defines the attribute `subtypePar` which delegates computation to the attribute `supertypeReal` in the same manner.

In PicoModelica, a `Model` class can not be a subtype of `RealClass`, or vice verse. It is therefore convenient to introduce a generic definition of the attribute `supertypeModel` for `ClassDecl`, which by default evaluates to false. An equation defining the attribute for `Model` is then added, which performs the actual supertype test for two models. The implementation is shown in Figure 5.8, and follows from the definition of subtyping of classes. If the test is to be performed considering local classes, (i.e. if the argument `objType` is `false`), the list of local classes is traversed and the subtype condition is checked for each class. Next, the component declarations are checked, and, finally, the superclasses. The test for `RealClass` is trivial.

The subtype test for `ComponentDecl` is defined by the attribute

```
syn boolean ComponentDecl.subtype(ComponentDecl superType) =
  myClass().subtypePar(superType.myClass(),true)
```

The actual computation is delegated to `subtypePar` defined for `ClassDecl`, with the second argument set to `true`. This indicates that local classes are not to be included in the computation of the subtype relation.

5.5 Summary

In this chapter, PicoModelica, has been used to demonstrate the core aspects of Modelica name and type analysis. It has been shown how design concepts developed for the JastAdd extensible Java compiler (JastAddJ), [Ekman and Hedin, 2007], can be adapted and applied to the context of the PicoModelica compiler. The framework used in the JastAddJ compiler has been extended to support non-trivial language features in in Modelica, such as structural subtyping, and name-lookup in modifications and redeclarations. It has been shown how these extensions can be expressed concisely in a declarative and modular way using JastAdd. The concept of types in the Modelica language has also been discussed, and difficulties in interpreting the Modelica specification, [The Modelica Association, 2005], have been high-lighted.

6

Flattening of Modelica Models with JastAdd

While object-oriented equation-based modeling in Modelica is beneficial for the modeler, the source code is not immediately suited for use with numerical algorithms. Typically, algorithms for simulation or optimization require a model to be represented in a form which is closely related to its mathematical definition, which is in the Modelica context usually referred to as the underlying Hybrid DAE. This model representation is flat, in the sense that it contains variables and equations, but no structural entities such as classes or components. The process of transforming a Modelica model into a flat representation is called *flattening*.

A Modelica program differs from many traditional programming languages in that it is not intended to be executed under the assumption of a program counter, which defines the point of execution. Neither is there a stack or heap. Rather, a Modelica model is defined by a static composition of components. The entire concept of nested components can therefore be eliminated at compile-time. All hierarchical constructs are then removed and the result is a flat description consisting of a set of uniquely named variables of primitive type, possibly arrays, and a set of equations and algorithms that operate on primitive values and variables only. The flat model may then be transformed further and eventually used for, for example, code generation to a numerical simulation algorithm.

Flattening of Modelica models is challenging due to *modifications*, which may change the value of a primitive variable or even change an entire declaration. In the former case the term *value modification* will be used and the latter case will be referred to as a *structural modification*[1]. As a consequence, the environment, consisting of a set of modifications,

[1]It is possible for a value modification to actually change the structure of the model, if it affects the size of an array. Such modifications may be viewed both as a value modification and structural modifications, but will not be considered here, for the sake of brevity.

must be considered when flattening a particular class. The concept of environments, and how to construct them, is therefore a key issue when designing a flattening algorithm.

The procedure of flattening a Modelica class is sometimes referred to as *instantiation*, or *elaboration*. The meaning of these terms is equivalent, however. In this chapter, the term *flattening* will be used.

This chapter starts with an introductory example of a flattening algorithm, which does not consider modifications, in Section 6.1. In Section 6.2, the modification mechanism is introduced and the effect of value and structural modifications are explained. In Section 6.3, the concept of the *instance AST*, which will play a key role in the flattening algorithm, is introduced. In Section 6.4, construction of the instance AST is explained conceptually and in Section 6.5, a JastAdd implementation for declarative construction of instance AST:s is presented. In Section 6.6, modifications, and their effects on the construction of the instance AST are discussed. In Section 6.7 the concept of environments is introduced, and in Section 6.8, computation and representation of environments in the instance AST are discussed. Component redeclarations, which are a special kind of modifications, are treated in Section 6.9. The chapter ends with Section 6.10, in which a flattening algorithm is presented, and a summary in Section 6.11.

6.1 Simplified Flattening without Modifications

Let us consider first a simplified flattening algorithm that does not deal with the complication of modifications. A class may be used in multiple component declarations, at different locations in a model. Enclosed components and primitive variables may thus be instantiated multiple times in different environments. A flattening algorithm need therefore compute the set of primitive variables and equations in all components that occur in the model. A simple way to implement flattening is to recursively flatten composite components. Unique names for the primitive variables that are encountered during flattening can then be obtained using a kind of *name mangling*. The objective is to generate a unique *flat* name for each primitive variable. The flattening process is then parametrized by a name prefix of type `String`. When a component is flattened, its name is concatenated to this string, which is then propagated when the component's children are flattened. Consider the code in Listing 6.1. Each `ClassDecl` iterates over its components and equations, and flattens them with the current name prefix as a parameter. Another parameter, `res`, holds a container for the resulting flattened code. Each `ComponentDecl` updates the prefix by extending it with its own name. This particular name mangling strategy generates names that are not only unique but also provide trace-

```
public void ClassDecl.flatten(Collection res, String prefix) {
  for(Iterator iter = components().iterator(); iter.hasNext(); ) {
    ComponentDecl d = (ComponentDecl)iter.next();
    d.flatten(res, prefix);
  }
  // similar for equations
}
public void ComponentDecl.flatten(Collection res, String prefix) {
  String newPrefix = prefix + getName();
  if(myClass().isPrimitive())
    res.add(new FVariable(newPrefix, ... ));
  else
    myClass().flatten(res, newPrefix);
}
```

Listing 6.1 A simple flattening algorithm.

ability back to the original model. Each `ComponentDecl` of primitive type generates a new variable in the resulting container, and non-primitive components are flattened recursively with the updated environment.

6.2 Flattening and Modifications

The concept of modifications constitutes a major challenge for the Modelica compiler constructor. In essence, the reason for this is that it is not sufficient to consider only the elements of the class to be flattened, since there may be modifications in the environment that affect the type or value of an element. The environment consists of all modifications that are applicable to the class, and is built by *merging* of the modifications collected at all levels of the instance hierarchy. Notice that it is possible for a particular element to have several applicable modifications at different levels. Merging of modifications is the procedure of determining which, of potentially several, applicable modification in an environment that should be applied when flattening a particular component. The Modelica specification states that *outer* modifications override *inner* modifications, that is, modifications which are specified inside a component have lower precedence than modifications specified outside of the component.

EXAMPLE 6.1
To illustrate the concept of environments and merging, consider Listing 6.2. Flattening of the class `A`, starting with an empty environment, results in flattening of the class `B` in the environment `{c(x = 5)}`. This,

```
model A
  model B
    model C
      Real x = 1;
      Real y = 2;
    end C;
    C c(x = 3, y = 4);
  end B;
  B b(c(x = 5));
end A;
```

Listing 6.2 A Modelica model illustrating merging of modification.

```
model A
  model B
    Real x = 1;
  end B;
  model C
    extends B;
    Real y = 2;
  end C;
  model D
    replaceable B b(x=2);
  end D;
  D d(redeclare C b(y=3));
end A;
```

Listing 6.3 A Modelica model illustrating component redeclaration.

in turn, leads to flattening of `C` in the environment `{x = 5, y = 4}`. In this case, an outer modification of `x` from the declaration of `b` overrides an inner modification from the declaration of `c`. The result of the flattening procedure is thus two variable declarations: `Real b.c.x = 5` and `Real b.c.y = 4`. □

Modifications may also express replacement of a component or type declaration. In this case, the actual structure of the instance hierarchy may be changed.

EXAMPLE 6.2
Consider Listing 6.3. The class `A` contains three local classes: `B`, `C`, which is a subtype of `B`, and `D` which contains a replaceable component. Now, flattening of `A` leads to flattening of `D` in the environment `{redeclare C b(y=3)}`, which is valid since `C` is a subtype of `B`. The redeclaration modification acting on the declaration of `b` in `D` results in the class `C` being flattened instead of the class of the original declaration. The result of the flattening procedure is thus `Real d.b.x = 2` and `Real d.b.y = 3`. The variable `d.b.y` is a direct result of the redeclaration of the component `b` in class `D`, since a variable `y` is present in class `C`, but not in the class `B`. Notice also that the modification `x=2` specified for the replaced component declaration `B b` must be taken into account, even though the declaration itself is redeclared. □

The examples in this section show that it is not sufficient to perform *local* tests, for example type checking, of the correctness of a Modelica program. Instead such tests must be performed considering both the class to be

flattened *and* its environment.

From the examples, it is also clear that the simplified flattening algorithm given in Section 6.1 is not sufficient in the presence of modifications. Instead, a flattening framework based on the concept of the *instance hierarchy* will be introduced in the next section. This data structure is orthogonal to the class hierarchy, in the sense that it reflects the hierarchical component structure of a particular model, rather than relations between classes, such as inheritance and aggregation. The strategy of creating a new data structure which is used for flattening has been previously used to translate Omola models, see [Andersson, 1994].

6.3 The Instance AST

The procedure of flattening a Modelica model is closely related to the *instance* hierarchy. This idea is described in [Andersson, 1994], in the context of flattening of Omola models. The instance hierarchy is conveniently represented as an abstract syntax tree, which reflects the structure of a particular model instance. In the following, the original syntax tree resulting from parsing of a Modelica file will be referred to as the *source* AST, and the corresponding instance hierarchy will be referred to as the *instance* AST. Further, the term *source node* will be used to refer to a node in the source AST and *instance node* will be used to refer to a node in the instance AST.

Each instance node corresponds, conceptually, to a component declaration. (This assumption will be relaxed in the following section.) A key observation is that the instance AST is recursively defined, in the sense that the children of a given instance node are computed from the set of components which are hierarchically contained in the class of the corresponding component declaration. This observation will be explored when constructing the instance AST. Also, the leaves of the instance AST correspond to variables of primitive types.

Each node in the instance AST has a reference, referred to as an *inter-AST reference*, to a node in the source AST. The inter-AST references enable access to relevant sub-trees in the source AST, such as modifications. The inter-AST references are similar to attributes, but differ in that they are defined when an AST is built, rather than through equations. This type of attributes are referred to as *intrinsic attributes*, see [Farrow, 1982]. More commonly, intrinsic attributes are used to store information from the parsing phase, such as identifier strings and literal values. In this case, however, the intrinsic attributes hold references to nodes in another AST. It is important to note that the source AST and the instance AST are two separate AST:s, and that the inter-AST refer-

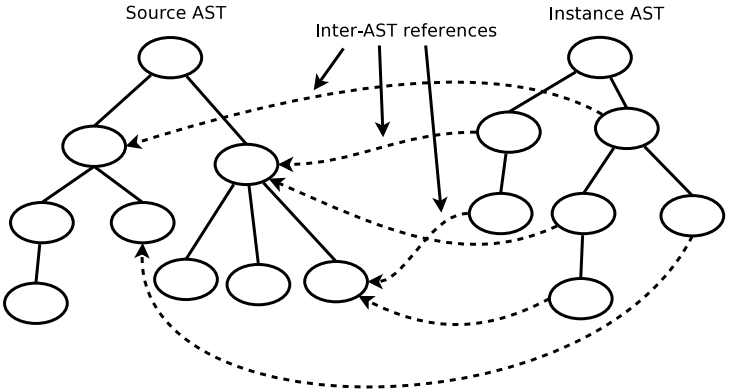


Figure 6.1 The relation between a source AST (left) and an instance AST (right). The dotted arrows represent intrinsic attributes.

ences link the nodes in the instance AST to corresponding nodes in the source AST. Clearly, a particular node in the source AST may be the target of several inter-AST references, if the corresponding class declaration is used in multiple locations in a model, see Figure 6.1. It is also worth mentioning that the actual structure of the instance AST may depend on modifications, which complicates its construction.

Notice that the instance hierarchy is orthogonal to the class hierarchy. The latter defines relations between classes in a model, such as inheritance and association, whereas the former represents the structure of a particular model instance.

The instance AST has several interesting properties which can be exploited by a flattening algorithm. Since the instance AST represents the instance hierarchy, the environment of any given component can be computed by searching the AST upwards, assuming that each instance node has a reference to its corresponding component declaration, and thereby also to associated modifications. Further, the flat name of a component is constructed by collecting all component names upwards in the AST.

Once the instance AST has been constructed, a flattening algorithm that can handle modifications is simple to construct. It will simply traverse the instance AST and collect variables and equations. Such an algorithm replaces the simplified algorithm given in Section 6.1, and will be described in Section 6.10.

6.4 Conceptual Construction of the Instance AST

The structure of the instance AST is conveniently expressed in terms of an abstract grammar, which defines the instance node types. In addition, the inter-AST references to nodes in the source AST, represented by intrinsic attributes, are declared in the abstract grammar.

The procedure of constructing the instance AST starts by locating, in the source AST, the class node which is to be flattened. Notice that this class node need not be the root of the source AST. Rather, this class is identified using a qualified name, e.g., `A.B.C`, and the corresponding class declaration node can found by traversing the source AST.

Conceptually, construction of the instance AST starts with a root node with no children. Then, for each component declaration node in the class to be flattened, a new child node is added to the instance AST root. In addition, the inter-AST reference from an instance node is set to its corresponding source node. When instance nodes corresponding to all component declarations have been created, the algorithm proceeds recursively. By following the inter-AST reference of an instance node, the class declaration node of the corresponding component declaration can be retrieved. New children of the instance node, that correspond to the component declarations of the retrieved class declaration, are then created. The recursion for a particular branch in the instance AST terminates when a component declaration of primitive type, for example `Real`, is encountered.

An important role of the instance AST is to provide a representation of environments, which in turn consist of a set of hierarchically ordered modifications. Therefore, it is useful to introduce additional node classes in the instance AST which correspond to elements which may have associated modifications. This can be expressed in the following informal definition, which relates node classes in the source AST to node classes in the instance AST:

For each node class in the source AST, that may have a modification subtree, a corresponding node class is introduced in the instance AST.

A modification subtree is the AST corresponding to a modification clause in a Modelica source file. This definition makes it possible to design a general framework for construction of the instance AST, considering only a small number of characteristic language constructs.

6.5 Construction of the Instance AST Using NTA:s

In Section 6.4, the recursive procedure for building the instance AST was explained conceptually. Using nonterminal attributes (NTAs), the con-

```
syn lazy List InstNode.getInstNodeList() {
  List l = new List();
① List comps = components();
① List supers = superClasses();

  // Add children corresponding to all components
② for (int i=0;i<comps.getNumChild();i++) {
    ComponentDecl cd = (ComponentDecl)comps.getChild(i);
    l.add(cd.newInstComponent(cd));
  }

  // Add children corresponding to all super classes
③ for (int i=0;i<supers.getNumChild();i++) {
    l.add(((ExtendsClause)supers.getChild(i)).newInstExtends());
  }
  return l;
}
```

Listing 6.4 Definition of the attribute `getInstNodeList`.

struction of the instance AST can actually be specified declaratively, by exploring its recursive structure.

In PicoModelica, there are two constructs that allow modifications, namely component declarations and extends clauses. Ignoring, for a moment, structural modifications, the following abstract grammar can be defined for the PicoModelica instance AST:

```
abstract InstNode ::= /InstNode*/;
InstRoot : InstNode ::= <ClassDecl:ClassDecl>;
abstract InstComponent : InstNode ::=
                         <ComponentDecl:ComponentDecl>;
InstComposite : InstComponent;
InstReal : InstComponent;
InstExtends : InstNode ::= <ExtendsClause:ExtendsClause>;
```

Class names in this grammar have the prefix `Inst` to emphasize that these classes are intended to be used to construct instance nodes rather than source nodes. The children of the abstract class `InstNode` are declared as a list NTA, and are defined by a synthesized attribute as shown in Listing 6.4. The use of NTAs to declare the children of an `InstNode` is a natural choice here, since the construction of the AST may then be made dependent on the context of a particular node.

The root node of an instance AST has the type `InstRoot`, which holds an inter-AST reference to a corresponding class declaration node in the source AST, i.e. the class to be flattened. The inter-AST references are

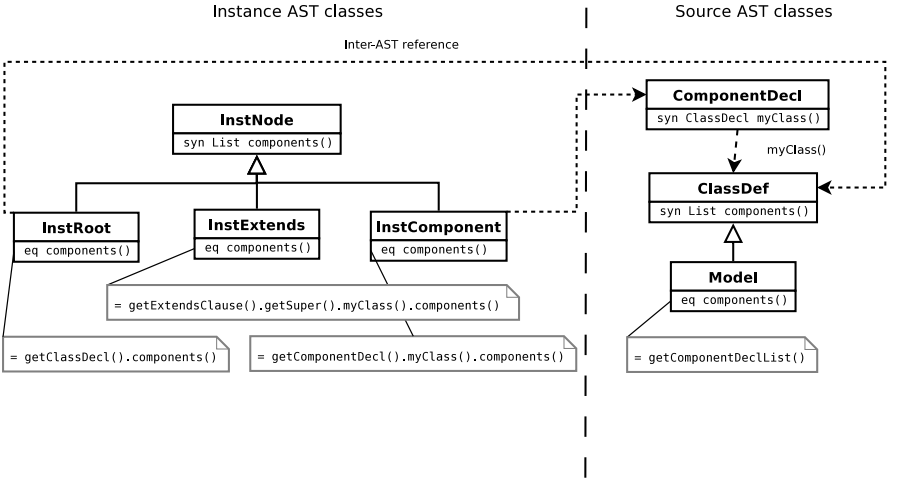


Figure 6.2 The attribute components defined for InstNode gives access to all component declarations of the InstNode.

modeled as intrinsic attributes, which are denoted by angle brackets[2]. The abstract class InstComponent has two subclasses, InstComposite and InstReal. These correspond, respectively, to a composite component and a component of the primitive type Real. Finally, the InstExtends class corresponds to an extends clause.

Let us first consider how the instance AST is built when ignoring the effects of structural modifications. By declaring the children of an InstNode as a list NTA, the tree is built recursively using the synthesized attribute getInstNodeList, which defines the NTA. Every InstNode carries a reference to a node in the source AST, for example, ComponentDecl, Model (which is a subclass of ClassDef) or ExtendsClause. By accessing the source AST nodes, the set of children of the InstNode can be computed. Consider the definition of the attribute getInstNodeList in Listing 6.4. First, the sets of components and superclasses of the source AST node are retrieved ①. The computation of these sets depends on the type of a particular InstNode, and will be discussed below. Then, for all components, new InstComponent nodes are created, and added to the set of children, ②. Finally, for all super classes, new InstExtends nodes are created, ③, and the list containing the InstNode children is returned. The actual creation of new InstNodes is performed within the scope of the source AST, and

[2]The syntax <S:T> means that the name of the intrinsic attribute is S and that its type is T. Notice that the name and type strings may be equal, for example <T:T>.

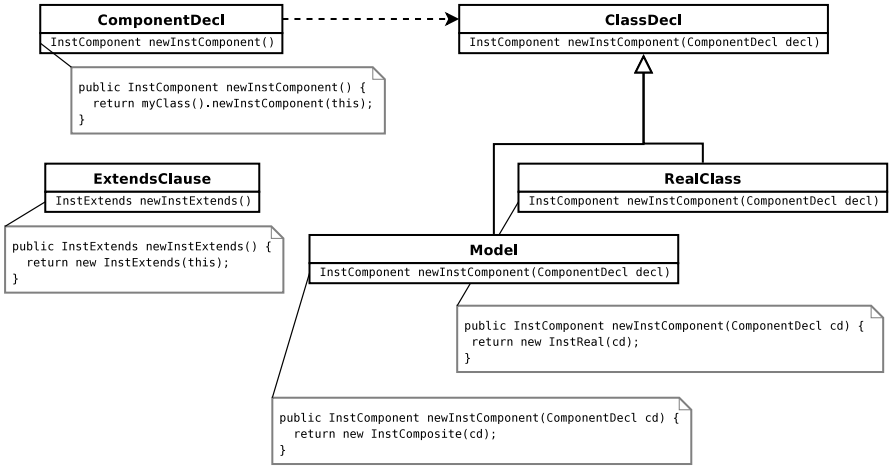


Figure 6.3 The attributes newInstComponent and newInstExtends are used to create new InstNodes based on the corresponding classes in the source AST.

will be explained below. Notice that the instance AST is built *on demand*, when the children of a particular node is accessed.

A generic interface for retrieving the components and superclasses of an InstNode is provided by the attributes components and superClasses. This interface enables the attribute getInstNodeList to be defined for the type InstNode, and thereby eliminates the need for specialized implementations for each node class. In Figure 6.2, the implementation of components is illustrated. The implementation relies on the inter-AST references for retrieving nodes in the source AST, as well as attributes previously defined in the name and type analysis, such as myClass.

It now remains to device a mechanism for creating an InstNode corresponding to a component declaration or extends clause. In PicoModelica, there are two kinds of components: composites and reals, which in turn corresponds to the instance node classes InstComposite and InstReal. By using dynamic dispatch with respect to the type of a component declaration, it is possible to distinguish between the two cases. The method newInstComponent, which is defined for the source class ComponentDecl, delegates the call to the corresponding ClassDecl. The new instance node is thus created within the scope of a ClassDecl, in the source AST, in which case the correct node type, InstComponent or InstReal can be determined, see Figure 6.3. The strategy for creating a new InstExtends is similar. Notice that new InstNodes are created using methods rather than attribute definitions, since each invocation should return a new object.

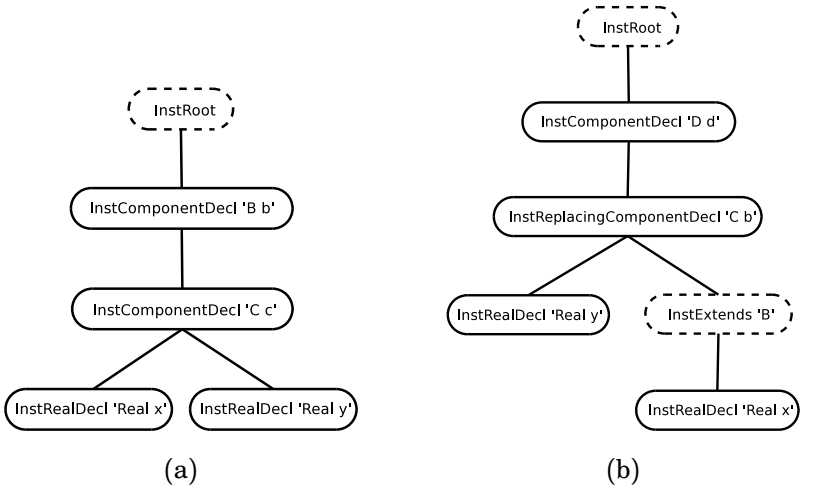


Figure 6.4 Instance AST:s resulting from the examples in Listings 6.2 and 6.3.

As an example, consider the model in Listing 6.2, and the resulting instance AST depicted in Figure 6.4a, which reflects the hierarchical component structure, as opposed to the class hierarchy. The leaves of the tree correspond to the variables `x` and `y` which are of the primitive type `Real`. A solid node indicates that a new *name scope* is introduced, whereas a dashed node corresponds to elements which do not give rise to a new scope. Examples of the latter are e.g. `InstRoot` and `InstExtends`. The concept of a name scope is closely related to the flat name of a primitive variable, in that each entry in the qualified flat name corresponds to a name scope in the instance AST.

6.6 Modification Trees

PicoModelica supports two kinds of modifications, namely value modifications and structural modifications in the form of component redeclarations. Value modifications must be considered in order bind the correct expression to a primitive variable in the flat model representation. Structural modifications on the other hand, affect how the actual instance AST is built, since a structural modification may cause a component declaration to be replaced by another declaration. Consequently, the set of children of the corresponding, replacing, instance node may differ from the set of children which would have resulted from the original declaration.

By definition, each `InstNode` has a reference to a node in the source

AST, for example a ComponentDecl, which may have a modification subtree. A modification subtree is a part of the source AST that corresponds to a modification construct in the Modelica source code, see Figure 6.5. Conceptually, it is therefore convenient to consider an instance node to be associated with a modification subtree in the source AST. Modifications associated with a particular instance node may be retrieved using the inter-AST references, see Figure 6.5. For convenience, the synthesized attribute InstNode.modificationTree(), which retrieves this modification subtree is introduced:

```
syn lazy Modification InstNode.modificationTree() = null;
eq InstComponent.modificationTree() =
  getComponentDecl().hasModification()?
    getComponentDecl().getModification(): null;
eq InstExtends.modificationTree() =
  getExtendsClause().hasModification()?
    getExtendsClause().getModification(): null;
```

This attribute provides a generic interface to modification subtrees for InstNodes. In the following, InstNodes are therefore considered to have modification subtrees directly associated with them, although the actual modification subtrees are located in the source AST. Again, notice that a modification subtree may be associated with several instance nodes.

By construction, all modification subtrees, that may contain applicable modifications for a particular instance node are located above the node itself, in the instance AST. This means that it is sufficient to search for applicable modifications for a particular instance node starting at the node and proceeding upwards towards the root of the instance AST.

The instance AST thus represents an ordering of modifications, given by the hierarchical tree structure. This ordering can be used to enforce the rules of merging of modifications, where outer modifications have precedence. This means that if several modifications applicable to a particular element is found when searching the instance AST upwards, the modification associated with the uppermost InstNode is selected. The ordered set of modifications applicable to an instance node is called an *environment*. In the next section, construction and representation of environments will be discussed.

6.7 Environments

A key concept in the flattening process is that of environments. The Modelica specification states that a class is flattened in an environment that consists of a set of applicable modifications resulting from merging. In

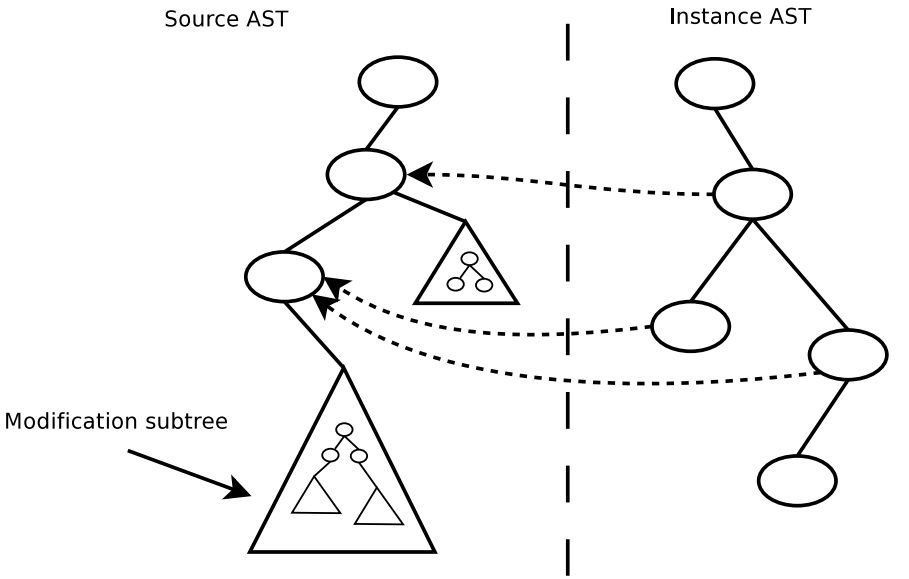


Figure 6.5 Modification subtrees in the source AST can be accessed from the instance AST using inter-AST references.

the context of the instance AST, this corresponds to each `InstNode` being associated with an environment, which must be considered when the children of the node are created. Accordingly, representation and computation of environments is a key design issue in the construction of a Modelica compiler.

For any instance node, the corresponding environment can be computed by merging of applicable modifications accessible from instance nodes located upwards in the instance AST. The importance of this observation is significant, since it implies that the environment of an instance node can be computed *before* the entire instance AST is constructed. In essence, this allows the instance AST to be constructed top-down, since when a new node is added, its environment can be computed from the part of the tree that is already built, that is above the new node.

In principle, it would be possible to construct the environment of an instance node by searching the instance AST upwards, retrieve modification subtrees associated with instance nodes encountered in the search, and then traverse the modification subtrees to identify applicable modifications. Such procedure would, although conceptually appealing, result in much unnecessary traversal of modification subtrees, as well as a cum-

bersome implementation[3].

A more efficient approach to representing environments is to associate with each instance node, a data structure representing the merged environment of the node. By introducing a list that contains entries with references to modification subtrees, located in the source AST, each instance node is provided with easy access to its environment. The list of references to modification subtrees is ordered according to the precedence rules of Modelica, and can be viewed as a projection of all applicable modifications located upwards in the instance AST onto a single data structure, see Figure 6.6.

Apart from a reference to a modification subtree, it is necessary to store a reference to the instance node which was originally associated with the modification subtree. The reason for this will become clear when discussing flattening of binding expressions retrieved from value modifications. In essence, the additional reference to an instance node is needed in order to compute the name prefix of a variable access in a binding expression.

The merged environment is then represented by a list containing entries, each holding a reference to a modification subtree and a reference to the corresponding instance node. Notice that the list entries of an instance node can hold instance node references to, either, the instance node itself, or instance node located *above* the node. Again, this is essential, in order for an instance node to have a well defined environment, even though the instance AST is only partially built.

Computation of environments is done by *merging* of modifications. The term merging refers to the procedure of determining which, of potentially several, modifications that are applicable to a particular element. There are two sources of modifications for an instance node. Outer modifications are retrieved from the immediate parent of the node and are then merged with the modifications, if any, which are associated directly with the node itself. The resulting, merged, environment then contains all modifications applicable to the elements hierarchically contained in the instance node. For example, the content of the merged environment is sought for structural modifications when the children of the instance node is created. The basic concepts of environments and merging of modifications were illustrated in Example 6.1.

[3]This strategy was implemented in an early version of the PM compiler, but was discarded, since apart from being overly complicated, it was also difficult to extend.

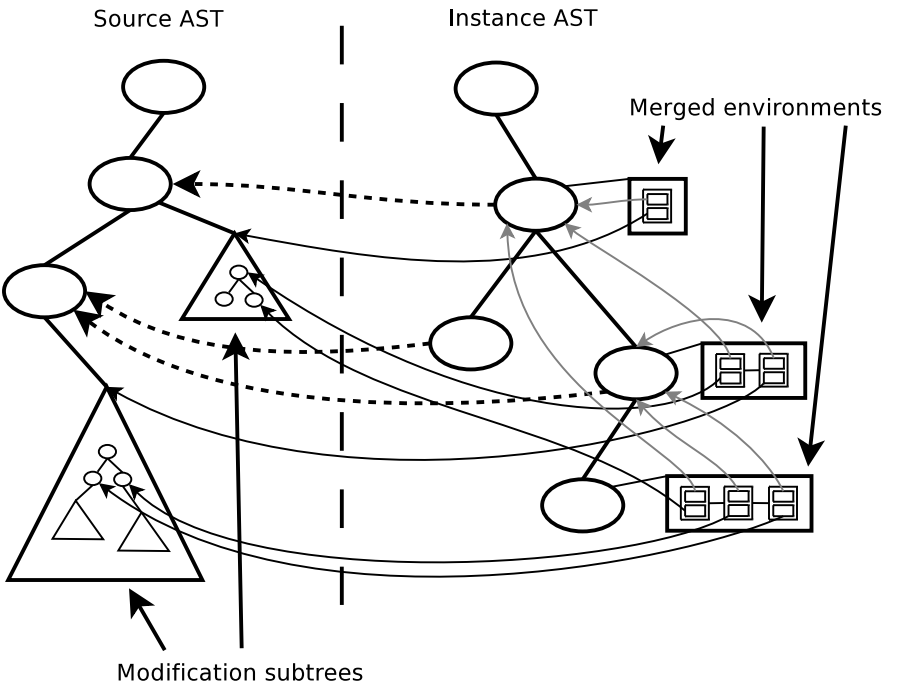


Figure 6.6 Representation of merged environments in the instance AST and their relations to nodes in the source AST.

6.8 Declarative Construction of Merged Environments

The discussion of how to introduce environments into the instance AST in Section 6.7 suggested that it is desirable to introduce a new data structure which explicitly represents an environment. The environment data structure is conveniently modeled by AST classes, which are used to create new nodes which are inserted into the instance AST, see Figure 6.6. Introduction of new nodes in an AST is supported by JastAdd through nonterminal attributes, as discussed in previous sections. Consider the following extension of the instance abstract grammar:

```
abstract InstNode ::= /InstNode*/
                      /MergedEnvironment:InstModification*/;
InstModification ::= <InstNode:InstNode>
                     <Modification:Modification>;
```

A new nonterminal attribute, `MergedEnvironment`, consisting of a list of

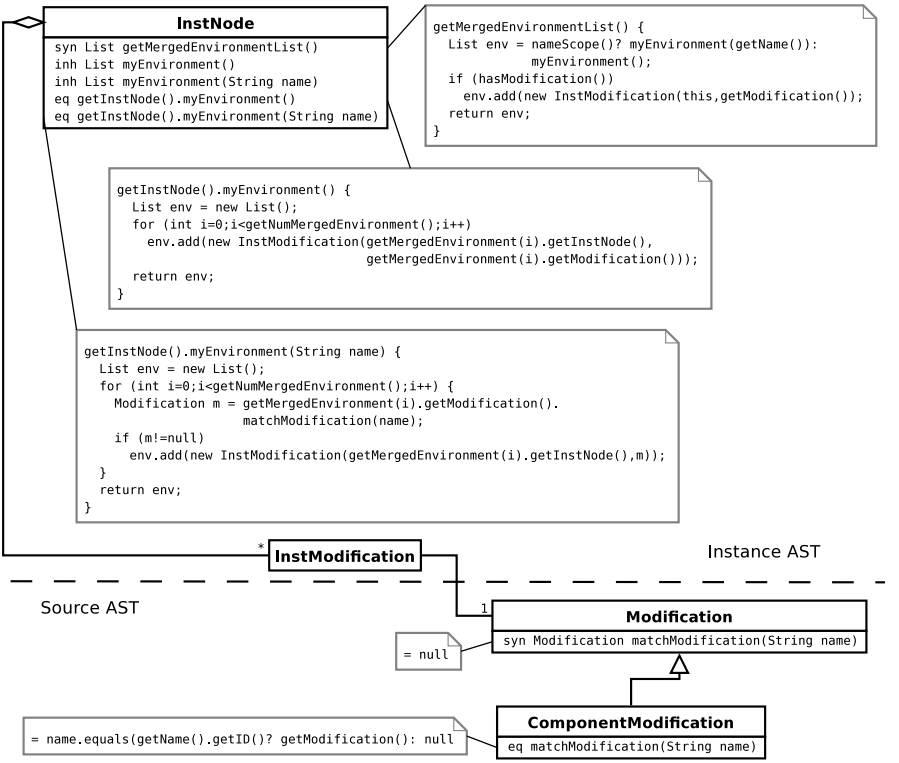


Figure 6.7 The framework for computation of environments.

nodes of type `InstModification` is added to the `InstNode` class. The type `InstModification`, in turn, serves as a wrapper for a modification node located in the source AST. The list in `MergedEnvironment` also represents the precedence between modifications, where the first entry is the outermost modification. The reference to the modification node is implemented as an inter-AST reference. The `InstModification` also keeps a reference to the `InstNode` which is associated with the modification subtree containing the modification node. As pointed out above, this is necessary in order to compute the name prefix of binding expressions. Notice that the reference to an `InstNode` contained in `InstModification` is implemented as an intrinsic attribute, but without being an inter-AST reference, since the target of the reference resides in the same AST.

Now, it turns out to be convenient to let the NTA `MergedModification` represent the environment of the *children* of an `InstNode`. This design simplifies merging of modifications as well as the procedure of locat-

ing applicable modifications. The NTA is defined by the synthesized attribute `getMergedEnvironmentList`, see Figure 6.7. The computation of `getMergedEnvironmentList`, starts by retrieving the environment of the `InstNode` itself, which contains applicable outer modifications. Finally, the set of modifications associated directly with the `InstNode` node itself is added to the list of `InstModifications`.

The environment of the instance node itself is defined by the inherited attribute `myEnvironment`, see Figure 6.7. This attribute is defined both with no arguments and with a string argument representing a component name. In the first case, the resulting environment is identical to the one defined by the `getMergedEnvironmentList` attribute of the `InstNode`'s parent. In the second case, the resulting environment contains only modifications that match the supplied component name. The need for both versions stems from the fact that some instance nodes introduce a new name scope, for example `InstComponent`, whereas others, such as `InstExtends` do not.

The search for applicable modifications in the definition of the attribute `myEnvironment(String name)` relies on the synthesized attribute `matchModification(String name)`, which is defined for the source AST class `Modification`. If the string argument corresponding to a component name matches that of a `ComponentModification`, the modification subtree represented by the child modifications of this node is returned.

EXAMPLE 6.3
To illustrate how environments are computed, consider Listing 6.2, and the corresponding instance AST in Figure 6.8. The procedure starts with an empty environment for the declaration `B b`. The environment of the children of the corresponding `InstComponent` is defined by the attribute `getMergedEnvironmentList`, and contains the modification `c(x=5)`. In the next step, the environment of the component declaration `C c`, defined by `myEnvironment("c")`, is computed. This results in a new environment containing the modification `x=5`. The procedure is repeated for each instance node `InstComponent` and terminates when a node corresponding to a primitive declaration is encountered. The binding expression of a primitive declaration, represented in the instance AST by an `InstReal`, is identified as the first modification in the corresponding environment. For example, for the declaration `Real x`, the binding expression is `=5`. □

To summarize, an advantage of the proposed framework for merging of modifications is that the environment of an `InstNode` can always be computed by using information which is available from its direct parent or which is associated with the node itself. This, in turn, eliminates the need for traversing modification subtrees whenever the environment of an in-

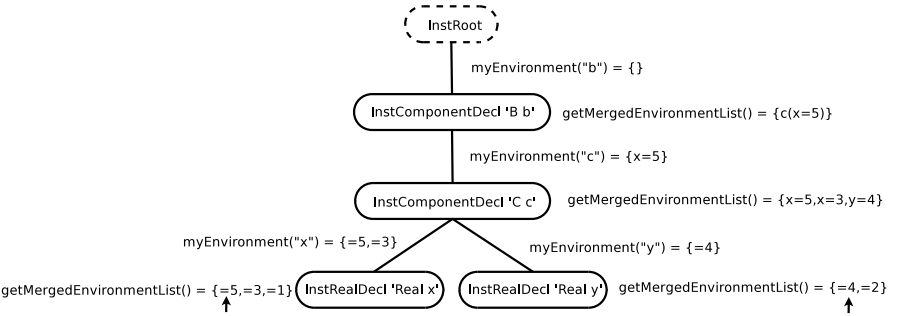


Figure 6.8 An example of how environments are computed.

stance node is constructed. In addition, the environment data structures are cached, since the corresponding attributes can be defined as lazy. This reduces the need for searching upwards the instance AST.

6.9 Handling of Structural Modifications

Structural modifications, corresponding in the PicoModelica language to component redeclarations, are challenging since they must be considered in the construction of the instance AST. In this respect, structural modifications are more difficult to handle than value modifications. However, the framework for merging and representation of environments introduced in Sections 6.7 and 6.8 is applicable also for structural modifications. In this section, it will be shown how the environment framework can be extended to take also structural modifications into account. This extension will be done in a modular fashion by adding equations to existing attributes and adding a small number of new attributes. The additions are made in a separate JastAdd aspect. The approach used in this section will also serve as an example of how the features of JastAdd can be used to create modular extensions of language semantics.

Conceptually, handling of structural modifications is straightforward. Before a child node, corresponding to a component declaration, of an instance node is built, the merged environment is checked for an applicable structural modification. Structural modifications are represented in environments by inter-AST references to `ComponentRedeclare` nodes in the source AST. If an applicable modification is found, a child instance node with a reference to the *replacing* `ComponentDecl` (source) node is created. Notice that this `ComponentDecl` resides in a modification subtree in the

```
refine InstantiationTree eq InstNode.getInstNodeList() {
  List l = new List();
  List comps = components();
  List supers = superClasses();

  // Add children corresponding to all components
  for (int i=0;i<comps.getNumChild();i++) {
    ComponentDecl cd = (ComponentDecl)comps.getChild(i);
①   ComponentDecl replacingCD =
          retrieveReplacingDecl(cd.getName().getID());

    if (replacingCD != null)
②     l.add(new InstReplacingComponentDecl(replacingCD,cd));
    else
③     l.add(cd.newInstComponent());
  }

  // Add children corresponding to all super classes
  for (int i=0;i<supers.getNumChild();i++) {
    l.add(((ExtendsClause)supers.getChild(i)).newInstExtends());
  }

  return l;
}
```

Figure 6.9 Definition of the attribute `getInstNodeList` taking the effect of component redeclarations into account. The new refined definition overrides the original definition which is specified in the aspect `InstanceTree`.

source AST. The construction of the instance AST then proceeds as in the previous case.

Now, a particular component declaration may match several structural modifications in the merged environment. This follows since it is valid to redeclare also replacing component declarations. The correct structural modification is identified simply by retrieving the first entry with a matching component name from the list which represents the merged environment. However, the modifications of *all* matching structural modifications, as well as of the original declaration must be considered. According to the Modelica specification, the modifications of replaced component declarations must be considered and merged into the environment, although the actual declaration is replaced. This means that the modifications of all structural modifications must be considered when merging environments.

The specific details of how to extend the basic environment framework introduced in Section 6.8 will now be discussed. First, a new instance AST

class is introduced:

```
InstReplacingComponent : InstComponent ::=
                         <OriginalDecl:ComponentDecl>;
```

This class represents a component declaration which is specified in a component redeclaration modification. In addition, a reference to the original declaration is stored, in order to provide access to its modification subtree. Notice that even though the original declaration is overridden by the replacing component declaration, its modification, if any, must be considered. This is achieved by adding an equation to `InstReplacingComponent` defining the attribute `modificationTree`:

```
eq InstReplacingComponent.modificationTree() =
           getOriginalDecl().getModification();
```

As noted above, it is valid to redeclare also a replacing component declaration. Therefore, there may be several redeclarations of the same component in an environment. It is straightforward to locate the correct replacing component declaration, by simply finding the first matching `ComponentRedeclare` in the environment. However, notice that each node `ComponentRedeclare`, also those which are themselves redeclared, may have a modification subtree which must be taken into account. This is done by introducing an additional equation for the synthesized attribute `matchModification(String name)` in the `ComponentRedeclare` node:

```
eq ComponentRedeclare.matchModification(String name) {
  if (name.equals(getName().getID()) &&
      getComponentDecl().hasModification())
    return getComponentDecl().getModification();
  return null;
}
```

Finally, it remains to revise the implementation of the attribute `InstNode.getInstNodeList` to also take structural modifications into account, see Listing 6.9. The main difference from the previous implementation is that the merged environment of an instance node is checked for structural modifications applicable to the declarations contained in the node, ①. If such a modification is found, an `InstReplacingComponentDecl` node is added to the set of children, ②. If no redeclaration modification was found, an `InstComponent` is created, ③. Notice that the new definition of `getInstNodeList` is specified using the `refine` keyword, which renders the original definition, which resides in the aspect `InstantiationTree`, to be overridden. Certainly, it would be possible to revise the original definition directly, but then the extension would not be fully modularized.

The revised implementation for computation of the set of children of an `InstNode` is supported by two new attributes. The synthesized attribute

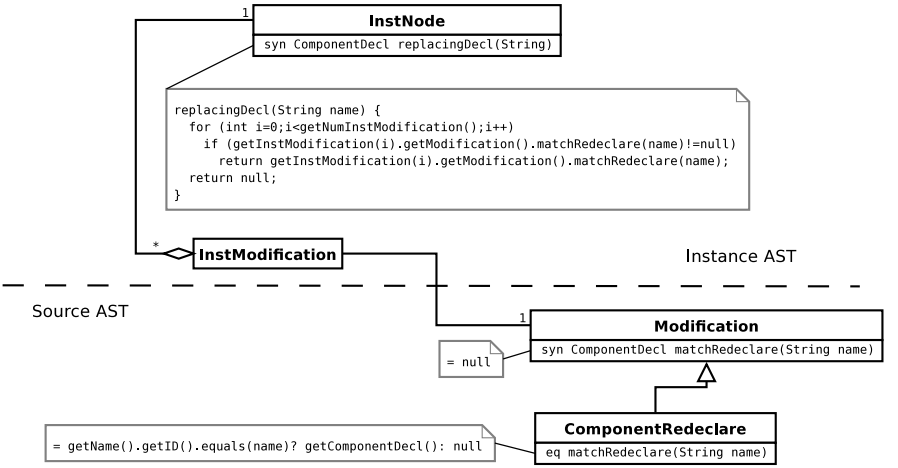


Figure 6.10 The attributes `replacingDecl` and `matchRedeclare` are used to retrieve structural modifications.

`replacingDecl(String name)`, which is defined for `InstNode`, retrieves the first structural modification matching a given component name, see Figure 6.10. An additional synthesized attribute, `matchRedeclare`, defined for the source AST node `Modification`, is introduced to retrieve the `ComponentDecl` from a component redeclaration node, see Figure 6.10.

EXAMPLE 6.4
The procedure of constructing the instance AST taking structural modifications into account will now be illustrated by the example in Listing 6.3. The resulting AST, as well as the environments associated with the nodes, are shown in Figure 6.11. First, the component declaration `D d` is flattened. Since the corresponding environment is empty, a new `InstComponent` is created. Next, the component declaration `B b` is considered. The environment of this component contains an applicable redeclaration, which is computed by the attribute `replacingDecl`. Accordingly, a node of type `InstReplacingComponent` is created. Construction of the AST proceeds recursively. Notice that the node `InstExtends 'B'` does not introduce a new name scope, and thus the modifiers in the environment are propagated unchanged. □

It is worth to notice that extension of the instance AST framework to handle structural modifications has been implemented solely by adding three equations to existing attributes and by introducing two new attributes. This illustrates how complex and context-dependent semantics can be im-

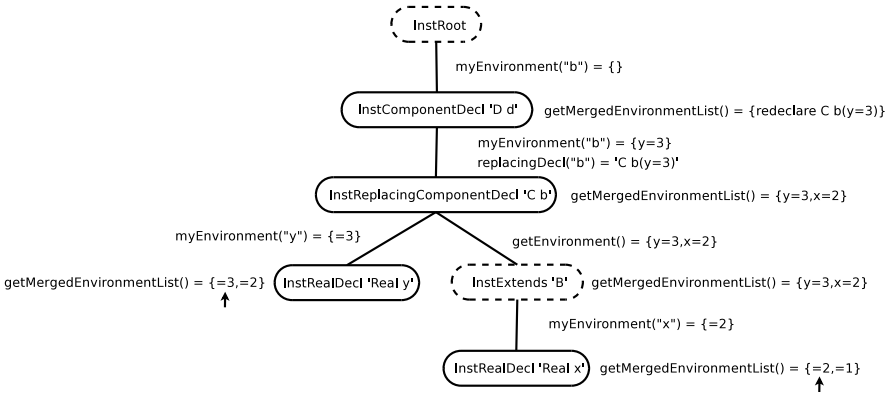


Figure 6.11 The instance AST resulting from the code example in Listing 6.2, and associated node environments.

plemented in a compact and modular fashion in JastAdd. Based on the experiences from implementing support for component redeclarations, no major difficulties are expected when implementing further extensions, for example support for type redeclarations.

6.10 A Flattening Algorithm

Given that the instance AST has been constructed, flattening of a model can be performed by simply traversing the instance AST and thereby collecting all equations and variables. The resulting model is represented by a new AST, referred to as the *flat* AST. In principle, it would be possible to use a subset of the source abstract grammar also to represent a flat model. However, since the flat representation constitutes an interface between the compiler front-end and different algorithms (symbolic and numerical), it is desirable to generate a new AST with a structure specifically designed for the purpose. Another reason for introducing a new abstract grammar for the flat model representation is that attributes defined for source AST classes may not be applicable in a flat context.

The flat abstract grammar consists in essence of a container class, which stores lists with references to variables and equations, respectively:

```
FClass ::= <Name:String> FVariable* FEquation*;
FVariable ::=  Name:FIdDecl [BindingExp:FExp];
FEquation ::= Left:FExp Right:FExp;
```

In addition, the flat abstract grammar contains node classes which dupli-

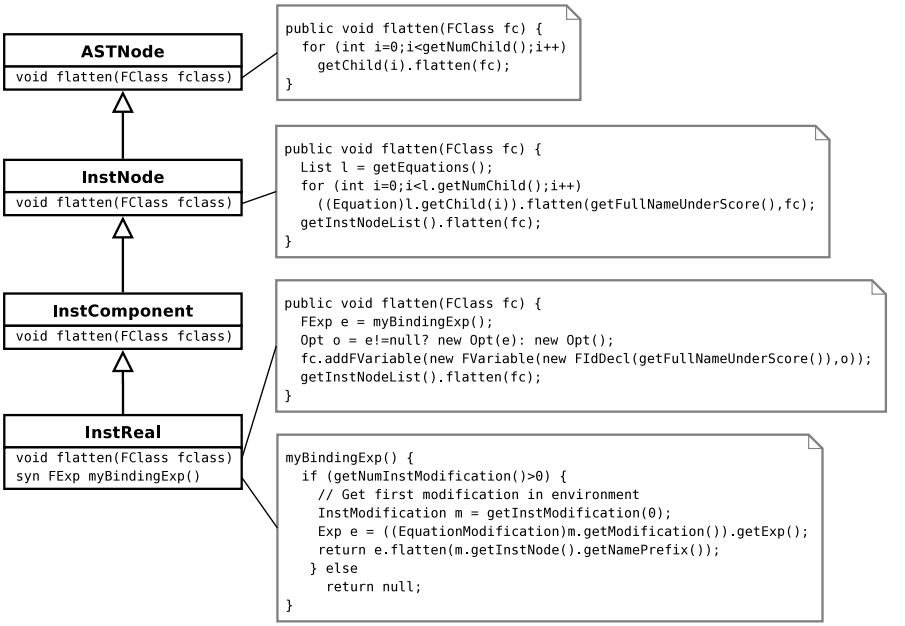


Figure 6.12 The implementation of the flattening algorithm.

cate the expression node classes of the source abstract grammar.

The design of the flattening algorithm is shown in Figure 6.12. A generic traversal method `flatten(FClass fc)` is defined for the generic node class `ASTNode`, which is the super class of all AST classes. The argument `FClass fc` represents the flat model. A specialization of the `flatten` method is defined for the node type `InstNode`, in which all equations are retrieved, flattened and added to the supplied `FClass`. A generic interface to the equations corresponding to an `InstNode` is provided by the attribute `equations`. This attribute is defined similarly to `components()`, see Figure 6.2.

Flattening of equations and expressions is straight forward–the flattened equation or expression is (almost) identical to the original one. One complication arises however. The name of a primitive variable is constructed by name mangling, as discussed above. Given the structure of the instance AST, computation of such names can easily be implemented by an inherited attribute. However, when an expression is flattened, this naming convention must be considered. This is done by providing a *name prefix* to the flattening method for equations and expressions. If an expression contains an identifier, the supplied name prefix is simply concatenated

with the name of the identifier.

A specialization of the flattening method is provided also for `InstReal`. In this case, a flat variable, represented by an `FVariable`, is added to the `FClass`. In addition, the environment of the primitive declaration must be checked for an applicable binding expression. The synthesized attribute `myBindingExp()`, which is defined for `InstReal`, computes the correct expression. Notice that the binding expression is flattened using the name prefix of the `InstNode` which was originally associated with the modification subtree. Although subtle, this point is essential, in order for binding expressions given in modifications to have the correct prefix. In fact, the main reason for including a reference to the `InstNode` of a modification in the `InstModification` class is to enable correct computation of the name prefix of expressions.

To summarize, the flattening algorithm is simple to implement, given that the instance AST, including merged environments, has been constructed. In essence, the flattening algorithm is implemented as a traversal of the instance AST, where all equations and primitive variables are collected.

6.11 Summary

In this chapter, the process of flattening a PicoModelica model has been discussed. In the flattening process, the most challenging problem is construction of the instance AST. The main difficulty is due to component redeclarations, which alter the structure of the AST. It has been shown how a framework for declarative construction of the instance AST can be encoded in JastAdd. The framework includes a strategy for merging of modifications and is based on nonterminal attributes. Further, it has been shown how the framework can be extended, in a fully modular fashion, to also support component redeclarations. Finally, a simple flattening algorithm, which constructs a flat model representation in the form of a new AST, has been presented. It is also worth noticing that due to the declarative specification of the instance AST, the construction is performed on demand, as the flattening algorithm executes.

7

Optimica

7.1 Introduction

Modelica is becoming a standard format for describing and communicating high-fidelity models of large-scale dynamic systems. Expert knowledge is being encoded into Modelica libraries, both in industry and in academia. The growing body of Modelica models also represents significant capital investments, and accordingly, Modelica models and libraries represent valuable assets for many companies. As a consequence, Modelica models are turning into legacy code, which cannot easily be replaced, simply because the cost of re-encoding the models in a different format is too large.

While the primary usage of Modelica models today is simulation, several other usages are emerging, as discussed in Chapter 4. Since it is not feasible, for the reasons mentioned above, to re-encode models for each new model usage, future Modelica tools, and also the Modelica language itself, should accommodate and promote new usages of Modelica models. This requirement has profound consequences for software design of Modelica tools, and also for the language design itself. In particular, some new usages may require new constructs, at the language level, in order to enable modeling of particular design problems.

One example of an emerging usage of Modelica models is dynamic optimization. A characteristic feature of realistic dynamic optimization problems is that the procedure of formulating such problems is highly iterative. It is common that extensive tuning of the cost function and constraints is required in order to obtain an acceptable solution, see for example the case study in Chapter 9. If a numerical algorithm is used to solve the dynamic optimization problem, there is an additional dimension that requires attention: the design of the transcription scheme. As described in Chapter 3, the scheme used to discretize the control and state variables

strongly influences the properties of the resulting solution, both in the case of simultaneous and sequential methods. For example, in the case of sequential methods, the choice of control parametrization quantifies the level of sub-optimality which is usually introduced by piece-wise polynomial control approximations. In the case of simultaneous methods, the choice of collocation scheme influences the accuracy of the solution, and it may even be required to adapt the mesh to a particular problem. In both cases, the choice of discretization method also affects the execution time for solving the problem, which is an important aspect in on-line applications. For these reasons, dynamic optimization problems are very rich in the sense that there are several aspects that require attention. Also, the user needs, and should be enabled to, model, using high-level language constructs, the optimization problem both in terms of cost functions and constraints and at the transcription level.

Sophisticated numerical optimization algorithms often have cumbersome APIs, which do not always match the engineering need for high-level description formats. For example, it is not uncommon for such numerical packages to be written in C, or in Fortran, and that they require the dynamic system to be modeled as an ODE/DAE, which is also encoded in C or Fortran. In addition, it may be required to also encode first and second order derivatives. Although there are efficient tools for automatic differentiation, as discussed in Chapter 3, encoding of dynamic optimization problems in low-level languages[1] like C or FORTRAN is often cumbersome and error-prone. An important goal of developing high-level languages for dynamic optimization is therefore to bridge the gap between the engineering need for high-level descriptions and the APIs of numerical algorithms.

There are several software packages supporting dynamic optimization, for example Dymola [Dynasim AB, 2007], gPROMS [Process Systems Enterprise, 2007], and GESOP [ASTOS Solutions GmbH, 2006]. However, most available software tools are restricted in the sense that they usually only support a particular optimization algorithm. While a particular algorithm may work well in some cases, the appropriate choice of numerical algorithm is usually dependent on the particular problem at hand. An analogy with differential equation solvers can be made. Stiff systems call for sophisticated, but potentially computationally demanding solvers, whereas less difficult systems may be more efficiently solved by a simpler algorithm. An additional goal in the development of tools supporting high-level formulation of dynamic optimization problems is therefore to provide an open architecture, were several different algorithms can be integrated.

In this chapter, an extension of Modelica, entitled Optimica, will be pre-

[1]The term low-level is relative, but is here used in relation to domain-specific languages like Modelica.

sented. Optimica consists of a number of new language elements, which enable high-level formulation of dynamic optimization problems based on Modelica models. The syntax as well as the semantics of Optimica will be described. In addition, a prototype implementation of an Optimica compiler, which is a modular extension of the JModelica compiler, will be presented. The effectiveness of Optimica and the prototype compiler will be illustrated by means of examples, and also by the case study in Chapter 9.

The chapter is organized as follows. In Section 7.2, issues related to extensions of languages are discussed. Different options regarding language extensions in Modelica are also treated. In Section 7.3, the scope of Optimica is discussed, i.e., the class of optimization problems that can be expressed using Optimica is defined. In Section 7.4 the syntax and the semantics of the Optimica extension are presented. Implementation issues related to the modular Optimica extension of the JModelica compiler are discussed in Section 7.5. In Section 7.6, some examples illustrating Optimica in practical use are given. The chapter ends with a summary and conclusions in Section 7.8.

7.2 Motivation of the Optimica Extension

Isn't Modelica Enough?

Although being a very rich language in terms of expressive power for describing complex hybrid dynamical systems, Modelica lacks important features desirable for expressing optimization problems. This is quite natural, since Modelica was not developed with optimization in mind. For example, the notion of cost functions, constraints, variable bounds and initial guesses are not included in the Modelica language. Some of these quantities may indeed be modeled using standard Modelica, to some extent. For example, a particular variable may be given the meaning of cost, and the `min` and `max` attributes may be interpreted as variable bounds. However, while this approach may work in simple cases, it becomes intractable for more complex optimization problems. For example, complicated constraints, several use cases, and tailoring of the transcription method would be difficult to express. Another example where it is inconvenient to use standard Modelica to model an optimization problem is variable bounds. Again, the `min` and `max` attributes may be used for the purpose. But these attributes are usually used to express regions of validity for a model, and giving them a new semantic meaning would be potentially misleading.

What About Annotations?

Modelica offers a mechanism for adding information to model, which may not be part of the actual mathematical description, but which is convenient to store in the model. Typical examples include graphical annotations and documentation. Annotations can also be used to supply information that can be used by a particular tool, for example, in order to influence properties of the translation process. In principle, it would be possible to specify parts of an optimization problem by introducing suitable annotations. For example, a variable could be marked as a cost function, and the semantic meaning of the equality operator in an equation could be changed to that of the inequality operator. There are two reasons why it is not a good idea to strictly use this approach. Firstly, and most importantly, annotations are designed to supply *complementary* information, whereas in this case, the elements of an optimization problem are rather *primary* information, that is essential for solving the actual problem. Also, since annotations are not intended for formulation of design problems, they do not provide a convenient modeling environment for the user. Secondly, annotations cannot currently be changed by means of modification. Since modification is one of the corner-stones of Modelica, this is a severe restriction. Also, it is not currently well defined how annotations are treated in the case of inheritance. Since one of the main objectives of the Optimica extension is to enable convenient formulation of dynamic optimization problems using high-level constructs, using only annotations does not seem to be a feasible alternative.

Whereas the above arguments are applicable to core elements of an optimization problem, such as cost function and constraints, annotations may well be used to specify a *solution algorithm*, and associated parameters. This type of information is not part of the actual optimization formulation, but it might still be essential in order to efficiently solve the problem numerically. By introducing annotations for specifying, for example, the collocation scheme used in a direct method, the user is able to model both the actual optimization problem at hand and the transcription method in a unified high-level description language. This approach is also in line with the intentions of Modelica annotations, because of the separation between formulation of the actual problem (by means of dedicated language constructs), and specification of the solution technique (by means of annotations).

Tool-oriented Support for Optimization?

Another potential strategy for enabling dynamic optimization of Modelica models is to develop tool-oriented solutions, for example Graphical User Interfaces (GUIs), within a simulation-based software tool. This approach

is used, for example, to enable optimization of Modelica models in Dymola, see Section 3.4. The user would then set up the optimization problem by entering information in dedicated fields in the GUI. Using this approach, the software tool needs to maintain an internal model of the optimization problem, as specified by the user. While this solution may be an attractive choice for interfacing a particular optimization method with existing simulation-based tools, it does not offer the flexibility, or portability, which is inherent in the Modelica language. It is therefore desirable to define, at the language level, a generic extension, which has a well defined syntax and semantics. Nevertheless, it may still be desirable to offer GUIs, in order to increase productivity in the design process, in the same way as current Modelica tools typically offer GUIs to simplify critical modeling tasks

To Extend or to Complement?

A key issue is whether to extend Modelica by introducing new language constructs, or to define a new, separate, language which complements Modelica. By introducing a new language, the syntax and semantics of Modelica would be kept entirely intact, which may be advantageous since it makes design and maintenance of the language simpler. Also, if several extensions are introduced, defining the interaction between the extensions, both at a syntactic and semantic level, may be difficult. On the other hand, Modelica has many generic built-in constructs, e.g., classes, functions and declarative equations, which are widely applicable in many contexts. Reinventing such constructs in new languages does not seem to be an attractive alternative. Another argument in favor of language extension is that Modelica offers strong support for modularization of models. In the case of dynamic optimization, the user may construct the model separately from the formulation of the optimization problem, in which the model is used. In this way, the same model may still be used for other purposes than optimization, such as, for example, simulation.

It is essential, however, that language extensions targeted at particular usages of Modelica models do not interfere unnecessarily with the original language. Preferably, extensions should be *modular*, in the sense that the new constructs are only allowed in a well defined language environment.

7.3 Scope of Optimica

Information Structure

In order to formulate a dynamic optimization problem, to be solved by a numerical algorithm, the user must supply different kinds of information.

It is natural to categorize this information into three levels, corresponding to increasing levels of detail.

- **Level I.** At the mathematical level, a canonical formulation of a dynamic optimization problem is given. This include variables and parameters to optimize, cost function to minimize, constraints, and the Modelica model constituting the dynamic constraint. The optimization problem formulated at this level is in general infinite dimensional, and is thereby only partial in the respect that it cannot be directly used by a numerical algorithm without additional information, for example, concerning transcription of continuous variables.
- **Level II.** At the transcription level, a method for translating the problem from an infinite dimensional problem to a finite dimensional problem needs to be provided. This might include discretization meshes as well as initial guesses for optimization parameters and variables. It should be noticed that the information required at this level is dependent on the numerical algorithm that is used to solve the problem.
- **Level III.** At the algorithm level, information such as tolerances and algorithm control parameters may be given. Such parameters are often critical in order to achieve acceptable performance in terms of convergence, numerical reliability, and speed.

An important issue to address is whether information associated with all levels should be given in the language extension. In Modelica, only information corresponding to Level I is expressed in the actual model description. Existing Modelica tools then typically use automatic algorithms for critical tasks such as state selection and calculation of consistent initial conditions, although the algorithms can be influenced by the user via the Modelica code, by means of annotations, or attributes, such as `StateSelect`. Yet other information, such as choice of solver, tolerances and simulation horizon is provided directly to the tool, either by means of a graphical user interface, a script language, or alternatively, in annotations.

For dynamic optimization, the situation is similar, but the need for user input at the algorithm level is more emphasized. Automatic algorithms, for example for mesh selection, exist, but may not be suitable for all kinds of problems. It is therefore desirable to include, in the language, means for the user to specify most aspects of the problem in order to maintain flexibility, while allowing for automatic algorithms to be used when possible and suitable.

Relating to the three levels described above, the approach taken in the design of Optimica is to extend the Modelica language with a few new

language constructs corresponding to the elements of the mathematical description of the optimization problem (level I). The information included in levels II and III, however, may rather be specified by means of annotations. This design is also consistent with the current use of Modelica and annotations, in that the actual modeling/problem formulation is done using dedicated language constructs, whereas solution algorithms and their parameters are specified using annotations.

Dynamic System Model

The scope of Optimica can be separated into two parts. The first part is concerned with the class of models that can be described in Modelica. Arguably, this class is large, since very complex, non-linear and hybrid behavior can be encoded in Modelica. From a dynamic optimization perspective, the inherent complexity of Modelica models is a major challenge. Typically, different algorithms for dynamic optimization support different model structures. In fact, the key to developing efficient algorithms lies in exploiting the structure of the model being optimized. Consequently, there are different algorithms for different model structures, such as linear systems, non-linear ODEs, general DAEs, and hybrid systems. In general, an algorithm can be expected to have better performance, in terms of convergence properties and shorter execution times, if the model structure can be exploited. For example, if the model is linear, and the cost function is quadratic, the problem can be obtained very efficiently by solving a Riccati equation. On the other hand, optimization of general non-linear and hybrid DAEs is still an area of active research, see for example [Barton and Lee, 2002]. As a result, the structure of the model highly affects the applicability of different algorithms. The Optimica compiler presented in this chapter relies on a direct collocation algorithm in order to demonstrate the proposed concept. Accordingly, the restrictions imposed on model structure by this algorithm apply when formulating the Modelica model, upon which the optimization problem is based. For example, this excludes the use of hybrid constructs, since the right hand side of the dynamics is assumed to be twice continuously differentiable. Obviously, this restriction excludes optimization of many realistic Modelica models. On the other hand, in some cases, reformulation of discontinuities to smooth approximations may be possible in order to enable efficient optimization. This is particularly important in on-line applications. The Optimica extension, as presented in this chapter, could also be extended to support other algorithms, which are indeed applicable to a larger class of models.

The Dynamic Optimization Problem

The second part of the scope of Optimica is concerned with the remaining elements of the optimization problem. This includes cost functions,

constraints and variable bounds. Consider the following formulation of a dynamic optimization problem:

$$\min_{u(t),p} \psi(x(t_i), y(t_i), u(t_i), p), \quad i \in 1 \ldots N_{cost}, \quad t_i \in [t_0, t_f] \tag{7.1}$$

subject to the dynamic system

$$F(\dot{x}(t), x(t), y(t), u(t), p, t) = 0, \quad t \in [t_0, t_f] \tag{7.2}$$

and the constraints

$$c_{ineq}(x(t), y(t), u(t), p) \leq 0 \quad t \in [t_0, t_f] \tag{7.3}$$
$$c_{eq}(x(t), y(t), u(t), p) = 0 \quad t \in [t_0, t_f] \tag{7.4}$$
$$c^p_{ineq}(x(t_j), y(t_j), u(t_j), p) \leq 0, \quad j \in 1 \ldots N_{ineq}, \quad t_j \in [t_0, t_f] \tag{7.5}$$
$$c^p_{eq}(x(t_k), y(t_k), u(t_k), p) = 0, \quad k \in 1 \ldots N_{eq}, \quad t_k \in [t_0, t_f] \tag{7.6}$$

where $x(t) \in R^{n_x}$ are the dynamic variables, $y(t) \in R^{n_y}$ are the algebraic variables, $u(t) \in R^{n_u}$ are the control inputs, and $p \in R^{n_p}$ are parameters which are free in the optimization. In addition, the optimization is performed on the interval $t \in [t_0, t_f]$, where t_0 and t_f can be fixed or free, respectively. In addition, the initial values of the dynamic and algebraic variables may be fixed or free in the optimization.

The constraints include inequality and equality path constraints, (7.3)-(7.4). In addition, inequality and equality point constraints, (7.5)-(7.6), are supported. Point constraints are typically used to express initial or terminal constraints, but can also be used to specify constraints for time points in the interior of the interval.

The cost function (7.1) is a generalization of a terminal cost function, $\phi(t_f)$, in that it admits inclusion of variable values at other time instants. This form includes some of the most commonly used cost function formulations. Obviously, terminal as well as initial costs are included. A Lagrange cost function can be obtained by introducing an additional state variable, $x_L(t)$, with the associated differential equation $\dot{x}_L(t) = L(x(t), u(t))$, and the cost function $\psi(t_f) = x_L(t_f)$. The need to include variable values at discrete points in the interior of the optimization interval in the cost function arises for example in parameter estimation problems. In such cases, a sequence of measurements, $y_d(t_i)$, obtained at the sampling instants t_i, $i \in 1 \ldots N_d$ is typically available. A cost function candidate is then:

$$\sum_{i=1}^{N_d} (y(t_i) - y_d(t_i))^T W (y(t_i) - y_d(t_i)) \tag{7.7}$$

where $y(t_i)$ is the model response at time t_i and W is a weighting matrix.

Another important class of problems is static optimization problems on the form:

$$\begin{aligned} &\min_{u,p} \varphi(x,y,u,p) \\ &\text{subject to} \\ &F(0,x,y,u,p,t_s) = 0 \\ &c_{ineq}(x,u,p) \leq 0 \\ &c_{eq}(x,u,p) = 0 \end{aligned} \tag{7.8}$$

In this case, a static optimization problem is derived from a, potentially, dynamic Modelica model by setting all derivatives to zero. Since the problem is static, all variables are algebraic in this case, and no transcription procedure is necessary. The variable t_s denotes the time instant at which the static optimization problem is defined.

Transcription

In this chapter a direct collocation method will be used to illustrate how also the transcription step can be encoded in the Optimica extension. The information that needs to be provided by the user is then a mesh specification, the collocation points, and the coefficients of the interpolation polynomials.

7.4 The Optimica Extension

Modelica is a very rich language, containing many general-purpose constructs. Therefore, an extension of Modelica to accommodate dynamic optimization problems requires only a small number of new constructs. In summary, the Optimica extension consists of the following elements:

- A new specialized class: `optimization`
- New attributes for the built-in type Real: `free` and `initialGuess`.
- A new function for accessing the value of a variable at a specified time instant
- Class attributes for the specialized class `optimization`: `objective`, `startTime`, `finalTime` and `static`
- A new section: `constraint`
- Inequality constraints
- An annotation for providing transcription information

```
model DoubleIntegrator
  Real x(start=0);
  Real v(start=0);
  input Real u;
equation
  der(x)=v;
  der(v)=u;
end DoubleIntegrator;
```

Listing 7.1 A Modelica model of a double integrator system.

In this section, the Optimica extension will be presented and informally defined. The presentation will be made using the following dynamic optimization problem, based on a double integrator system, as an example:

$$\min_{u(t)} \int_0^{t_f} 1\, dt \tag{7.9}$$

subject to the dynamic constraint

$$\begin{aligned} \dot{x}(t) &= v(t), \quad x(0) = 0 \\ \dot{v}(t) &= u(t), \quad v(0) = 0 \end{aligned} \tag{7.10}$$

and

$$\begin{aligned} x(t_f) &= 1 \\ v(t_f) &= 0 \\ v(t) &\leq 0.5 \\ -1 \leq u(t) &\leq 1 \end{aligned} \tag{7.11}$$

In this problem, the final time, t_f, is free, and the objective is thus to minimize the time it takes to transfer the state of the double integrator from the point $(0,0)$ to $(1,0)$, while respecting bounds on the velocity $v(t)$ and the input $u(t)$. A Modelica model for the double integrator system is shown in Listing 7.1.

A New Specialized Class

It is convenient to introduce a new specialized class, called `optimization`, in which the proposed Optimica-specific constructs are valid. This approach is consistent with the Modelica language, since there are already several other specialized classes, e.g., `record`, `function` and `model`. By introducing a new specialized class, it also becomes straightforward to check the validity of a program, since the Optimica-specific constructs

are only valid inside an `optimization` class. The `optimization` class corresponds to an optimization problem, static or dynamic, as specified in Section 7.3. Apart from the Optimica-specific constructs, an `optimization` class can contain also component and variable declarations, local classes, and equations.

It is not possible to declare components from `optimization` classes in the current version of Optimica. Rather, the underlying assumption is that an `optimization` class defines an optimization problem, that is solved off-line. An interesting extension would, however, be to allow for `optimization` classes to be instantiated. With this extension, it would be possible to solve optimization problems, on-line, during simulation. A particularly interesting application of this feature is model predictive control, which is a control strategy that involves on-line solution of optimization problems during execution.

As a starting-point for the formulation of the optimization problem (7.9)-(7.11), consider the `optimization` class:

```
optimization DIMinTime
  DoubleIntegrator di;
end DIMinTime;
```

This class contains only one component representing the dynamic system model, but will be extended in the following to incorporate also the other elements of the optimization problem.

Attributes for the Built-in Type Real

In order to superimpose information on variable declarations, two new attributes are introduced for the built-in type Real[2]. Firstly, it should be possible to specify that a variable, or parameter, is free in the optimization. Modelica parameters are normally considered to be fixed after the initialization step, but in the case of optimization, some parameters may rather be considered to be free. In optimal control formulations, the control inputs should be marked as free, to indicate that they are indeed optimization variables. For these reasons, a new attribute for the built-in type Real, `free`, of boolean type is introduced. By default, this attribute is set to `false`.

Secondly, an attribute, `initialGuess`, is introduced to enable the user to provide an initial guess for variables and parameters. In the case of free optimization parameters, the `initialGuess` attribute provides an initial guess to the optimization algorithm for the corresponding parameter. In the case of variables, the `initialGuess` attribute is used to provide the numerical solver with an initial guess for the entire optimization interval.

[2]The same attributes may be introduced for the built-in type Integer, in order to support also variables of type Integer in the optimization formulation

This is particularly important if a simultaneous or multiple-shooting algorithm is used, since these algorithms introduce optimization variables corresponding to the values of variables at discrete points over the interval. Notice that such initial guesses may be needed both for control and state variables. For variables, however, this strategy for providing initial guesses may sometimes be inadequate. In such cases, a better solution is to use simulation data to initialize the optimization problem. This approach is also supported by the Optimica compiler. In the double integrator example, the control variable u is a free optimization variable, and accordingly, the `free` attribute is set to `true`. Also, the `initialGuess` attribute is set to 0.0.

```
optimization DIMinTime
  DoubleIntegrator di(u(free=true,initialGuess=0.0));
end DIMinTime;
```

A Function for Accessing Instant Values of a Variable

An important component of some dynamic optimization problems, in particular parameter estimation problems where measurement data is available, is variable access at discrete time instants. For example, if a measurement data value, y_i, has been obtained at time t_i, it may be desirable to penalize the deviation between y_i and a corresponding variable in the model, evaluated at the time instant t_i. In Modelica, it is not possible to access the value of a variable at a particular time instant in a natural way, and a new construct therefore has to be introduced.

All variables in Modelica are functions of time. The variability of variables may be different—some are continuously changing, whereas others can change value only at discrete time instants, and yet others are constant. Nevertheless, the value of a Modelica variable is defined for all time instants within the simulation, or optimization, interval. The time argument of variables are not written explicitly in Modelica, however. One option for enabling access to variable values at specified time instants is therefore to associate an implicitly defined function with a variable declaration. This function can then be invoked by the standard Modelica syntax for function calls, `y(t_i)`. The name of the function is identical to the name of the variable, and it has one argument; the time instant at which the variable is evaluated. This syntax is also very natural since it corresponds precisely to the mathematical notation of a function. Notice that the proposed syntax `y(t_i)` makes the interpretation of such an expression context dependent. In order for this construct to be valid in standard Modelica, `y` must refer to a function declaration. With the proposed extension, `y` may refer either to a function declaration or a variable declaration. A compiler therefore needs to classify an expression `y(t_i)`

based on the context, i.e., what function and variable declarations are visible. An alternative syntax would have been to introduce a new built-in function, that returns the value of a variable at a specified time instant. While this alternative would have been straightforward to implement, the proposed syntax has the advantages of being easier to read and that it more closely resembles the corresponding mathematical notation.

Class Attributes

In the optimization formulations (7.1)-(7.6) and (7.8), there are elements that occur only once, i.e., the cost function and the optimization interval in (7.1)-(7.6), and in the static case (7.8), only the cost function. These elements are intrinsic properties of the respective optimization formulations, and should be specified, once, by the user. In this respect the cost function and optimization interval differ from, for example, constraints, since the user may specify zero, one or more of the latter.

One option for providing this kind of information is to introduce a built-in class, call it `Optimization`, and require that all `optimization` classes inherit from `Optimization`. Information about the cost function and optimization interval may then be given as modifications of components in this built-in class:

```
optimization DIMinTime
  extends Optimization(objective=cost(finalTime),
                       startTime=0,
                       finalTime(free=true,initialGuess=1));
  Real cost;
  DoubleIntegrator di(u(free=true,initialGuess=0.0));
equation
  der(cost) = 1;
end DIMinTime;
```

Here, `objective`, `startTime` and `finalTime` are assumed to be components located in `Optimization`, whereas `cost` is a variable which is looked up in the scope of the `optimization` class itself. Notice also how the cost function, `cost`, has been introduced, and that the `finalTime` attribute is specified to be free in the optimization. This approach of inheriting from a built-in class has been used previously, in the tool Mosilab [Nytsch-Geusen, 2007], where the Modelica language is extended to support statecharts. In the statechart extension, a new specialized class, `state`, is introduced, and properties of a `state` class (for example whether the state is an initial state) can be specified by inheriting from the built-in class `State` and applying suitable modifications.

The main drawback of this approach is its lack of clarity. In particular, it is not immediately clear that `Optimization` is a built-in class, and that its contained elements represent intrinsic properties of the `optimization`

class, rather than regular elements, as in the case of inheritance from user or library classes.

To remedy this deficiency, the notion of *class attributes* is proposed. This idea is not new, but has been discussed previously within the Modelica community. A class attribute is an intrinsic element of a specialized class, and may be modified in a class declaration without the need to explicitly extend from a built-in class. In the Optimica extension, four class attributes are introduced for the specialized class `optimization`. These are `objective`, which defines the cost function, `startTime`, which defines the start of the optimization interval, `finalTime`, which defines the end of the optimization interval, and `static`, which indicates whether the class defines a static or dynamic optimization problem. The proposed syntax for class attributes is shown in the following `optimization` class:

```
optimization DIMinTime (objective=cost(finalTime),
                        startTime=0,
                        finalTime(free=true,initialGuess=1))
  Real cost;
  DoubleIntegrator di(u(free=true,initialGuess=0.0));
equation
  der(cost) = 1;
end DIMinTime;
```

The default value of the class attribute `static` is `false`, and accordingly, it does not have to be set in this case. In essence, the keyword `extends` and the reference to the built-in class have been eliminated, and the modification construct is instead given directly after the name of the class itself. The class attributes may be accessed and modified in the same way as if they were inherited.

Constraints

Constraints are similar to equations, and in fact, a path equality constraint is equivalent to a Modelica equation. But in addition, inequality constraints, as well as point equality and inequality constraints should be supported. It is therefore natural to have a separation between equations and constraints. In Modelica, initial equations, equations, and algorithms are specified in separate sections, within a class body. A reasonable alternative for specifying constraints is therefore to introduce a new kind of section, `constraint`. Constraint sections are only allowed inside an `optimization` class, and may contain equality, inequality as well as point constraints. In the double integrator example, there are several constraints. Apart from the constraints specifying bounds on the control input u and the velocity v, there are also terminal constraints. The latter are conveniently expressed using the mechanism for accessing the

value of a variable at a particular time instant; `di.x(finalTime)=1` and `di.v(finalTime)=0`. In addition, bounds may have to be specified for the `finalTime` class attribute. The resulting optimization formulation may now be written:

```
optimization DIMinTime (objective=cost(finalTime),
                        startTime=0,
                        finalTime(free=true,initialGuess=1))
  Real cost;
  DoubleIntegrator di(u(free=true,initialGuess=0.0));
equation
  der(cost) = 1;
constraint
  finalTime>=0.5;
  finalTime<=10;
  di.x(finalTime)=1;
  di.v(finalTime)=0;
  di.v<=0.5;
  di.u>=-1; di.u<=1;
end DIMinTime;
```

Annotations for Specification of the Transcription Scheme

The transcription scheme used to transform the infinite-dimensional dynamic optimization problem into a finite-dimensional approximate problem usually influences the properties of the numerical solution. Nevertheless, transcription information can be considered to be complimentary information, that is not part of the mathematical definition of the optimization problem itself. Also, transcription information is closely related to particular numerical algorithms. It is therefore reasonable not to introduce new language constructs, but rather new annotations for specification of transcription schemes. This solution is also more flexible, which is important in order easily accommodate transcription schemes corresponding to algorithms other than the direct collocation method currently supported.

Following the guidelines for vendor-specific annotations in the specification of Modelica 3.0 [The Modelica Association, 2007a, p. 147], a hierarchical annotation for supplying the information needed to specify a direct collocation method based on interpolation polynomials has been introduced. This annotation is defined by the following Modelica `record`:

```
record DirectCollocationInterpolationPolynomials
  parameter Real mesh[:];
  parameter Real collocationPoints[:];
  parameter Real polynomialCoefficientsAlgebraic[:];
  parameter Real polynomialCoefficientsDynamic[:];
end DirectCollocationInterpolationPolynomials;
```

The variable mesh contains the lengths of the collocation elements. The number of elements is given implicitly by the length of the vector. The practice of specifying the element lengths rather than the location of the start times of the elements is common in direct collocation formulations. The variable collocationPoints is used to specify the location of the collocation points within a normalized element, i.e., the values in the vector collocationPoints should be in the interval $[0, 1]$. As described in Chapter 3, the degree of the polynomials used to approximate the dynamic variables is usually one higher than the degree of the interpolation polynomials for the algebraic variables. Therefore, the coefficients of the interpolation polynomials used to approximate the dynamic and algebraic variables are given separately.

A transcription scheme based on third order Lagrange polynomials and Radau points is specified in Listing 7.2. In this example, the parameters for specifying the transcription scheme are declared directly in the body of the optimization class. A more convenient approach would be to organize the parameters of this particular scheme, and others, into records. These records, and in addition, functions for calculation of collocation points and polynomial coefficients, could then be organized into a Modelica library, which would enable easy access for users. A convenient method for inclusion of a transcription scheme in an optimization class would then be to inherit from a library class containing both the parameters specifying scheme, and the actual annotation statement. However, this method is complicated by the fact that the semantics of how annotations are inherited is not clearly stated in the Modelica specification. In addition, it would be desirable to enable modification of elements in annotations, in order to increase the flexibility of annotations.

7.5 The Optimica Compiler

A prototype implementation of the JModelica compiler, that also supports the Optimica extension has been developed. The extended compiler will be referred to as the Optimica compiler in the following. In terms of the front-end, the compiler supports a subset of Modelica, as described in Chapter 4, and an early version of Optimica. The syntax of Optimica that is supported by this compiler is different than the one presented in this chapter, although the functionality is essentially the same. The new, improved syntax and semantics that have been presented in this chapter, were defined based on the comments and experiences from the users of the very first version of Optimica. A new version of the Optimica compiler, supporting the revised Optimica syntax is currently under development, with the intention of replacing the initial prototype.

```
optimization DIMinTime (objective=cost(finalTime),
                        startTime=0,
                        finalTime(free=true,initialGuess=1))

  Real cost;
  DoubleIntegrator di(u(free=true,initialGuess=0.0));

  parameter Integer N = 100;
  parameter Real m[N] = ones(N)*(finalTime-startTime)/N;
  parameter Real c_p[3] = {0.1550,
                           0.6449,
                           1.0000};
  parameter Real p_c_a[3,3] = [2.4158, -3.9739, 1.5581;
                               -5.7491, 6.6406, -0.8914;
                               3.3333, -2.6667, 0.3333];
  parameter Real p_c_d[4,4] = [-10, 18, -9, 1;
                               15.5808, -25.6296, 10.0488, 0;
                               -8.9141, 10.2963, -1.3821, 0;
                               3.3333, -2.6667, 0.3333, 0];
  annotation(__Optimica(DirectCollocationInterpolationPolynomials(
                        mesh=m,
                        collocationPoints=c_p,
                        polynomialCoefficientsAlgebraic=p_c_a,
                        polynomialCoefficientsDynamic=p_c_d)));
equation
  der(cost) = 1;
constraint
  finalTime>=0.5;
  finalTime<=10;
  di.x(finalTime)=1;
  di.v(finalTime)=0;
  di.v<=0.5;
  di.u>=-1; di.u<=1;
end DIMinTime;
```

Listing 7.2 The complete Optimica specification of the double integrator optimal control problem.

The main functionality of the Optimica compiler (both the prototype and the new version) is to translate Modelica/Optimica source code into AMPL, see [Fourer *et al.*, 2003], source code. In addition, the Optimica compiler can read simulation data from file, in order to obtain an initial guess for the optimization problem. The AMPL code may then be executed in the AMPL tool, which renders the problem to be solved by a numerical algorithm, such as IPOPT [Wächter and Biegler, 2006]. The result is then

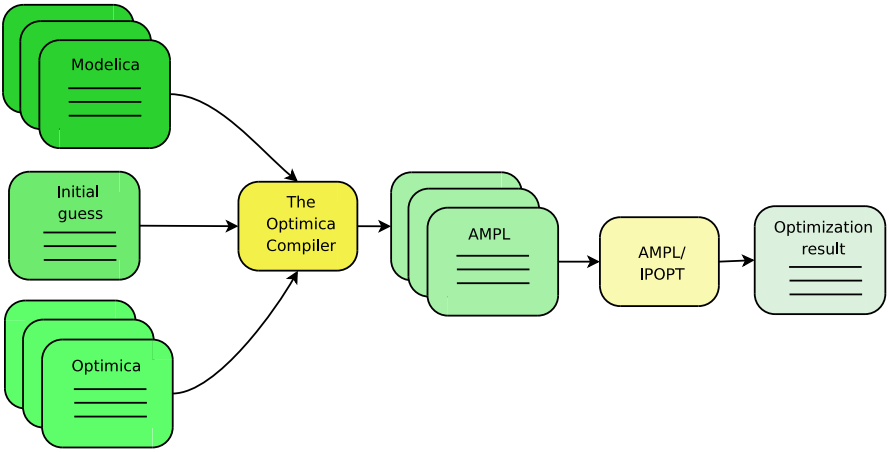


Figure 7.1 The transformation from Modelica/Optimica code to optimization result.

written to file for further analysis or implementation. See Figure 7.1 for an illustration of the transformation steps involved when using the Optimica compiler and AMPL to solve a dynamic optimization problem.

Relating to Figure 4.1, extension of a compiler involves addition of functionality at three levels. Firstly, the front-end of the compiler has to be extended to accommodate the new syntactic and semantic constructs. Secondly, the canonical data structure representing the mathematical problem formulation has to be extended, in order to accommodate for the new elements. Thirdly, if the target of the extension is a new algorithm, then a corresponding back-end has to be developed. The major challenges lie in the extension of the front-end and of the canonical data structures, in particular since an important objective is to implement the extension modularly.

The Optimica Compiler Front-end

The front-end of the new version of the Optimica compiler has been implemented as a fully modularized extension of the JModelica compiler. As in the case of the extension of the JModelica compiler to support some constructs of the MetaModelica language, see Section 4.9, the implementation of the front-end extension can be organized into three parts. The first part is concerned with extending the parser, so that the new syntactic constructs are supported. In the case of Optimica, there are two new syntactic constructs, namely the specialized class `optimization`, with the associated class attribute modification clause, and the `constraint` section.

Productions corresponding to these constructs were encoded in the format supported by JastAdd's parser preprocessor, and the resulting parser supports both the subset of Modelica that is supported by the JModelica compiler, and the Optimica extension.

The second part is concerned with addition of new AST classes, representing the new constructs. Accordingly, AST classes corresponding to the specialized class `optimization`, constraints and the new instant value function were defined.

The third part, which is also the most challenging, consists of extending the functionality of the compiler front-end so that a flat representation of the optimization problem corresponding to an Optimica description can be obtained. Firstly, the name analysis framework must be extended. In particular, support for class attributes should be implemented. In addition, the compiler should recognize the new attributes for the built-in type Real. Secondly, expressions that are syntactically equivalent to function calls should be classified as either being an ordinary function call, or a call to the new instant value function, which is implicitly defined for variable declarations. Thirdly, the flattening algorithm must be extended to take class attributes and constraints into account.

These extensions are conveniently implemented in JastAdd as new attributes, or new equations to attributes already defined in the JModelica compiler. The new attributes and equation declarations are defined in separate aspects, which are then included in the weaving procedure. The classification of function calls was implemented using a strategy similar to that used for classification of ambiguous names, see Section 5.2. Extension of the flattening algorithm is straightforward, and consists mainly of adding new attributes to enable access to the constraints from the instance AST.

Canonical Representation and API

The canonical data structure used in the JModelica is referred to as flat Modelica. Also, the result of the flattening procedure in the JModelica compiler is a model representation expressed in this format. Optimica, however, supports additional elements, which must also be represented by the canonical data structure. Therefore, an extension of flat Modelica, flat Optimica, has been defined. Like in the case of flat Modelica, flat Optimica is defined as an abstract grammar and is represented in the compiler by an AST. In essence, flat Optimica contains, apart from variable declarations and equations, also constraints, the optimization interval, the cost function, and transcription information. Flat Optimica also provides an API for accessing the contained elements, which is useful for code generation purposes.

Code Generation to AMPL

One of the main features of the Optimica compiler is that it performs automatic transcription of continuous variables, using a direct collocation method, see Chapter 3. The user is thus relieved from the burden of encoding the collocation equations, which is a tedious and error-prone procedure. Whereas the prototype version of the Optimica compiler supported one particular collocation scheme, future versions will support the annotation that was introduced above to specify the transcription method.

In order to solve the transcribed optimization problem by means of a numerical algorithm, the Optimica compiler generates AMPL code, see Chapter 3. The transcribed problem is purely static, and can therefore be encoded using the constructs available in AMPL. The AMPL representation of the optimization problem can be viewed as an additional intermediate representation format. The purpose of using AMPL is twofold. Firstly, AMPL provides an additional debugging level, that is very useful during compiler development. In particular, the AMPL tool offers a shell, where variables and constraints can be inspected. Secondly, the AMPL solver interface provides solvers with sparsity information, as well as first and second order derivatives. This information may be essential for performance and convergence of a numerical optimization algorithm. However, development of such an interface in the Optimica compiler requires a major coding effort, and is beyond the scope of this thesis. The numerical algorithm IPOPT has been used to solve the non-linear program resulting from the transcription procedure. For additional details on IPOPT, see Chapter 3.

Implementation Status

A front-end for the new, improved version of Optimica has been implemented, as a modular extension of the JModelica compiler. Support for all the suggested constructs in the Optimica extension have been implemented, including class attributes, the new attributes for the built-in class Real, the instant value function for variables and constraints. The compiler front-end produces as its output a flat representation of an Optimica optimization problem. Future extensions include support for the annotations dedicated to specification of the transcription scheme, and adaption of the AMPL code generation back-end of the prototype compiler to the new compiler version.

7.6 Examples

In Section 7.4, the syntax and semantics of Optimica were illustrated by means of an optimal control problem. In this section, the features of Opti-

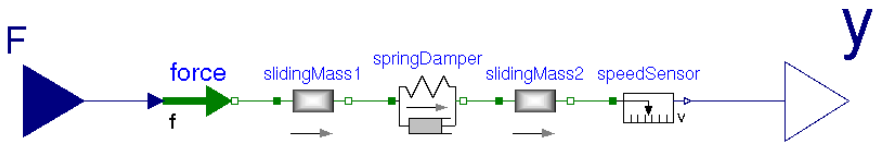


Figure 7.2 A Modelica model of a mechanical servo.

```
model Servo
  Modelica.Mechanics.Translational.SlidingMass slidingMass1(m=m1);
  Modelica.Mechanics.Translational.SlidingMass slidingMass2(m=m2);
  Modelica.Mechanics.Translational.Force force;
  Modelica.Blocks.Interfaces.RealInput F;
  Modelica.Mechanics.Translational.SpringDamper
    springDamper(c=c, d=d,s_rel(min=-2000)) ;
  Modelica.Blocks.Interfaces.RealOutput y;
  parameter Real m1 = 1;
  parameter Real m2 = 1;
  parameter Real d = 0.1;
  parameter Real c = 0.01;
  Modelica.Mechanics.Translational.Sensors.SpeedSensor speedSensor;
equation
  connect(F, force.f);
  connect(springDamper.flange_b, slidingMass2.flange_a);
  connect(slidingMass1.flange_b, springDamper.flange_a);
  connect(force.flange_b, slidingMass1.flange_a);
  connect(speedSensor.flange_a, slidingMass2.flange_b);
  connect(y,speedSensor. v);
end Servo;
```

Listing 7.3 A Modelica model of a linear servo.

mica will be illustrated by additional examples. In particular, a parameter optimization problem, a static optimization problem, and a multi-case optimization problem will be formulated in Optimica. For the reason of brevity, the annotations for specifying the transcription scheme have been removed from the Optimica descriptions.

A Parameter Optimization Problem

In this section, parameter optimization of a mechanical servo model, depicted in Figure 7.2, is considered. The servo consists of two point masses connected by a spring-damper component. The system is driven by a force

```
optimization ServoParameterOptimization (objective=cost,
                                         startTime=0,
                                         finalTime=100)
① Servo servo(m1(free=true,initialGuess=0.7),
②             slidingMass.s(start=s0));
③ Modelica.Blocks.Sources.Sine sine(amplitude=1, freqHz=0.1);
② parameter Real s0(free=true,initialGuess=0.1);
  parameter Integer N = 1000;
  parameter Real data_vals[N] = {...};
  parameter Real data_times[N] = {...};
④ Real cost = sum((data_vals[i]-servo.y(data_times[i]))^2
                  for i in 1:N);
equation
③ connect(sine.y, servo.F);
constraint
  servo.m1>=0.5;
  servo.m1<=1.5;
  s0>=-1;
  s0<=1;
end ServoParameterOptimization;
```

Listing 7.4 An Optimica specification for a parameter optimization problem.

that is applied to the first mass, and the output of the servo model is the velocity of the second mass. The Modelica code for the model, excluding graphical annotations, is shown in Listing 7.3. The model is composed, essentially, of components from `Modelica.Mechanical.Translational`, which is part of the Modelica standard library.

In this example, the mass of the first point mass, `servo.m1`, is assumed to be unknown. The objective is to find the particular value of `servo.m1` that minimizes a cost function similar to (7.7). It is assumed that data has been obtained from the process, and that the input during data acquisition was a sinusoidal force profile. The corresponding optimization problem is encoded in the Optimica description shown in Listing 7.4. In order to mark the parameter `servo.m1` as a free optimization parameter, the `free` attribute is set to `true`, ①. In addition, an initial guess is specified. The initial position of the second point mass is considered to be unknown. Therefore, the `start` attribute of the variable `servo.slidingMass2.s` is set equal to the parameter `s0`, which in turn is a free optimization variable, ②. The input of the servo component, `servo`, is connected to a sinusoidal signal source, ③. The measurement data points, and the corresponding sampling times, are stored in the arrays `data_vals` and `data_times`, respectively. Based on this data, the cost function, `cost`, is defined as the

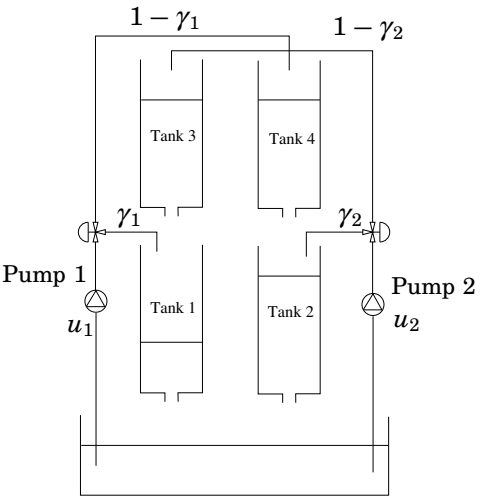


Figure 7.3 A schematic picture of the quadruple tank process

sum of the squared output errors at each sampling instant, ④. Finally, constraints that impose bounds on the optimization variables `servo.m1` and `s0` are introduced. Notice that these bounds are included mainly to avoid large steps in the optimization algorithms, resulting in unrealistically small or large values. The complete flat Optimica description produced by the Optimica compiler is given in Appendix D.

Static Optimization of Operating Point for the Quadruple Tank

In this example, static set-point optimization of a quadruple tank process is considered. The process consists of four interconnected tanks, and two pumps, where the latter are used to control the levels in the tanks. The process is depicted in Figure 7.3, and is described in detail in [Johansson, 1997]. A Modelica model for the process is shown in Listing 7.5 and a corresponding Optimica description for the optimization problem is shown in Listing 7.6. The objective of this static optimization problem is to minimize the criteria

```
quadTank.u[1]^2 + quadTank.u[2]^2
```

subject to the equality constraint

```
quadTank.x[1]=4
```

This corresponds to finding the operating point where the level of the first tank is 4 cm, and where the squared sum of the inputs is minimized. Accordingly, the `static` class attribute is set to `true`, ①, and the array

```
model QuadTank
  // Process parameters
  parameter Real A1=28, A2=32, A3=28, A4=32;
  parameter Real a1=0.071, a2=0.057, a3=0.071, a4=0.057;
  parameter Real kc=0.5;
  parameter Real g=9.81;
  parameter Real k1_nmp=3.14, k2_nmp=3.29;
  parameter Real g1_nmp=0.30, g2_nmp=0.30;
  // Tank levels
  Real x[4](start={1,1,1,1},
            min={0.1,0.1,0.1,0.1},
            max={20,20,20,20});
  // Inputs
  input Real u[2];
equation
  der(x[1]) = -a1/A1*sqrt(2*g*x[1]) + a3/A1*sqrt(2*g*x[3]) +
                 g1_nmp*k1_nmp/A1*u[1];
  der(x[2]) = -a2/A2*sqrt(2*g*x[2]) + a4/A2*sqrt(2*g*x[4]) +
                 g2_nmp*k2_nmp/A2*u[2];
  der(x[3]) = -a3/A3*sqrt(2*g*x[3]) + (1-g2_nmp)*k2_nmp/A3*u[2];
  der(x[4]) = -a4/A4*sqrt(2*g*x[4]) + (1-g1_nmp)*k1_nmp/A4*u[1];
end QuadTank;
```

Listing 7.5 A quadruple tank model.

```
optimization QuadTankOptimization (objective=cost,
①                                  static=true)
② QuadTank quadTank(u(each free=true, initialGuess={1,1}));
  Real cost = quadTank.u[1]^2+quadTank.u[2];
constraint
  quadTank.u <= {10,10};
  quadTank.u >= {0,0};
  quadTank.x <= {20,20,20,20};
  quadTank.x >= {0.2,0.2,0.2,0.2};
  quadTank.x[1] = 4;
end QuadTankOptimization;
```

Listing 7.6 An Optimica specification for a static optimization problem

variable quadTank.u is marked as free in the optimization, ②. In addition, bounds for quadTank.u and quadTank.x are specified.

```
model DoubleTank
  parameter Modelica.SIunits.Area A = 2.8e-3
    "Cross section area of the tanks";
  parameter Modelica.SIunits.Area a = 7e-6
    "Cross section area of the holes";
  parameter Real k(unit="m^3/s/V") = 2.7e-6
  "Constant of proportionality for the pump";
  parameter Real beta(unit="m/s/V") = k/A;
  parameter Real gamma = a/A;
  parameter Real g(unit="m/s^2") = 9.81;

  Modelica.SIunits.Length h1(start=0.0682) "Level of upper tank";
  Modelica.SIunits.Length h2(start=0.0682) "Level of lower tank";

  Modelica.Blocks.Interfaces.RealInput u ;
  Modelica.Blocks.Interfaces.RealOutput y=h2;

equation
  der(h1) = -gamma*sqrt(2*g*h1) + beta*u;
  der(h2) = gamma*sqrt(2*g*h1) - gamma*sqrt(2*g*h2);

end DoubleTank;
```

Listing 7.7 A Modelica model of a double tank system.

Multi-Case Optimization — Tuning of a PID Controller

In this section, an example of a multi-case optimization problem will be given. The particular problem is to find one set of PID-parameters, that gives acceptable performance in a wide operating range, when applied to a non-linear double tank system. The process consists two water tanks, where the first tank is mounted above the second tank. Water is then puring freely from the upper tank to the lower, and a pump is used to control the water flow to the upper tank. The input of the process is the voltage fed to the pump, and the output is the level of the lower tank. The objective of the PID control system is to keep the level of the lower tank at a specified reference value, while rejecting input load disturbances. A Modelica model of the process is shown in Listing 7.7.

In order to evaluate the closed loop performance, a Modelica model implementing the control system, `DoubleTankCL`, was developed, see Figure 7.4. Apart from the double tank system and the PID controller, the model also contains an input for the reference value and a disturbance input. In the Optimica description, see Listing 7.8, three different instances of `DoubleTankCL`, corresponding to different stationary operating points,

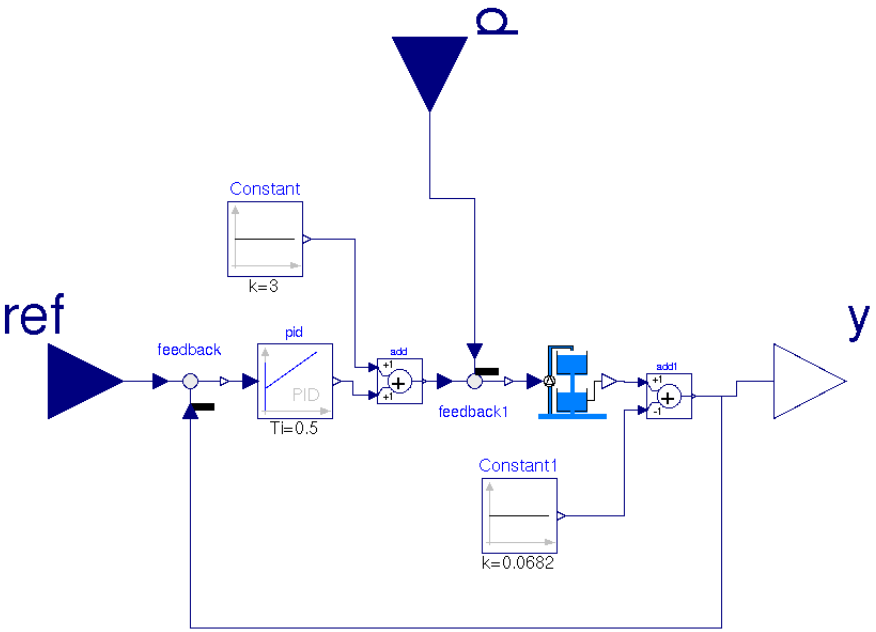


Figure 7.4 A feedback loop consisting of a double tank system and a PID controller.

are created. These operating points correspond to the input pump voltages 1 V, 3 V, and 5 V, which result in lower tank levels of 0.007583 m, 0.0682 m, and 0.1896 m, respectively. Notice that the same PID parameters are used in all instances of `DoubleTankCL`. In the scenario for which the performance is evaluated, a positive reference step of 0.01 m is applied at $t = 30$ s, and a unit input disturbance is applied at $t = 1000$ s. The objective of the optimization problem is then to minimize the sum of the integrated squared errors in the three operating cases. Also, lower bounds for the PID parameters are specified in Listing 7.8.

7.7 Generalizations

While the Optimica extension is dedicated to dynamic optimization of Modelica models, the proposed extension contains two constructs that are of interest also in a more general context. These constructs are the constraint section and the notion of class attributes. In this section, these

```
optimization DoubleTankOptimization (objective=cost(finalTime),
                                     startTime=0,
                                     finalTime=2000)

  parameter Real K(free=true, initialGuess=29);
  parameter Real Ti(free=true, initialGuess=24);
  parameter Real Td(free=true, initialGuess=60);

  Real cost(start=0);

  DoubleTankCL doubleTankCL_1(Constant(k=1),
                              Constant1(k=0.007583),
                              doubleTank(h1(start=0.007583),
                                         h2(start=0.007583)),
                              pid(k=K,Ti=Ti,Td=Td));

  DoubleTankCL doubleTankCL_3(pid(k=K,Ti=Ti,Td=Td));

  DoubleTankCL doubleTankCL_5(Constant(k=5),
                              Constant1(k=0.1896),
                              doubleTank(h1(start=0.1896),
                                         h2(start=0.1896)),
                              pid(k=K,Ti=Ti,Td=Td));

  Modelica.Blocks.Sources.Step refStep(offset=0,
                                       startTime=30,
                                       height=0.01);
  Modelica.Blocks.Sources.Step dStep(startTime=1000,
                                     height=1);

equation
  connect(dStep.y, doubleTankCL_1.d);
  connect(refStep.y, doubleTankCL_1.ref);
  connect(dStep.y, doubleTankCL_3.d);
  connect(refStep.y, doubleTankCL_3.ref);
  connect(dStep.y, doubleTankCL_5.d);
  connect(refStep.y, doubleTankCL_5.ref);
  der(cost) = (refStep.y-doubleTankCL_1.y)^2 +
              (refStep.y-doubleTankCL_3.y)^2 +
              (refStep.y-doubleTankCL_5.y)^2;
constraint
 K>=0;
 Ti>=0;
 Td>=0;
end DoubleTankOptimization;
```

Listing 7.8 An Optimica description for a multi-case optimization problem.

constructs will be discussed from a more general point of view.

Constraints for Specifying Model Validity Regions

Most models have a limited region of validity. For example, approximate linear models derived by linearization of non-linear models are typically valid in a vicinity of the linearization point. Other examples are physical quantities that are inherently positive. Regions of validity are often defined as subsets of the variable space of a model. Currently, Modelica supports specification of validity regions by means of the `min` and `max` attributes of the built-in type `Real`. A tool may then use this information to check that all variables remain within the specified bounds during initialization and simulation. In some cases, however, it may be useful to be able to specify more general regions of validity, for example ellipses. The `constraint` section, which was introduced above, then offers convenient means to specify inequality relations, that can be interpreted as regions of validity.

Class Attributes

The notion of a class attribute is a generally applicable construct, that is not restricted to dynamic optimization problems. It has been noted that a similar mechanism has been introduced in the statechart extension supported by Mosilab, but with a different syntax. The same need may arise also in other potential extensions. Consider, for example, an extension of Modelica to also support embedded systems. Such an extension may contain a specialized class, say `task`, which corresponds to a task, or process, executing on a multi-threaded embedded processor. A task is typically associated with a set of attributes, such as e.g., priority, input latency and dead-line. Also in this case, class attributes would provide a convenient mean to supply this information.

7.8 Summary and Conclusions

In this chapter an extension of the Modelica language, Optimica, that enables high-level formulation of dynamic optimization problems, has been presented. The Optimica extension enables the user to specify important elements of a dynamic optimization problems such as cost functions, constraints and optimization interval. The dynamic model, upon which the dynamic optimization problem is based, is expressed using standard Modelica. Optimica also supports an annotation that enables the user to specify the properties of a transcription method, based on direct collocation. Because of these properties, Optimica supports formulation of dynamic

optimization problems, using high-level constructs, both at the mathematical level and at the numerical transcription level.

A prototype implementation of the Optimica compiler has been used in the work on start-up optimization of a plate reactor (see Chapter 9), in two master's thesis projects (see [Danielsson, 2007] and [Hultgren and Jonasson, 2007]) and in the PhD course "Optimization-Based Methods and Tools in Control", that was given at the Department of Automatic Control, Lund University in September 2007. A new version of the Optimica compiler, supporting the syntax and semantics presented in this chapter, is currently under development. The front-end of the new compiler is fully functional, and it remains to implement support for the transcription annotation, and to adapt the AMPL code generation back-end used in the prototype compiler.

An important objective of the JModelica compiler is to offer a modularly extensible Modelica compiler. In this respect, the experiences and results from developing the Optimica extension are very promising. In particular, the coding effort needed to implement the extension of the compiler front-end, including extension of the name analysis framework and the flattening algorithm, was very moderate.

Case Studies

III

III

8

DryLib

8.1 Introduction

The topic of this chapter is modeling, model reduction, parameter optimization and control of a paper machine dryer section. The dryer section is the last part of the paper machine and consists of a large number of rotating steam heated cast iron cylinders. The moist paper is led around these cylinders and the latent heat of vaporization in the steam is used to evaporate the water from the web. When the steam releases its thermal energy it condenses into water which is drawn off by suction with a siphon and fed back to the boiler house. The cylinders are divided into separate dryer groups where the steam pressure can be individually controlled in each group. By adjusting the steam pressure in the dryer groups, and thereby the heat flow to the paper, the moisture in the paper web is controlled. The moisture ratio in the web is reduced from 1–1.5 kg water/kg dry substance when entering the dryer section to a final product of 0.03–0.1, i.e., a significant amount of water is removed in the dryer section.

To support and transport the paper web through the dryer section, dryer fabrics are used. The dryer fabric is also used to press the web onto the cylinders to provide good thermal contact between the two surfaces.

The dryer section is enclosed inside a dryer hood. The main purposes of the hood are to create a controlled environment for the drying process, improve energy utilization, and also to establish good working conditions in the machine room. The exhaust air removes the evaporated water from the paper web while preheated dry air is added to the hood by the supply air.

Moisture is one of the most important quality parameters of the final paper product. It is essential to keep this property well regulated, both at steady-state and at state-transitions. A good model of the dynamics of

drying is therefore vital for good moisture control. Based on the work in [Slätteke, 2006], a Modelica library, DryLib, has been developed. DryLib implements the physical phenomena involved in the drying process, as well as convenient components and connectors which enable rapid development of dryer section models. An important feature of DryLib is its ability to express models which are scalable, in the sense that the complexity of the models can be easily changed. This feature is quite useful, since the need for granularity depends on the application—a high fidelity model may be suitable for simulation, whereas a course model capturing the main behavior may be appropriate for control design.

Three contributions are given in this chapter. Firstly, the Modelica library DryLib is presented. Secondly, parameter optimization, model reduction and an optimization based control scheme (Non-linear Model Predictive Control (NMPC)), are treated. Some of these topics have a general character, while others are dealing specifically with dryer section issues. Thirdly, the applications of the chapter serves as examples of the wide range of relevant optimization problems that naturally follow the availability of high-fidelity models.

The work presented in this chapter has been done without the use of Optimica, and the Optimica compiler, which were presented in Chapter 7. Certainly, the Optimica compiler would have been very useful for solving the optimization problems formulated and solved in this chapter. In particular, the same Modelica model as was used for simulation and parameter optimization could then have been reused in the context of the model reduction problem. The main reason why the Optimica compiler was not used to solve this problem is that it was not developed at the time when this work was done. It should be noticed, however, that the current version of the Optimica compiler is not immediately applicable to this problem, since the model contains constructs, notably dynamic name lookup (inner/outer constructs) that are currently not supported. The optimization problems presented and solved in this chapter do, however, serve as an excellent motivation for the development of Optimica. In particular, an extensive coding effort was required to tailor the particular problems to the algorithms that were used to solve the problems numerically. The availability of a dedicated high-level language for dynamic optimization, and supporting tools, would drastically have reduced this effort.

This chapter is organized as follows. In Section 8.2 the physical model upon which DryLib is based, is presented. Section 8.3 deals with the structure and implementational details of DryLib. The Sections 8.4, 8.5 and 8.6 treat parameter optimization, model reduction and moisture control by means of non-linear MPC. In Section 8.7, the software used to solve the optimization problems presented in the paper is described. The chapter ends with conclusions and future work in Section 8.8.

8.2 Physical Modeling

Modeling of the Dryer Section

Mathematical modeling of cylinder drying started with the pioneer work [Nissan and Kaye, 1955]. An extensive review of drying models up to 1980 with some 130 references is given in [McConnell, 1980]. Many of these models have different objectives and are of different type. There are both static and dynamic models, and a majority of the models are first principles models but some describe black-box modeling of the dryer section. One mutual characteristic of the models is that they often focus on modeling the paper sheet and neglect the steam system. Consequently it is assumed that the steam pressure in the cylinders is a manipulated variable or that a collected data series of the steam pressures is used as an input to the model. This makes the model unsuitable for simulation of feedback control. The model described in this work includes the dynamics of the steam system and the inflow of steam is controlled by a steam valve. It is therefore possible to mimic the entire moisture loop of the feedback system in the paper machine.

The model library that is developed and used in this chapter is built upon physical relations in terms of mass and energy balances, in combination with constitutive equations for the mass and heat transfer. The objective is to obtain a non-linear model that captures the key dynamical properties for a wide operating range. The model formulation used here is developed in [Slätteke, 2006], which in turn is based on [Wilhelmsson, 1995] and [Slätteke and Åström, 2005]. The model for the paper web is based on [Wilhelmsson, 1995] whereas the model for the cylinder, and steam system is taken from [Slätteke and Åström, 2005]. The model description given here is mainly given for completeness, but there are also some minor additions as compared to the description given in [Slätteke, 2006].

While the physical behavior of the process is formulated using partial differential equations (PDE:s), numerical simulation usually require the PDE:s to be discretized in the spatial dimension(s). In this work, the paper process is discretized by partitioning the process into control volumes where a mass and energy balance are defined for each volume. These control volumes are then put together so that the outflow of one becomes the inflow of the next. The precision of the model then depends on the size of the control volumes, where a finer discretization grid gives improved accuracy, but also increased computational complexity. In the models developed in in [Wilhelmsson, 1995] and [Karlsson, 2005], the paper web is discretised both in the machine direction and in the z-direction, using

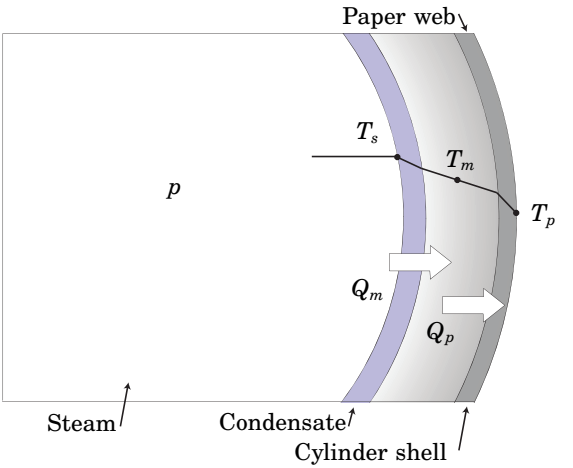


Figure 8.1 A piece of the cross-section of a drying cylinder, showing the steady-state temperature profile and energy flows.

small elements. In the model used here, a courser discretization is used, where the paper web is discretized only in the machine direction. The length of a control volume corresponds here to the length of a free draw or the length of a paper web section in contact with a cylinder. While this discretization scheme offers decreased accuracy as compared to those used in [Wilhelmsson, 1995] and [Karlsson, 2005], it has the distinct advantage of significantly decreased simulation time.

In order to increase the clarity of the presentation, the indices identifying each individual control volume have been dropped. In Figures 8.2 and 8.3, however, the indices have been included to emphasize the discrete nature of the paper process model.

The Steam and Cylinder Process

Let q_s [kg/s] be the mass flow rate of steam into the cylinder, q_c [kg/s] the condensation rate, q_{bt} [kg/s] the blow through steam, and q_w [kg/s] the siphon flow rate. Also, let V_s [m^3] and V_w [m^3] be the volumes of steam and water, respectively, in the cylinder, and let ρ_s [kg/m^3] and ρ_w [kg/m^3] be the respective densities. The mass balances for water and steam are then

$$\begin{aligned} \frac{d}{dt}(\rho_s V_s) &= q_s - q_c - q_{bt} \\ \frac{d}{dt}(\rho_w V_w) &= q_c - q_w \end{aligned} \tag{8.1}$$

The energy balances for steam, water and metal are given by

$$\begin{aligned}\frac{d}{dt}(\rho_s u_s V_s) &= (q_s - q_{bt})h_s - q_c h_s \\ \frac{d}{dt}(\rho_w u_w V_w) &= q_c h_s - q_w h_w - Q_m \\ \frac{d}{dt}(m C_{p,m} T_m) &= Q_m - Q_p\end{aligned} \tag{8.2}$$

where Q_m [W] is the power supplied from the water to the metal, Q_p [W] is the power supplied from the metal to the paper, h_s [J/kg] is the steam enthalpy, h_w [J/kg] is the water enthalpy, m [kg] the mass of the cylinder shell, $C_{p,m}$ [J/(kg·K)] the specific heat capacity of the shell, T_m [K] the mean temperature of the metal, and u_s [J/kg] and u_w [J/kg] are the specific internal energies of steam and water. From the thermodynamic definition of specific enthalpy, we get

$$\begin{aligned}h_s &= u_s + \frac{p}{\rho_s} \\ h_w &= u_w + \frac{p}{\rho_w}.\end{aligned} \tag{8.3}$$

where p [Pa] is the steam cylinder pressure. The steam and water volumes add up to the total cylinder volume,

$$V = V_s + V_w \tag{8.4}$$

The energy flow to the metal is given by the heat transfer equation

$$Q_m = \alpha_{sc} A_{cyl}(T_s - T_m) \tag{8.5}$$

where α_{sc} [W/(m^2·K)] is the heat transfer coefficient from the steam-condensate interface to the centre of the cylinder shell, A_{cyl} [m^2] is the inner cylinder area, and T_s [K] the steam temperature. Experiments have shown that α_{sc} depends on both condensate thickness, machine speed, and the number of spoiler bars [Pulkowski and Wedel, 1988]. However, the condensate has a turbulent behavior and the heat transfer coefficient has proven to be difficult to model. Typical values range between 1000 and 4000 W/(m^2·K). The power flow to the paper is

$$Q_p = \alpha_{cp} A_{cyl} \eta (T_m - T_p) \tag{8.6}$$

where T_p [K] is the paper temperature, α_{cp} [W/(m^2·K)] the heat transfer coefficient from the cylinder shell to the paper, and η [·] is the fraction

of dryer surface covered by the paper web. Figure 8.1 illustrates the heat flows in the steam and cylinder model. An empirical model for α_{cp} has been developed in [Wilhelmsson, 1995] where a linear relation with moisture ratio in the paper web, u[1] [kg moisture/kg dry solids] is proposed. The relation is given by

$$\alpha_{cp}(u) = \alpha_{cp0} + \alpha_{cpk} u \tag{8.7}$$

where α_{cp0} varies between 200-500 W/(m^{2}K) and α_{cpk} has typical values in the range of 900-1200 W/(m^{2}K). It is well known that α_{cp} depends on other things, e.g. the web tension, and surface smoothness of both paper and cylinder, but these phenomena are omitted here. The energy flow from the part of the cylinder not covered by paper due to convection or radiation to ambient air has been reported to represent only 1-2% of the total energy flow and is therefore neglected, see [Wilhelmsson, 1995]. Since the steam flow to the cylinder cannot be manipulated directly, a valve model is also needed.

From [Thomas, 1999] we have

$$q_s = C_v f_v(x_v) \sqrt{(p_{sh} - p)\rho_s}, \tag{8.8}$$

where C_v [m^2] is the valve conductance, x_v is the position of the valve stem and the function f_v is the valve characteristics called valve trim. The valve stem varies from 0 (minimum valve opening) to 1 (maximum valve opening). The supply pressure at the steam header is p_{sh}. We use equal percentage trim, since it is the most common characteristic in the process industry [Hägglund, 1991]. This assumption gives

$$f_v(x_v) = R_v^{x_v - 1}. \tag{8.9}$$

where R_v is a constant known as the "rangeability" since it is the ratio between the maximum and minimum valve opening.

For simplicity, all steam within the cylinder cavity is assumed to be homogeneous, with the same pressure and temperature. We also assume that the steam in the cylinder is saturated. This means that the enthalpy, density, and temperature are functions of the pressure only. Fitting polynomials to the tabulated values for saturated steam in [Schmidt, 1969],

[1]The symbol u is used to denote moisture ratio in this chapter, but is also used to denote control inputs elsewhere in this thesis.

gives

$$\begin{aligned} T_s &= 0.1723(\log p)^3 - 3.388(\log p)^2 + 37.71\log p + 124.5 \\ h_s &= (-0.07402(\log p)^4 + 2.887(\log p)^3 \\ &\quad - 39.58(\log p)^2 + 260\log p + 1824) \times 10^3 \\ h_w &= (0.8842(\log p)^3 - 18.77(\log p)^2 + 200\log p - 748.5) \times 10^3 \\ \rho_s &= (0.005048p + 64.26) \times 10^{-3} \\ \rho_w &= -0.3136(\log p)^3 + 6.792(\log p)^2 - 52.43\log p + 1141 \end{aligned} \tag{8.10}$$

The Paper Web Process

The water and fiber content of the paper web are modeled by mass balances, whereas the temperature of the web is modeled by an energy balance. Starting with the mass balance of water, an expression defining the evaporation rate (condensation rate) between the paper surface and the surrounding air is needed. From [Wilhelmsson, 1995] we get the Stefan equation

$$q_{evap} = \frac{p_{tot} K_G M_w}{R_g T_p} \log\left(\frac{p_{tot} - p_{v,a}}{p_{tot} - p_{v,p}}\right), \tag{8.11}$$

where q_{evap} [kg/m^{2}s] is the evaporation rate, K [m/s] is the mass transfer coefficient, M_w [kg/mole] is the molecular weight of water, p_{tot} [Pa] the total pressure of the air, $p_{v,a}$ [Pa] the partial pressure for water vapor in the air, $p_{v,p}$ [Pa] the partial pressure for the water vapor at the paper surface, R_g [J/mole·K] the gas constant, and T_p [K] the paper temperature. The partial pressure $p_{v,a}$ is given by the moisture content of air, x [kg water vapor/kg dry air], and the total pressure, [Karlsson, 2000]

$$p_{v,a} = \frac{x}{x + 0.62} p_{tot}. \tag{8.12}$$

The vapor partial pressure at the paper surface is given by

$$p_{v,p} = \varphi p_{v0} \tag{8.13}$$

where φ is the sorption isotherm, and p_{v0} [Pa] is the partial vapor pressure for free water, and is given by Antoine's equation

$$p_{v0} = 10^{\left(10.127 - \frac{1690}{T_p - 43.15}\right)} \tag{8.14}$$

As long as capillary transport can bring new water to the paper surface, the vapor partial pressure at the paper surface is equal to the partial pressure for free water. When the paper becomes dryer a correction factor

called sorption isotherm, φ, is invoked which has a value between zero and one. In [Pettersson and Stenström, 2000] an investigation of some sorption isotherms found in the literature, is given. Many of those give a heat of sorption that goes to infinity as u goes to zero. This is physically unrealistic since the bond energy between the last fraction of water and a cellulose fiber must be finite. From [Heikkilä, 1993], a finite heat of sorption at the origin which matches the hydrogen bond energy between water-fiber is given and is therefore found to be most appropriate. The sorption isotherm of a paper web depends on its composition and temperature. It is not very well investigated when compared to other materials, [Pettersson and Stenström, 2000], but [Heikkilä, 1993] gives an empirical expression for paper pulp,

$$\varphi = 1 - \exp(-47.58u^{1.877} - 0.10085(T_p - 273.15)u^{1.0585}) \tag{8.15}$$

Now, let v_x [m/s] be the speed of the paper web, d_y [m] the width of the paper web, A_{xy} [m^2] the area of the dryer surface covered by paper, and g [kg/m^2] the dry basis weight. Notice that $A_{xy} \approx A\eta$ assuming that the thickness of the cylinder is small. Then the mass balance of moisture for a paper sheet in contact with a cylinder can be written

$$\frac{d}{dt}(ugA_{xy}) = d_y v_x g_{in} u_{in} - A_{xy} q_{evap} - d_y v_x g u. \tag{8.16}$$

A similar mass balance for moisture in the free draws can be derived from Figure 8.2, which shows a schematic picture of the mass flows in a paper sheet. Analogously, the mass balance for fiber in the paper web is given by

$$\frac{d}{dt}(gA_{xy}) = d_y v_x g_{in} - d_y v_x g. \tag{8.17}$$

To model the energy balance, introduce

$$C_{p,p} = \frac{C_{p,fiber} + uC_{p,w}}{1+u} \tag{8.18}$$

where $C_{p,p}$ [J/kg·K], $C_{p,fiber}$ [J/kg·K], and $C_{p,w}$ [J/kg·K] are the specific heat capacity of paper, fiber and water, respectively. As we can see, $C_{p,p}$ is a weighted sum of the heat capacities of the parts. From [Wilhelmsson, 1995] we have $C_{p,fiber} = 1256$ J/(kg·K). Also, let T_p be the paper temperature and ΔH be the amount of energy needed to evaporate the water. Analogously to the discussion about the mass balance, if the web is wet enough this energy is equal to the latent heat of vaporization for free water. When the paper becomes dryer, however, an extra amount of

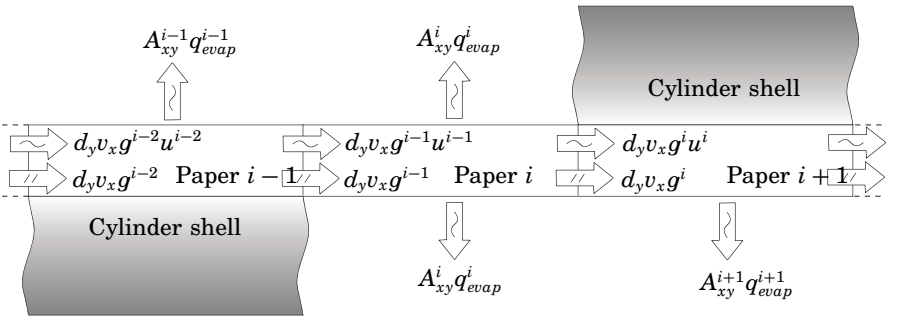


Figure 8.2 The mass transport for water and fiber in the paper web. The shaded areas represent cylinder walls. When the paper is in the transition between two cylinders (the free draw), evaporation occurs at both paper surfaces.

energy ΔH_s (the heat of sorption) is necessary besides the latent heat of vaporization for free water. The heat of sorption can be derived from the sorption isotherm by thermodynamic theory and this relation is known as the law of Clausius-Clapeyron

$$\Delta H_s = -\frac{R_g}{M_w}\left(\frac{d(\ln\varphi)}{d(1/T_p)}\right). \tag{8.19}$$

By applying this relation to (8.15), we obtain

$$\Delta H_s = 0.10085u^{1.0585}T_p^2 R_g \frac{1-\varphi}{M_w\varphi} \tag{8.20}$$

The amount of energy required to evaporate water from the surface of the web is then given by

$$\Delta H = \Delta H_{vap} + \Delta H_s \tag{8.21}$$

where H_{vap} is the latent heat of vaporization for water, equal to 2260 kJ/kg (at atmospheric pressure). Furthermore, let the energy transport due to convection between the paper surface and the air be

$$Q_{conv} = \alpha_{pa}A_{xy}(T_p - T_a) \tag{8.22}$$

where α_{pa} [W/(m^2·K)] is the heat transfer coefficient from paper to air and T_a [K] the ambient air temperature. Since water is an incompressible

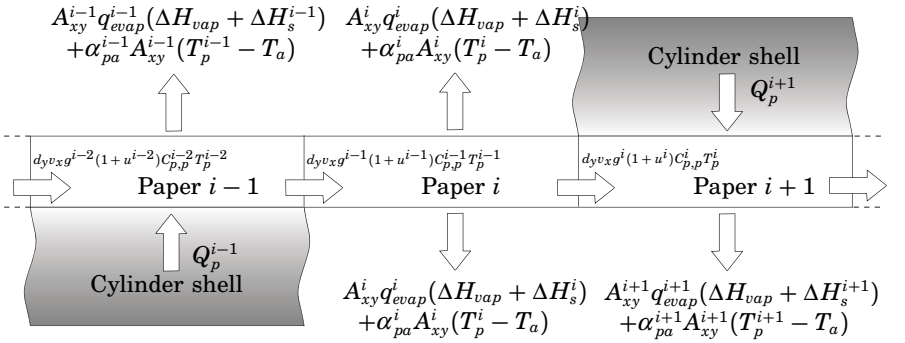


Figure 8.3 The energy balance of the paper web. The shaded areas represent cylinder walls. When the paper is in the transition between two cylinders (the free draw), energy flow to ambient air occurs at both paper surfaces.

medium, there is no pressure volume work on the surroundings, and we write the energy balance as a change in enthalpy. The energy balance of the paper web in contact with a cylinder is thus modeled as

$$\begin{aligned}\frac{d}{dt}(g(u+1)A_{xy}C_{p,p}T_p) &= d_y v_x g_{in}(1+u_{in})C_{p,p,in}T_{p,in}\\ &- A_{xy}q_{evap}(\Delta H_{vap}+\Delta H_s)-\alpha_{pa}A_{xy}(T_p-T_a)\\ &- d_y v_x g(1+u)C_{p,p}T_p+Q_p\end{aligned} \tag{8.23}$$

The energy balance for the free draws is similar, an can be formulated using the schematic illustration of energy flows shown in Figure 8.3.

8.3 DryLib

DryLib is implemented in Modelica, see Chapter 2. The objective of building the Modelica library DryLib has been to create a user friendly and extensible platform for modeling of paper machine dryer sections. In particular, the aim has been to design the library so that, at the user level, the appropriate level of model detail can be easily selected. The current implementation of DryLib contains a few examples of components where the level of detail can be specified by the user. More importantly, the library classes are designed to enable advanced users to add new behavior

to key components in order to extend the functionality of the library. An important concept in the design process has been that of *model scalability*, which means that the granularity of the model behavior should be easy to change, without the need to re-build the model.

Hierarchical Structuring

Having formulated a mathematical model for the paper machine dryer section, as presented in Section 8.2, the issues of structuring the equations into Modelica classes, and definition of interface classes (connectors) need attention. A paper machine dryer section model can be assembled using very few basic component types. In essence, there are only two fundamental entities, namely a steam heated cylinder and a sheet of paper. These two component types may then be combined, in large numbers, into a complete dryer section model. However, it is convenient to introduce additional hierarchical levels. As discussed above, the cylinders of a typical dryer section are organized into steam groups, in which a number of cylinders are operated at the same pressure. The introduction of steam groups into the library provides a convenient hierarchical level for the user, since many decisions regarding e.g. operating points and control design and evaluation are made at the steam group level. For basic usage of DryLib, it is also sufficient to utilize only classes defined at the steam group level in order to create a fully working dryer section model.

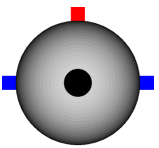
In order to increase the flexibility of the library, the boundary conditions of the physical entities have been factored out and modeled as separate classes. As an instructive example we consider a paper sheet, where the boundary conditions of the surfaces defining the sheet depend on the environment. For example, different boundary conditions are imposed on the surface if the paper is in contact with the air or a cylinder shell. The key to building a flexible Modelica library using this principle of separation is the design of generic connector classes. This topic will be discussed in detail below.

From a user's perspective, DryLib is intended to enable easy modeling of a dryer section. However, the user should remain in control of the implementational details of key components, e.g. paper sheets and cylinders. Also, advanced users should have the possibility to introduce new behavior of existing components. Two key features of Modelica have been used to satisfy these requirements. In the first case, parametrized classes (`replaceable`/`redeclare`) has been used to propagate type information downwards in the component hierarchy from the main user level (which is the steam group level) to lower level components. This strategy enables the user to easily select the appropriate level of detail for e.g. the cylinder dynamics. In the second case, inheritance has been used in order to simplify introduction of new component behavior. For the ba-

sic components such as cylinders and paper sheets, generic base classes have been introduced, which in turn serve as super classes for particular implementations. DryLib currently provides a few alternative implementations for key components, and additional behavior is easily added using the pre-defined base classes.

Connectors and variable bindings The interface structure in DryLib is based on three connector classes. While the connectors for heat flow and mass flow (for connecting components with steam flow) are straight forward, the connector class for a paper surface deserves to be discussed. The paper web is modeled by separate mass balances for water and fiber, and an energy balance, as described above. Natural flow variables are thus mass flow of water and fiber, q_w [kg/s] and q_f [kg/s], and energy flow Q [W]. As for the potential variables, there are several feasible choices. However, since DryLib is likely to be used by domain experts in the field of paper drying, it was decided to use the standard variables within this domain. The natural choices are then moisture ratio, u [kg water/kg dry substance], dry basis weight, g [kg/m^2] and temperature T [°C].

A particular feature of Modelica that has been used to simplify the propagation of parameters and variables between components in DryLib is dynamic name look-up in the instance hierarchy (`inner/outer`). For example, the machine speed is used in various components, but is common for the entire dryer section. Implementation using `inner/outer` constructs is thus convenient. Examples of variables that may be assumed to be shared by the components of a steam group are ambient temperature and air moisture, which are also implemented using `inner/outer`.

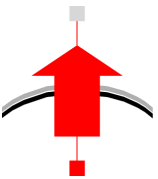


Cylinder Models The cylinder base class `CylinderBase`, which is partial, contains mainly connector components and serves as a unifying class for particular implementations of dynamic behavior. The cylinder base class has two mass flow connectors corresponding to steam inlet and outlet, and one heat flow connector. Currently, DryLib contains two implementations of cylinder dynamics. The first implementation is based on Equations (8.1)-(8.5) and (8.10), whereas the second implementation is based on the simplified linear dynamics derived in [Slätteke, 2006], Chapter 4.

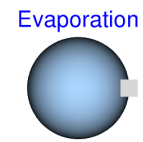
Paper Models The paper web base class contains essentially four paper connectors corresponding to the cross section areas and the upper and lower surfaces. This design enables separation of the actual paper web behavior, and the physical phenomena defined by the boundary conditions of the paper. The design also adds to the flexibility of the library, enabling, for example, easy extension to

modeling of multi-ply paper drying. There are two particular implementations of paper web behavior. In the first implementation, the dynamics is included, whereas in the second implementation the balance equations are given as algebraic relations. The latter case is motivated by the fact that the time constants of the paper web is small compared to the cylinder dynamics. Neglecting the fast dynamics of the paper may be attractive for applications where it is important to minimize the number of dynamical states.

Interfaces A key component is the connection of the cross section areas of two paper sheets. It is important to note that the single mechanism that drives the mass transport in the machine direction is the mechanical transportation of the paper, defined by v_x. Diffusion due to moisture gradients is neglectable given the high velocity of the paper through the dryer section, and is therefore not modeled. As a consequence, the mass and energy flows through the cross section area cannot be determined locally (compare e.g. mass transport driven by pressure gradients), but relies solely on the machine speed v_x. Using these arguments, implementation of the `PaperPaperInterface` class is straight forward and involves only encoding of appropriate terms of the right hand sides of Equations (8.16), (8.17) and (8.23).



The interface between a steam cylinder and a paper surface is modeled by the class `CylinderPaperInterface`, which has one heat flow connector and one paper connector. The behavior of the class is defined by Equation (8.6), which implies that energy transport takes place but not mass transport.



Much of the modeling effort in Section 8.2 was devoted to describing evaporation of water from the paper surface. This phenomena is encapsulated in the class `Evaporation`, which contains the associated equations ((8.11)-(8.15) and (8.20)-(8.22)) defining the mass and energy flows through the paper surface.

Steam group models The classes described above have the character of *specifying physical behavior*. We shall now turn our attention to classes which are mainly used as *structuring entities* in the sense that they introduce new hierarchical levels, and that they contain instances of behavior classes. Basic usage of DryLib may involve only classes introduced at this level.

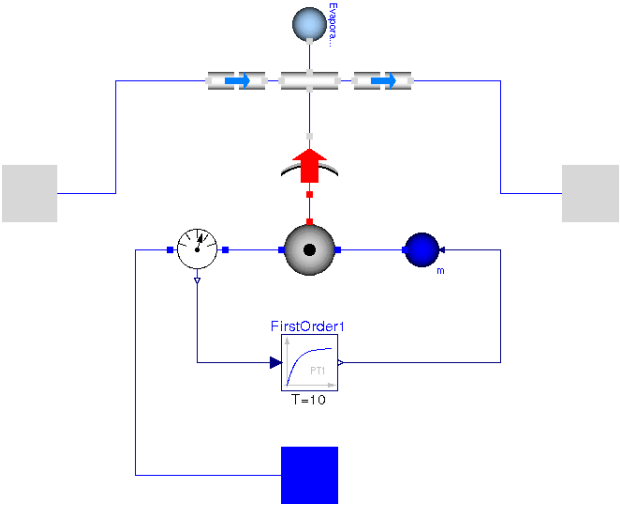
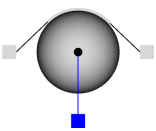
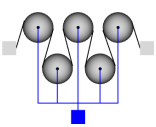


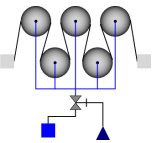
Figure 8.4 The component diagram for `CylinderUnit`.



In order to efficiently explore the strong repetitive character of a typical dryer section, the class `CylinderUnit` was introduced. As can be seen in Figure 8.4, this class combines a steam cylinder and a paper sheet which is attached to an evaporation component. While different cylinders may have different physical parameters, the structure of `CylinderUnit` is valid in most cases. A difficulty when modeling a steam cylinder is to determine the behavior governing the blow through steam and condensate flows. In [Slätteke, 2006], a simplified model which relates the input mass flow and the output mass flow by a first order system is used. Evaluation by simulation has shown that this model for the cylinder outlet gives acceptable results, and also that the choice of time constant for the first order system is not critical.

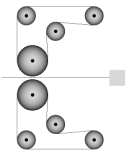


As noted above, a steam group is an important entity of a dryer section. In order to obtain increased flexibility, a steam group in DryLib is modeled by two classes—one for the actual cylinders and one for the associated control system. The class `CylinderArray` contains an arbitrary number of `CylinderUnit` components, and provides a convenient way to create large cylinder groups.

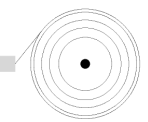


The actual control system, typically consisting of a valve, a pressure sensor and a PID controller, is encapsulated in the class `SteamGroup`, which also contains a `CylinderArray`, component representing the actual cylinders and the paper sheet. The `SteamGroup` class has four connectors corresponding to incoming and outgoing paper, the steam header and an input signal representing the reference value of the pressure controller.

Sources and sinks Apart from the classes presented above, DryLib also contains classes which are used to drive a dryer section model, referred to as sources and sinks.



In a paper plant, there are several process steps preceding the dryer section. In particular, the wet end, consisting of the wire part and the press, is also considered to be part of the paper machine. Since these components are not included in DryLib, it is necessary to introduce a class which generates an output corresponding to the wet end. This mechanism is encapsulated in the class `PaperSource`. This class is equipped with a paper connector and simulates the incoming paper sheet given specifications for water and fiber mass flow and paper temperature.



Since the dryer section is the last part of the paper machine, the ending interface is straight forward and consists mainly of a paper connector which interfaces the last cylinder group of a dryer section. In addition, there is a version of the `PaperSink` class which offers outputs corresponding to output temperature, output moisture and output dry basis weight.

PM7, Husum, Sweden

To demonstrate the capabilities of DryLib, a dryer section model corresponding to that of PM7 located at the M-real mill, Husum, Sweden, has been developed. The PM7 paper machine is a multi-cylinder machine producing copy paper. The dryer section of the machine is divided into a pre-dryer and an after dryer section with the surface sizing in the middle. The objective of the after-dryer section is only to dry the mixture added by the surface sizing and it cannot take care of moisture problems from the pre-dryer section. Only the pre-dryer is modeled here. The PM7 drying cylinders are divided into six groups, consisting of one, two, two, three, ten and twelve cylinders respectively. For a detailed description of the plant, see [Ekvall, 2004].

In Figure 8.5, the top level of the PM7 dryer section model is shown, including six steam groups, a paper source, a paper sink, a mass flow source representing the steam header and a set point distribution for

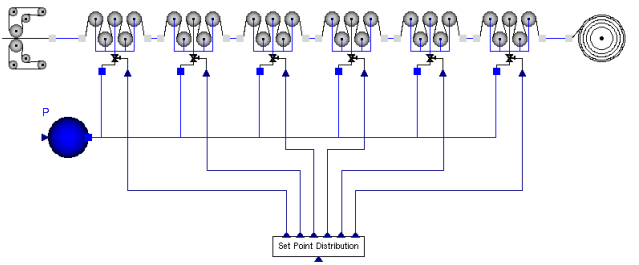


Figure 8.5 The top level of a complete dryer section model.

calculation of pressure set points for the groups. The final model consists of 7453 equations and 312 dynamical states when translated with Dymola.

Extensions

Possible extensions of DryLib can be sorted mainly into two categories. Firstly, the library may be extended by adding components modeling process equipment or physical phenomena not covered by the current implementation. For example, modeling of systems in direct connection with the dryer section, such as the condensate system, the steam production and the ventilation system would enable simulation of a larger part of the process. Also, adding this functionality would simplify connection of the dryer section model to models of other important parts of the paper machine, e.g. the press section, the wire section or other process units utilizing the same steam header.

In an effort to extend the functionality of DryLib, the library has been extended to include components of the condensate systems, such as tanks, pipes, and siphons, see [Windahl, 2006].

Secondly, DryLib may be extended by introducing components which enable simulation of the drying process at an increased level of detail. The current design of DryLib is based on a particular choice of discretization of the underlying PDE:s (describing mass and energy transport), which yields a model with a reasonable level of detail, while maintaining acceptable simulation times. While this choice of discretization is suitable for analysis of moisture, temperature and pressure profiles in the machine direction, other applications may require different levels of detail. For example, in the work [Karlsson, 2005], the underlying PDE:s are discretized at a very high level of detail. This enables e.g. analysis of the risk of delamination in cartonboard manufacturing, as well as detailed study of

moisture and temperature profiles, in the machine and thickness directions. Other applications, such as control design, may benefit from simple models capturing only the input-output behavior of the system. This issue is addressed in Section 8.5, where a model reduction scheme is proposed to reduce the complexity of a dryer section model.

8.4 Parameter Optimization

It is desirable that the behavior of the model is similar to that of the real plant, in order for results obtained from using the model to be applicable to the plant. It is usually necessary to modify the original model to obtain a better match with measurement data. A common method to minimize the plant-model mis-match is to select one or more parameters of the model, and then tune these until a satisfactory model response is obtained. This procedure of tuning parameters, without taking stochastic properties of the measurement data into account, is referred to as white-box identification, see [Bohlin and Isaksson, 2003]. Parameter tuning may in simple cases be done by hand, but more complex problems require structured methods for finding the parameter set that yields the best result. One such method is parameter optimization, which, in addition to selection of parameters to optimize, also includes definition of a performance criterion to minimize.

Model parameter values can be determined in several ways. Some parameters are available in tables, and are not associated with uncertainty, whereas others may be determined from experiments. Mechanical systems may for example be disassembled and their components can be measured and weighted. Yet some parameters may be inherently hard to find accurate values for. In the dryer section model, heat transfer coefficients typically fall into this category.

When selecting parameters to optimize, parameters which are uncertain are attractive choices. However, it should be kept in mind, that the parameter optimization procedure does not necessarily produce the physically correct parameter values. Rather, the selected parameters are used to compensate for all types of model-data mismatch given a particular performance criterion. This implies that the actual parameter values obtained from optimization should not be interpreted as the true physical values, but rather those that achieve the best model-data match. On the other hand, it is usually desirable to ensure that parameters have physically reasonable values.

Table 8.1 Optimization parameters

Parameter	Nom.	Min.	Max.
α_{sc} [W/(m^{2}K)]	500	400	5000
K [m/s]	0.06	0.02	0.1
α_{p0} [W/(m^{2}K)]	400	200	1000

Problem Definition

Setting up a parameter optimization problem requires insight into which aspects of the model are most important. In this case, both the dynamic and static model response is of importance. However, here, only the static behavior has been considered. Specifically, cylinder and paper temperatures of the paper machine, as well as the output moisture, have been measured during stationary operation conditions. The aim of the optimization has been to improve the stationary response of the model in the sense that the difference between simulated temperatures and moisture and measured temperatures and moisture, should be minimized.

A reasonable cost function to minimize is then

$$\begin{aligned} J = &\gamma_{T_m} \sum_{i=1}^{N_{cyl}} (T^m_{m,i} - T^s_{m,i})^2 + \\ &\gamma_{T_p} \sum_{i=1}^{N_{cyl}} (T^m_{p,i} - T^s_{p,i})^2 + \gamma_u (u^m_{out} - u_{out})^2 \end{aligned} \tag{8.24}$$

where N_{cyl} is the number of cylinders, superscript m indicates measured quantities, superscript s indicates simulated quantities and γ_{T_m}, γ_{T_p}, and γ_u are weights. While the measurement method used to determine cylinder temperatures is reliable, the measurements of paper temperatures should be regarded as uncertain. In particular, the paper temperature is varying considerably in the machine direction depending on the position, relative to a cylinder contact area, at which the measurement is taken, [Slätteke, 2006]. Therefore, the weight γ_{T_p} was set to a small value. The moisture, on the other hand, is an important quality variable that should be matched with high accuracy. Accordingly, γ_u was set to high value.

Three parameters were selected for optimization:

- The heat transfer coefficient between steam and condensate in a cylinder, α_{sc} in equation (8.5)

- The mass transfer coefficient K_G which is used in the expression (8.11) defining evaporation of water from the paper surface

- The heat transfer coefficient between cylinder and paper is given by the expression $\alpha_p = \alpha_{p_0} + \alpha_{p_K} u$, where u is the moisture of the paper. α_{p_0} was selected for optimization.

These parameters are used to specify the properties of each individual steam cylinder, and could therefore, in principle, have different values for different cylinders. In the following, two different approaches will be taken. Firstly, parameter optimization where the same parameter values are used for all cylinders will be performed. Secondly, the possibility to further reduce the model-data mismatch by assuming that the cylinders in different groups may have different parameter values, will be explored. Table 8.1 summarizes nominal, maximum and minimum values for the parameters.

Solving the Problem

The minimization of (8.24) should be performed subject to the constraint constituted by the DAE representation of the model. Since the minimization is performed in stationarity, all derivatives may be set to zero, and the model is then represented by a purely algebraic constraint, $F(x, y, p) = 0$, where x is the state vector, y represents the algebraic variables and p are the parameters.

The optimization problem may now be written

$$\begin{aligned} \min_{x,y,p} J = \min_{x,y,p} \; & \gamma_{T_m} \sum_{i=1}^{N_{cyl}} (T^m_{m,i} - T^s_{m,i})^2 + \\ & \gamma_{T_p} \sum_{i=1}^{N_{cyl}} (T^m_{p,i} - T^s_{p,i})^2 + \\ & \gamma_u (u^m_{out} - u_{out})^2 \\ \text{subject to} \\ 0 = & F(x, y, p) \end{aligned} \tag{8.25}$$

The problem was solved by a custom-made application coded in C, which is based on the `dsblock` interface for accessing the model description generated by Dymola, and the NLP code IPOPT, see [Wächter and Biegler, 2006], which is dedicated to solving large scale algebraic optimization problems. The software is described in Section 8.7.

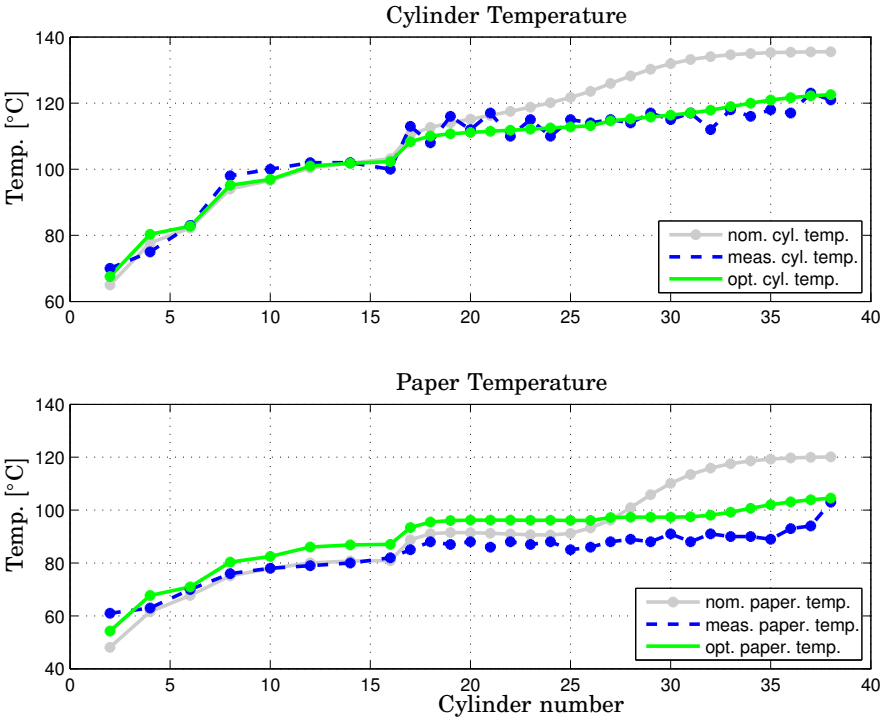


Figure 8.6 Stationary temperature profiles in the case of three parameters.

Parameter Optimization Results

Solving the problem (8.25) yields the optimal temperature profiles shown in Figure 8.6 and the optimal cost 256. For comparison, the nominal profiles, corresponding to the nominal parameter values, are plotted. As can be seen, there is a significantly improved fit between simulated and measured responses. In particular, the output moisture in the nominal case is unrealistically low too early in the dryer section, as can be seen in Figure 8.7. It can also be noted that the fit of the cylinder temperature profile is better than that of the paper temperature profile. This phenomenon is expected, since the weight associated with the paper temperature errors was set to a low value. It can be noted, however, that there is still a temperature mismatch for some cylinders.

In order to further reduce the errors, particularly in the paper temperatures, the number of optimization parameters could be increased. There are two main reasons why additional parameters may achieve a better fit.

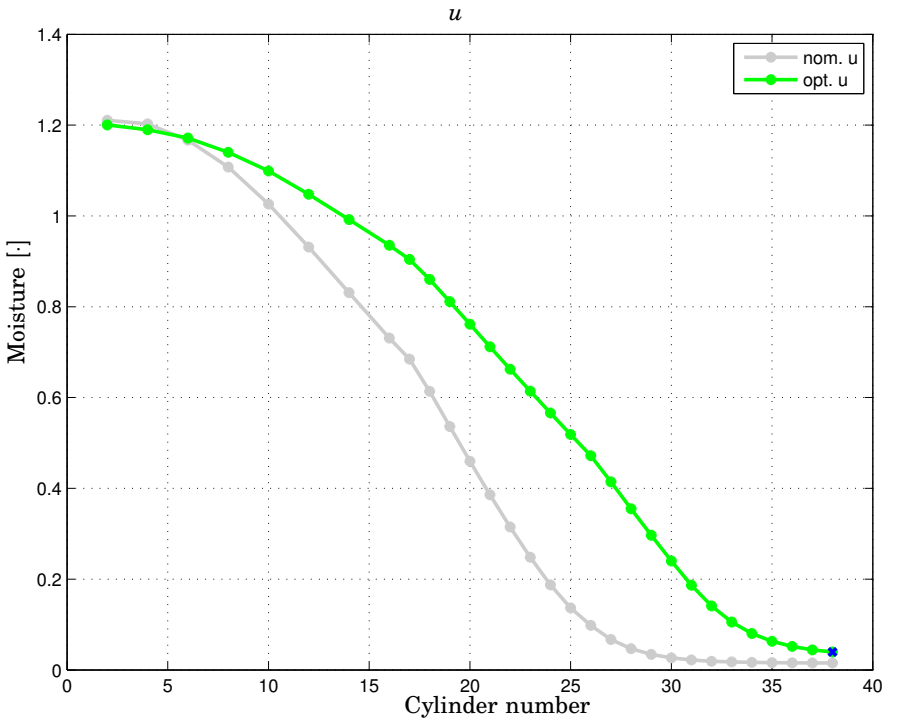


Figure 8.7 The moisture profile. The x-axes shows cylinder numbers.

Firstly, the initial assumption that all cylinders of the dryer section share the same parameter set may not be true. This is because the cylinders are operated under different conditions (pressures, temperatures etc.). Secondly, additional optimization parameters may give improved compensation for unmodeled phenomena, such as, for example, varying air conditions along the dryer section.

In this case, the structure of the model offers a natural way to introduce additional parameters. The cylinders of the dryer section are organized into steam groups, where all cylinders of a group are operated at the same pressure. It is then reasonable to assume that cylinders within a group share the same characteristics, and could share the same parameter set. The model consists of six steam groups, which yields a maximum of 18 optimization parameters.

Now, the introduction of additional parameters may lead to an over-parametrized model. This may lead to a situation where the model is

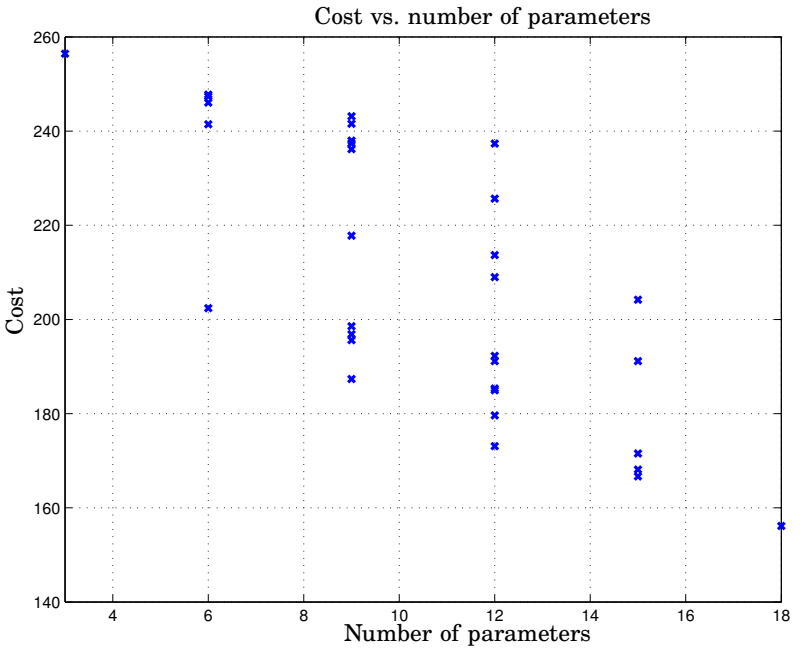


Figure 8.8 Optimal cost as a function of the number of parameters.

fitted to a particular data set, including potential measurement errors. In order to avoid this situation, the marginal benefit of introducing additional parameters should be quantified and analyzed. This has been done by assuming that neighbouring steam groups may share the same parameter set. For example, if we let steam groups one and two, three and four and finally five and six share parameter sets, three parameter sets yielding 9 parameters are obtained. Using this approach, there are 32 combinations, including the extreme cases with three and 18 parameters.

In Figure 8.8 the optimal cost obtained in each of the 32 cases is indicated by (x). As can be seen, the marginal benefit from introducing additional parameters is decreasing. The lowest cost function value, 156, is achieved for the case with 18 free parameter. The optimal temperature and moisture profiles for this case are shown in Figures 8.9 and Figures 8.10.

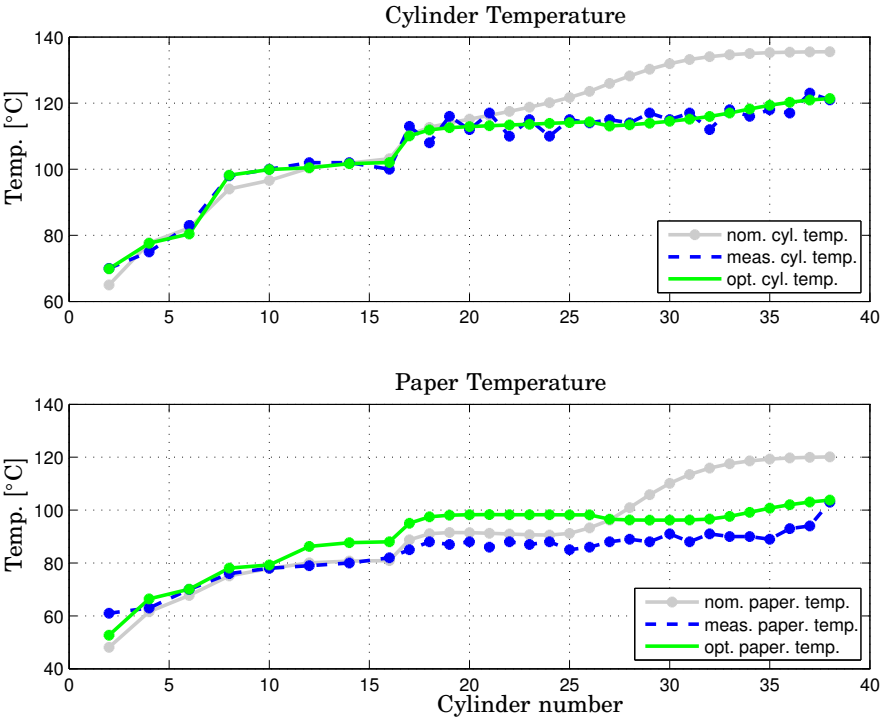


Figure 8.9 Stationary temperature profiles in the case of 18 free parameters.

8.5 Model Reduction

Dryer section models built using DryLib result in large-scale models, even though a sparse discretization scheme for mass and energy balances has been applied. For control design, however, a model describing the dynamic relationship between the inputs and the quality variables at the last free draw is usually sufficient. In practice, low order models (e.g. KLT-models with a gain, a time delay and a time constant) valid at a specific operating point are commonly used for dryer section control. In this section, a reduced model targeted towards moisture control design is developed. Since the output moisture measurement signal available for feedback control is usually obtained at the end of the dryer section, the aim of the reduction scheme is to develop a simpler model, which captures the non-linear dynamical behavior relating the steam pressure reference signal, input moisture, input temperature and dry basis weight (from the press section) to output moisture. Accordingly, accurate simulation of the paper

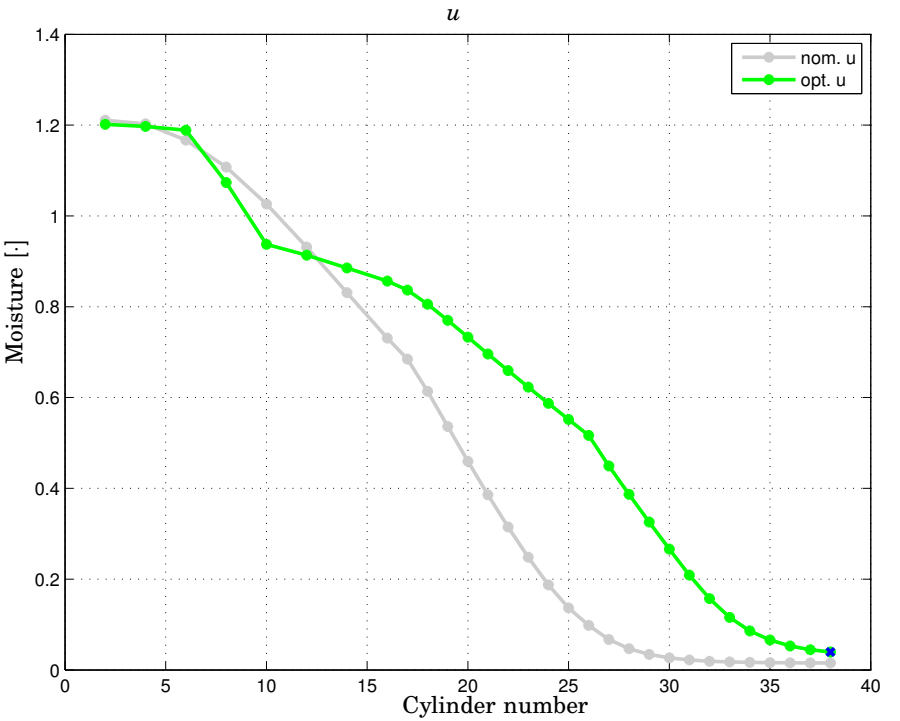


Figure 8.10 Moisture profile. The x-axis shows cylinder numbers.

temperature and the moisture profile can be compromised in order to obtain a lower order model, which describes only the phenomenon of interest, i.e. the behavior of the moisture, accurately.

For linear systems, there exists methods which along with a reduced linear model also gives a bound on the maximum approximation error. The basic approach is usually to find a norm, where it is possible to actually solve the optimization problem resulting from posing a problem where the norm of the difference between the original and the reduced model is minimized. The most common method has historically been that of balanced truncation, where the Hankel norm is used to measure the distance between the models, see [Moore, 1981].

For non-linear systems, however, the situation is different in that there are few methods which offer a structured way of obtaining a lower order model and an upper bound for the approximation error. An additional complication in this case is that the underlying DAE is not easily accessible

for e.g. coordinate transformations which is a common ingredient in model reduction schemes.

In the following, a method based on the equivalent dryer concept and optimization will be presented.

The Equivalent Dryer

In this chapter, the structure of the dryer section will be exploited, in order to obtain a model of lower order. A previously reported concept is that of the equivalent dryer, which is described in [Rao *et al.*, 1994]. Instead of modeling each cylinder as a separate unit, the equivalent dryer concept suggests that one, larger cylinder can be used to approximate an entire steam group. This approach has several attractive features. *i)* It preserves the structure of the dryer section, since each steam group is replaced by its corresponding equivalent dryer, *ii)* each equivalent dryer has an intuitive physical interpretation *iii)* the reduction potential is large, especially for large steam groups.

The Reduction Problem

At the steam group level, the reduction problem can be stated as
"Find the dimensions of ***one*** *steam cylinder, including associated incoming and outgoing free draws and contact paper, which approximates as well as possible, the behavior of a given steam group"*.

This qualitative objective needs, however, to be quantified, and specifically, the meaning of "as well as possible" should be given a mathematical interpretation. In principle, it should be possible to adopt the scheme commonly used for linear model reduction. The problem can then be stated as to minimize the maximum approximation error over the physical dimensions of the equivalent dryer cylinder. Solving this problem involves finding the solution to a dynamic optimization problem, where the search space consists of *i)*, the inputs (for generating the maximum error) and *ii)* the physical dimensions of the equivalent dryer. Since the dryer section model is very large, this approach does not seem attractive. Instead, a method based on physical insight will be used to formulate a tractable, yet challenging, reduction problem.

It is reasonable to assume that the main time constant of the steam group model is dominated by the dynamics of heating the cast iron cylinders. Clearly, the mass of the paper is neglectable compared to the mass of the steam cylinders. This means that when a reference step is applied to the pressure control loop of a cylinder group, the changes in the temperature of the paper sheet will be closely related to the temperature of the cylinder shell. Consequently, since the drying process is driven by the heat transferred from the cylinders to the paper, it is reasonable to assume

that associated variables, most importantly moisture, will be governed by the same time constant.

In line with this reasoning, we suggest that the dynamic and stationary response of the equivalent dryer cylinder may be treated separately. As for the dynamics, we assume that the mass and volume of the equivalent cylinder can be set to N_{cyl}^{g} times those of an individual cylinder in the steam group, where N_{cyl}^{g} is the number of cylinders in the group. Simulation experiments reveal that the time constant of an equivalent cylinder, constructed based on this assumption, corresponds well to the time constant of the full steam group. However, the same result does not seem to hold for the stationary gains, where there is a significant mismatch. Intuitive ways to set the lengths of the free draw and contact papers, using the same reasoning as for mass and volume, do not produce acceptable results. A more sophisticated way of finding the physical dimensions and parameters is thus necessary.

Reduction of One Steam Group

A static model for a paper sheet in contact with a steam cylinder, can be formulated using algebraic versions of the dynamic mass and energy balances presented in Section 8.2. Assuming that the steam pressure, p, the input paper moisture, u_{in}, the input paper temperature, $T_{p,in}$, and the dry basis weight, g, are fixed and known, a system of five equations and five unknown can be derived. The unknowns of the system of equations are the energy flow from the cylinder to the paper, Q_p, the cylinder metal temperature, T_m, the mass flow of steam into the cylinder, q_s, the paper temperature, T_p and the paper moisture, u. The system of equations is then given by

$$
\begin{aligned}
q_s &= \frac{\alpha_{cp} A_{cyl} \eta \alpha_{sc} (T_s - T_p)}{\alpha_{sc} h_s - \alpha_{sc} h_w + \alpha_{cp} \eta h_s - \alpha_{cp} \eta h_w} \\
Q_p &= \frac{\alpha_{cp} A_{cyl} \eta \alpha_{sc} (T_s - T_p)}{\alpha_{sc} + \alpha_{cp} \eta} \\
T_m &= \frac{\alpha_{cp} \eta T_p + T_s \alpha_{sc}}{\alpha_{sc} + \alpha_{cp} \eta} \\
0 &= d_y v_x g u_{in} - A_{xy} q_{evap} - d_y v_x g u \\
0 &= d_y v_x g (1 + u_{in}) C_{p,p,in} T_{p,in} \\
&\quad - A_{xy} q_{evap} (\Delta H_{vap} + \Delta H_s) - \alpha_{pa} A_{xy} (T_p - T_a) \\
&\quad - d_y v_x g (1 + u) C_{p,p} T_p + Q_p
\end{aligned}
\tag{8.26}
$$

were $C_{p,p,in}$, ΔH_s and q_{evap} are functions of the unknowns T_p and u.

In a similar way, a static model for a paper sheet in the free draw can be formulated. The model is given by a system of equations containing

two equations and the two unknowns u and T_p,

$$\begin{aligned} 0 &= d_y v_x g u_{in} - 2A_{xy} q_{evap} - d_y v_x g u \\ 0 &= d_y v_x g (1 + u_{in}) C_{p,p,in} T_{p,in} \\ &\quad - 2A_{xy} q_{evap} (\Delta H_{vap} + \Delta H_s) - 2\alpha_{pa} A_{xy} (T_p - T_a) \\ &\quad - d_y v_x g (1 + u) C_{p,p} T_p. \end{aligned} \tag{8.27}$$

These systems of equations can then be put together to formulate a static model for a steam group.

As stated in the introduction of this section, the most important quality variable, at least for moisture control, is paper moisture. Therefore, a reasonable objective is to minimize the deviation between the moisture in the last free draw of the cylinder group, and the moisture in the outgoing free draw of the equivalent cylinder. In addition, as a secondary objective, it was decided to minimize the deviation in steam consumption. This objective was added since it may be desirable to limit the steam consumption during moisture control.

Performing this minimization for a single operating point is not sufficient, however. In order to obtain a good fit over a wider operating range, a set of operating cases was introduced, over which the optimization was performed. Each case consists of a specification of the operating point in terms of steam pressure, input moisture, input temperature and basis weight. The cost function to be minimized, can now be be written as

$$J = \sum_{i}^{N_c} \gamma_u (u_{out,i} - u^r_{out,i})^2 + \gamma_{q_s} \left(\sum_{j}^{N_{cyl}} q_{s,j,i} - q^r_{s,i} \right)^2, \tag{8.28}$$

where $u_{out,i}$ is the output moisture of the steam group in the i:th case, $u^r_{out,i}$ is the corresponding output moisture of the equivalent cylinder, N_c is the number of cases and N_{cyl} is the number of cylinders in the group. As for the steam flow, the squared sum of deviations between the total steam flow for the cylinder group and the equivalent dryer is penalized. γ_u and γ_{q_s} are weights representing the relative importance of a good match in moisture and steam flow respectively. The minimization of the criterion (8.28) is performed subject to the equations (8.26) and (8.27), which are repeated based on the number of cases, N_c, and the number of cylinders in the group N_{cyl}.

It remains to define the optimization parameters, over which the minimization of (8.28) is performed. Six parameters of the equivalent dryer were selected for optimization, namely the length of the free draws, the length of the contact paper, the heat transfer coefficient between steam

and condensate, α_{sc}, the convection coefficient, α_{pa}, and the mass transfer coefficient, K. The number of variables that are actually needed to obtain a good fit is not unambiguous, however. For small steam groups, or if few cases are used, some of the suggested optimization variables may well be fixed, without any increase in the approximation error. In fact, it is desirable to find an appropriate trade-off between the number of optimization variables and optimization performance, in order to avoid over-parametrization, as discussed previously.

Reduction of a Dryer Section

A straight forward approach for deriving a reduced order dryer section would be to simply apply the method described in the previous section for each individual steam group. Recalling our main objective, which is to predict the moisture in the last free draw, this approach would not explore the full potential of the method. Instead, a larger optimization problem, incorporating all groups, may be formulated where most attention is given to minimizing the deviation of the last group. This means that *all* groups are reduced at the same time, and that the full reduction potential is used according to the main objective, which is to predict the output moisture. It may, however, be advantageous to include the deviations, with small weights, of all groups in the optimization criterion, in order to avoid a physically unrealistic model.

Initial optimization runs showed that the total length of the paper process in the reduced model was significantly larger than the total length of the paper process in the original model. This deficient of the reduced model may be suppressed by introducing a term in the optimization criterion penalizing the deviation of total paper process length between the original and the full model. This modification of the original problem resulted in a better match of the dynamic response, without a penalty in terms of degraded static match.

An additional modification of the problem concerning the matching of the steam flow rate was made in the final formulation. Since the *total* steam consumption of the dryer section is of interest, rather than the consumption of individual groups, the penalties on deviations in steam flows at the group level was replaced by single penalty on deviations in the total steam consumption.

The overall performance criterion can now be written

$$J_{tot} = \sum_{i=1}^{N_c} \sum_{k=1}^{N_g} \gamma_{u,k} (u_{out,i,k} - u^r_{out,i,k})^2 + \gamma_{q_s} \left(\sum_{k=1}^{N_g} \sum_{j=1}^{N_{cyl,k}} q_{s,i,j,k} - \sum_{k=1}^{N_g} q^r_{s,i,k} \right)^2 + \gamma_l \left(\sum_{k=1}^{N_g} \sum_{l=1}^{N_{p,k}} l_{p,k,l} - \sum_{k=1}^{N_g} \sum_{l=1}^{3} l^r_{p,k,l} \right)^2 \tag{8.29}$$

where N_g is the number of groups, $N_{p,k}$ is the number of paper segments in group k, $l_{p,k,l}$ is the length of a paper segment in the original model and $l^r_{p,k,l}$ is a length of a paper segment in the reduced model. In line with the arguments given above, $\gamma_{u,N_g} >> \gamma_{u,k}, k \neq N_g$.

Solving the Optimization Problem

The resulting algebraic optimization problem is challenging, both due to its size and its non-linear character. The final problem consists of 9536 free variables and 9504 equality constraints, of which 8568 are non-linear. Efficient solution of large scale NLP problems of this type requires state of the art numerical algorithms, exploring the sparse structure of the problem as well as analytical Jacobian and Hessian information.

The problem definition was programmed in AMPL, which is a language for mathematical programming, [Fourer *et al.*, 2003]. AMPL enables encoding of linear and non-linear algebraic optimization problems, using optimization-oriented language constructs. The problem description, i.e. the AMPL code, is then executed within the AMPL tool, which in turn interfaces several numerical solvers. In this application, the NLP code KNITRO, [Waltz, 2005], has been used. The combination of AMPL and KNITRO is extremely powerful, since the AMPL interface to numerical solvers offers analytic evaluation of Jacobians and Hessians as well as sparsity information. This enables KNITRO to operate in its most efficient mode, resulting in acceptable execution times also for large systems. The reduction problem formulated in the previous section is solved in about 2-5 minutes, depending on configuration and initial starting point.

The proposed method has the distinct drawback of requiring complete re-encoding of the the model description. This was necessary, however, in order to enable utilization of the appropriate symbolical and numerical algorithms.

It is important to note, however, that the problem is non-convex, and that only local optimality can be expected. However, in this case, the solu-

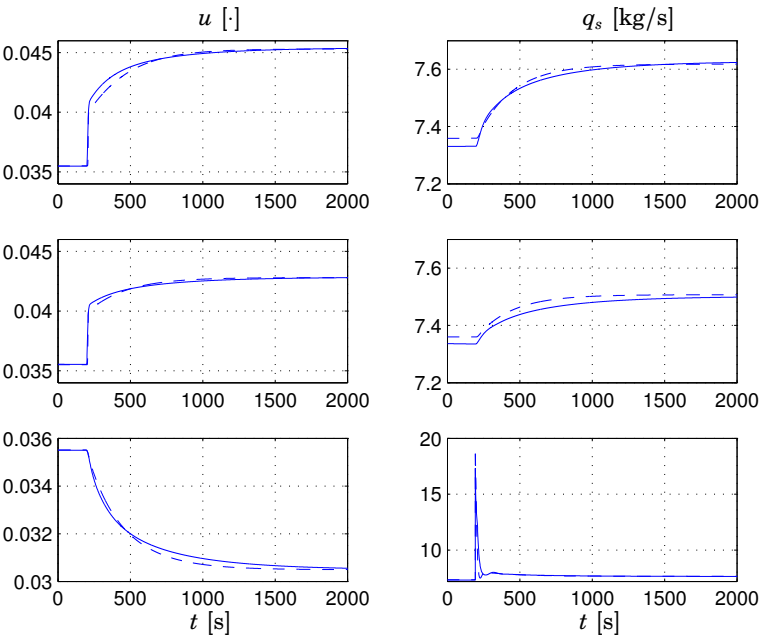


Figure 8.11 Step responses for moisture (left) and steam flow rate (right) of the original (dashed) and the reduced (solid) models. The responses corresponds to, from above, steps in input moisture, input dry basis weight and pressure reference, applied at 200 s.

tion to the reduction problem seemed to be robust with respect to different starting points. Also, the obtained solution is reasonable in the sense that the optimized parameter values lie within physically feasible limits.

Model Reduction Results

As mentioned above, a set of operating conditions need to be specified, in order to complete the problem formulation. Clearly, the operating range over which the reduced model is valid, is influenced by this choice. As the nominal case, values for steam pressures, input moisture, input temperature and dry basis weight corresponding to a typical grade were chosen. Based on the nominal case, 35 additional cases were defined by varying the nominal input parameters.

The result of the reduction procedure was evaluated by means of step responses in input moisture, dry basis weight and pressure set point, see Figure 8.11. As can be seen, there is a good match between the stationary

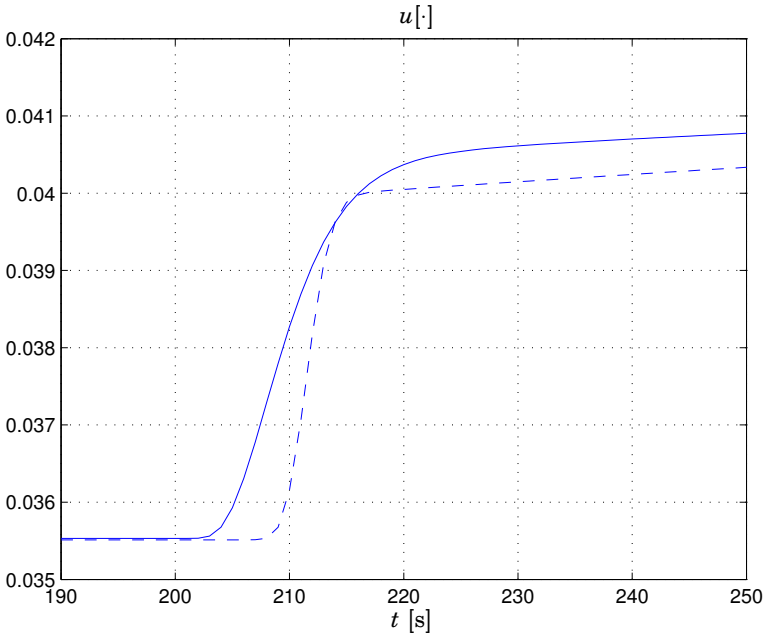


Figure 8.12 Response in output moisture, resulting from a step disturbance in the input dry basis weight. The dashed curve corresponds to the original model and the solid curve corresponds to the reduced model. The step disturbance is applied at t=200 s.

responses of the original and reduced models. Also, the (slow) dominating time constant is captured well by the reduced system. However, the reduced model does not fully capture the fast transient behavior of the original model. The steep initial ascent in the step response, which is shown in Figure 8.12, is due to the transport delay of the model. Using the length of the paper web through the dryer section and the machine speed, the theoretical transport delay can be calculated. In this particular case, the delay is 12s, which is matched well by the original model, represented by the dashed curve. The reduced model, on the other hand, seems to have a smaller delay but a more smoothed initial response. This is, however, to be expected. The original model consists of a large number of paper components, which together forms a high-dimension compartment system. The reduced model consists of significantly fewer segments, and cannot approximate the time delay with the same accuracy.

The original motivation for performing the model reduction was to

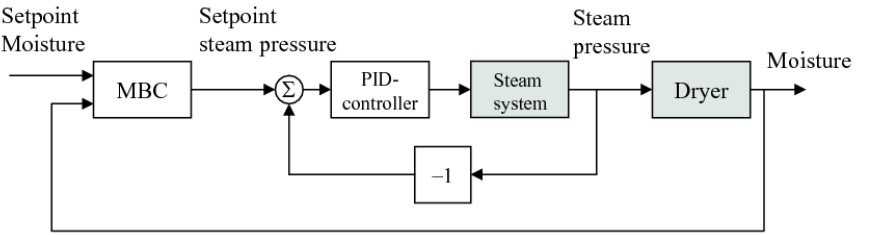


Figure 8.13 The moisture control cascade loop. The inner loop controls the steam pressure by manipulating a steam valve. The inner loop generally consists of a PID-controller and it gets its set point from some type of Model Based Control (MBC), commonly an IMC (Internal Model Control), a Dahlin controller, or a linear MPC (Model Predictive Control).

obtain a model of lower complexity. Indeed, the reduced model has fewer dynamical states, namely 85, as compared to 318 for the original model. Also, the simulation time for a typical scenario was approximately 85% shorter for the reduced model.

8.6 NMPC of Output Moisture

The structure of the moisture control loop is depicted in Figure 8.13. It is usually the case that all cylinder groups are tied to the same steam pressure set point which gives a single loop cascade control. It is common to let the MBC controller calculate the pressure set-point for the last steam group, p^{sp}, and then calculate the pressure set-points for the other groups as functions of p^{sp}. Using this method, it is straight forward to ensure that there is an monotonically increasing pressure profile in the machine direction, which is important since steam from high pressure groups is re-used to heat groups running at lower pressure.

The MBC controller is usually based on a low-order linear model of the dryer section. While a well tuned controller works well at a given set-point, the non-linear character of the dryer section dynamics results in degraded performance if the set-point is changed. Since the plant is operated at several different set-points, corresponding to different grades, a traditional control system maintains several parameter sets for the MBC controller. Switching of controller parameters is then done after a grade change. In other cases, the MBC may be tuned to fit all grades of the specific machine.

In this section, a different approach to moisture control is considered. Based on the reduced non-linear dryer section model derived in Section 8.5, a basic Non-Linear Model Predictive Control (NMPC) scheme is implemented. The main benefit of using a non-linear model in the control design is that the operating range of the controller may be increased. In addition, successful implementation of a controller which achieves good performance in a wide operating range may serve as a unifying strategy for stationary and transition (grade change) control, whereas common practice today is to use separate controllers for these two control modes.

A realistic implementation of an MPC controller consists of three main parts—reference target calculation, state estimation and solution of the optimal control problem. In this section, the problem of solving the optimal control problem is addressed. The resulting controller is evaluated under the assumption of full state information.

Model Predictive Control

MPC refers to a family of controllers which are based on the receding horizon principle. At each sample, a finite horizon open loop optimal control problem is solved, and the first part (corresponding to the first sample) of the resulting optimal control profile is applied to the plant. At the next sample, the procedure is repeated and a new optimal control problem with the horizon shifted one sample is solved. Thereby the name receding horizon control. Two of the most important advantages of using MPC is that it works well for MIMO plants and that it takes state and control bounds into account *explicitly*. However, an MPC controller, including the on-line solution of an optimization problem, is computationally demanding, which makes application to processes with fast dynamics troublesome. During the last decade, MPC has emerged as a major control strategy, mainly in the process industry, see [Qin and Badgwell, 2003] for an overview.

MPC comes in many flavors. The theory for MPC based on LTI models is well developed. The linear case has particularly attractive features in that the arising optimization problem is convex, and the availability of stability results. For non-linear systems, the situation is somewhat different. While stability results exist, see [Mayne *et al.*, 2000], the problem of solving the arising optimal control problem is complicated, since the problem is in general non-convex, which means that global optimality cannot be guaranteed. Still, several algorithms exist, and non-linear MPC has received increased industrial interest during the last few years.

Dynamic Optimization

Optimization of Dymola models has previously been considered in the work [Franke *et al.*, 2003], where the Simulink interface (the S-function interface) provided with Dymola was used to access the model. The main

difference between the approach used in [Franke *et al.*, 2003] and this work lies in the methods of accessing the model. In this work, the `dsblock` interface which is also provided with Dymola has been used. The `dsblock` and S-function interfaces offers essentially the same functionality.

In order for a (gradient based) NLP algorithm to have fast convergence, it is important to provide to the algorithm not only the cost function, but also its gradient with respect to the optimization parameters. Calculation of high accuracy gradients for dynamical systems generally involves calculation of the state sensitivities with respect to parameters. This can be done by integration of the sensitivity equations. By utilizing that the sensitivity equations have the same Jacobian as the original DAE, integration can be done efficiently.

The algorithm used to solve the dynamic optimization problem described in this section is a straight forward implementation of a sequential single shooting algorithm, see [Vassiliadis, 1993].

State Estimation

There are two main approaches to state estimation for non-linear systems, namely the extended Kalman filter, [Anderson and Moore, 1979], and receding horizon estimation, [Rao *et al.*, 2003]. The problem is challenging, and also computationally demanding, especially for large systems. Solution of the state estimation problem is not treated here, although the problem must be solved in order to apply the MPC scheme to a real plant. Accordingly, the same model is used both in the NMPC controller and for simulating the system.

The Optimal Control Problem

An integral part of an NMPC controller is the formulation of the open-loop optimal control problem to be solved in each sample. Since the aim of the control scheme in this application is to control the moisture ratio, it is natural to penalize deviations from the target moisture. The control trajectory in the optimization problem, p^{sp}, is parametrized by a piece-wise constant function with N_u segments. In order to avoid violent control moves, which may introduce disturbances in the steam system, a term penalizing the deviation between two successive control moves is introduced in the cost function. In addition, there are hard limits acting on the control variable.

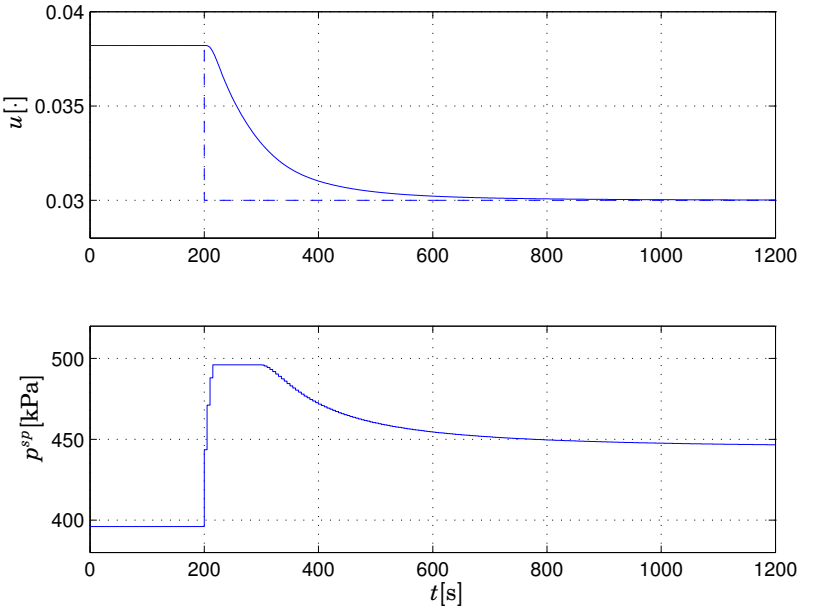


Figure 8.14 Step response of the NMPC controller.

This yields the following optimization problem

$$\min_{\hat{p}_i^{sp}} \int_0^{T_f} \gamma_u (u_{out}^{sp} - \hat{u}_{out}(t))^2 dt + \sum_{i=0}^{N_u-1} \gamma_p (\Delta \hat{p}_i^{sp})^2$$

subject to (8.30)

$$F(x, \dot{x}, y, p^{sp}) = 0 \text{ (DAEdynamics)}$$

$$466 \text{ kPa} \leq p^{sp} \leq 596 \text{ kPa (control constraint)}$$

where T_f is the prediction horizon, u_{out}^{sp} is the target moisture, $\hat{u}_{out}(t)$ is the predicted moisture profile, $\hat{p}_i^{sp}$ is the predicted pressure set point trajectory and $\Delta \hat{p}_i^{sp} = \hat{p}_i^{sp} - \hat{p}_{i-1}^{sp}$. γ_u and γ_p are weights. In the simulation, the parameters were set to $\gamma_u = 10000$, $\gamma_p = 0.01$, $N_u = 4$, and the sampling interval was set to 5 s.

Results

A simulation where the NMPC controller is applied to the reduced dryer section model is shown in Figure 8.14. In the simulation, a reference step,

from $u_{out}^{sp} = 0.038$ to $u_{out}^{sp} = 0.03$ is applied at $t = 200$ s. As can be seen, the moisture reaches the desired set-point, while the control signal respects the specified constraints.

An important, and often limiting, factor when using MPC controllers, is the execution time for solving the on-line optimization problem. In this case, execution times ranged from 10 s to 80 s, with a mean of 13.5 s. Typically, execution times are longer when reference changes and disturbances occur, while shorter and more predictable execution times are obtained during stationary operation. Assuming a sampling interval of $h = 5$ s, it is clear that the execution times must be decreased. There are several approaches to reducing execution times, e.g. modifying the lengths of the control and prediction horizons, reducing the complexity of the model or using a more efficient optimization algorithm.

8.7 Software Tools

The dryer section model has been implemented, as mentioned above, in Modelica and Dymola. The parameter optimization problem and the NMPC problem, however, were solved by integrating several software packages into a custom application, which utilized the C-code generated by Dymola, representing the model.

The software packages used in the development of the custom application are:

- a C programming interface to access functions generated by Dymola, `dsblock`. Using this interface, custom applications can be developed for, for example, simulation or, like in this case, optimization. The interface provides basic routines for obtaining information about model parameter and initial state values, evaluation of the right side of the resulting ODE (DAE) and the associated Jacobian.
- a DAE-solver, DASPK 3.1 [Maly and Petzold, 1996]. This code solves DAE:s as well as calculates sensitivities required for optimization. The code is written in Fortran and was translated to C using `f2c`.
- an NLP-code, IPOPT [Wächter and Biegler, 2006]. This code implements a primal-dual interior point method and was used to solve the NLP resulting from the parameter optimization and NMPC problems.
- a package for managing the communication between the Dymola C interface and DASPK, which has been developed in order to enable simplified development of optimization applications based on models generated by Dymola. This package, in the following referred to as

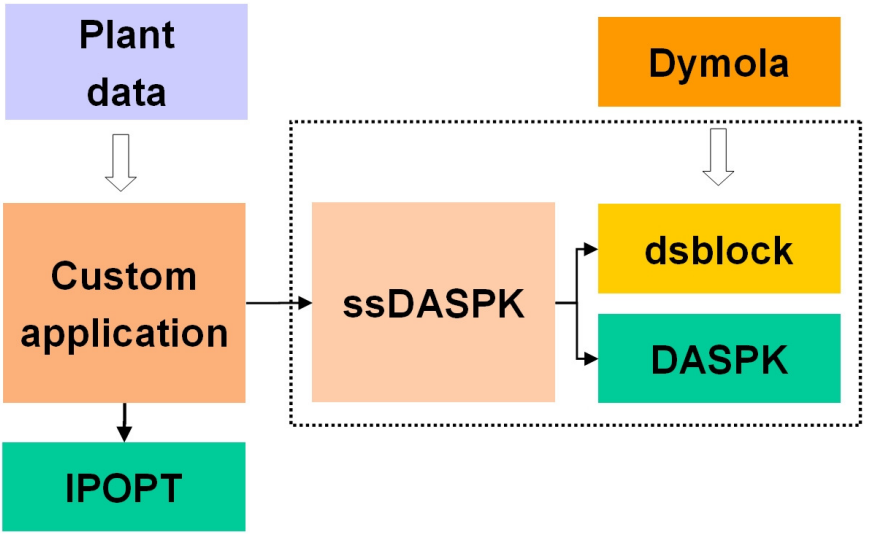


Figure 8.15 Software application structure.

ssDASPK, provides e.g. simulation and sensitivity calculation for use in custom applications.

These packages were compiled and linked with the code representing the model generated by Dymola, into an application which was used to set up and solve the particular optimization problems. The structure of the applications is shown in Figure 8.15.

The main drawback with using the software described in this section is that the formulation of the optimization problem needs to be encoded in a low level language, such as C, for each particular problem. Much effort is thus needed to encode the problem, whereas, as was argued in Chapter 7, high-level description languages enable the user to rather focus on formulation of the problem. It would, however, be possible to use ssDASPK as a back-end in the Optimica compiler. Such a back-end would then typically generate C-code corresponding to the "Custom application" block in Figure 8.15.

8.8 Summary and Conclusions

In this chapter, modeling, model reduction, parameter optimization and NMPC control design for a paper machine dryer section have been considered. It has been demonstrated how Modelica models of high complexity can be used for purposes other than simulation. The resulting optimization problems are challenging and require state of the art numerical solvers. In particular, solution of the model reduction problem, which has more than 9000 free variables, is dependent on algorithms exploring the problem structure. Our experience from this project is that there is no single tool or software that can address all problems arising in simulation and optimization. Rather, in order to solve problems effectively, it is essential that Modelica tools are designed to be interfaced with software for solution of optimization problems. In general, it is highly desirable that software for complex systems is provided with interfaces so that they can be combined.

There are several possible extensions of the work presented in this chapter. The DryLib library may be extended as outlined in Section 8.3, and the parameter optimization scheme would benefit from including also time series data. Regarding the model reduction scheme, it may be desirable to derive models with further reduced complexity valid over a wide operating range. Finally, the NMPC scheme outline in Section 8.6 needs to be further elaborated in order to be applicable to the real plant.

9

Start-up Control of a Plate Reactor

9.1 Introduction

Start-up control of chemical reactors has been an area of research for many years. The main focus has been on batch and semi-batch reactors. Start-up of continuous reactors is similarly of great interest, since the risk of incidents are much higher during start-up than during steady state operation. At low temperature, almost no reaction occurs due to the temperature sensitive reaction rate. To start the reaction, heat must be provided into the reactor system, typically through the feed flows or the cooling water. At some reactor temperature, the reaction reaches the ignition point. The heat release from a strongly exothermic reaction leads to self-acceleration and the reactor temperature quickly increases to the nominal operating point, or, if care is not taken, above.

The transient from initial conditions to an optimal operating point and the temperature at which ignition occurs are highly nonlinear functions of the inputs and the states of the system. Therefore, the system is very sensitive to small changes in reactor inlet conditions or variations in physical parameters. Previous studies have been focusing on developing criteria to detect operating regions with parametrically sensitive behavior, see e.g. [Bauman *et al.*, 1990] or [Varma *et al.*, 1999]. However, the studies are often limited to finding non-sensitive operating points for steady-state and the impact of a feedback control system is often neglected. In [Zaldívar *et al.*, 2003], a general criterion to define runaway limits for tank reactors is presented, where the effect of feedback control on the runaway boundaries is briefly discussed. In general, feedback control can reduce the impact of model uncertainty, but due to actuator limitations present

in all industrial systems, the available bandwidth may not be sufficient to guarantee a safe start-up in regions with parametrically sensitive behavior.

The focus in this chapter is to develop methods for formulating the optimization problem to achieve safe, robust and efficient start-up of an exothermic reaction. With actuator limitations, it is necessary to consider the interplay between open-loop optimal start-up trajectories and feedback control. By studying the parametric sensitivity, the start-up optimization problem can be formulated so that the optimal solutions have reduced sensitivity to uncertainty. This facilitates the task of the feedback controller to maintain safe operation despite its actuator limitations.

In [Hahn *et al.*, 1971], open loop optimal start-up trajectories for a tubular reactor are computed, based on a distributed maximum principle for a given optimal steady-state operating point. The reaction is exothermic and reversible. The reactor temperature and thus the yield are controlled by manipulating the reactor wall temperature with a constraint on the maximum reactor temperature. The optimal control trajectory is of bang-bang type with a singular arc to the steady-state. The study in [Hahn *et al.*, 1971], however, does not consider uncertainties or dynamic limitations in the actuator, and does not consider any closed loop feedback control.

In [Verwijs *et al.*, 1996], the start-up and safeguarding of an adiabatic tubular reactor system is considered. There, open loop trajectories of the manipulated variables are calculated by a generalized-reduced-gradient optimization. The safeguarding is realized through plant start-up rules e.g. minimizing the total amount of unreacted chemicals exiting the reactor during the start-up period. High levels of unreacted chemicals in the reactor outlet may lead to continued reaction and heat release in storage tanks. Without adequate cooling systems in these tanks, this temperature increase may start by-product formation and lead to a thermal runaway. During start-up, the safeguarding is implemented by monitoring the difference between the actual response of thermoelements and the optimal trajectories calculated in the model-based optimization. When the difference exceeds some limit, the reactor should be brought to shutdown to prevent the process from running into a hazardous situation state.

In this chapter, start-up control of the newly developed Alfa Laval Plate Reactor is considered, see [Alfa Laval AB, 2006] and [Prat *et al.*, 2005]. This type of reactor is conceptually a combination of a tubular reactor and a plate heat exchanger. The key concept is to combine efficient micromixing with excellent heat transfer into one operation. The plate reactor has a flexible reactor configuration, where reactants can be injected at multiple points along the reactor length to enhance the reactor performance. There may also be individual cooling zones inside the reactor.

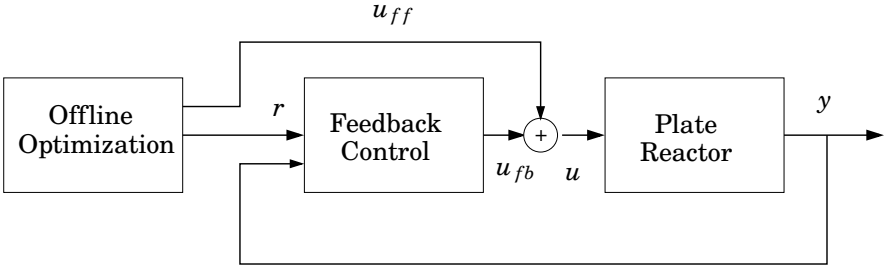


Figure 9.1 General block diagram for the start-up control with off-line optimization and online feedback control.

These design features enable improved temperature control. Better process design and control possibilities allow higher operating temperatures, thus reducing the reaction time. Reactants of higher concentration can also be used. This type of process intensification enables a plate reactor to replace larger semi-batch reactors and significantly reduces the energy consumption. Start-up and transition control is more important for this type of process, as the plate reactor may be operated in a more flexible production schedule for fine chemicals or pharmaceuticals syntheses. In contrast to the work in [Hahn *et al.*, 1971] and [Verwijs *et al.*, 1996], the plate reactor is equipped with a cooling system and multiple injection points for the reactants. Feedback control is used to increase safety and robustness in the presence of uncertainty.

In this chapter, off-line dynamic optimization of the start-up trajectories is considered. The optimal trajectories are used as feedforward and set-point terms in a local feedback system, see Figure 9.1, which should ensure that the optimal trajectories are tracked also in the presence of disturbances and modeling errors. In [Haugwitz *et al.*, 2007], methods for modeling and control for stationary operating conditions of the plate reactor are presented. The approach presented in this chapter extends that work by proposing a method addressing the start-up problem for the plate reactor.

The optimal control problem formulated in this chapter has been solved by means of the Optimica compiler, which transcribes optimal control problems using a simultaneous method. For an overview of dynamic optimization and simultaneous methods, see Chapter 3, and for the presentation of Optimica, see Chapter 7.

Dynamic optimization can quite easily give high performance solutions based on a nominal model. A greater challenge is to find solutions with robust performance, so that the solutions have reduced sensitivity to model uncertainty. In [Diehl *et al.*, 2006], an approximate technique is presented

for robust nonlinear optimization, which utilizes a linearization of the uncertainty set. The main contribution is two methods to preserve sparsity to achieve efficient computation for large scale problems.

The robustness of the optimal solutions are in this chapter analyzed in terms of parametric sensitivity. By introducing two key specifications in the optimization formulation, high-frequency penalties on the control inputs and state constraints on one of the chemical reactants, the sensitivity of the optimal solutions can be significantly reduced, which increases the robustness of the closed loop start-up control problem. The robustness is introduced based on process insight and the extra specifications are only a small addition to the computational complexity. The robustness of the proposed method is verified in Monte-Carlo simulations, where the values of the parametric uncertainties are randomly generated from a uniform distribution.

The chapter is organized as follows. The plate reactor is briefly presented in Section 9.2. Section 9.3 outlines the specific problem formulation of the reactor start-up. Section 9.4 shows how the actual optimization problem is stated and the results of the optimization. The implementation of the closed loop control is presented in Section 9.5 and simulation results are given in Section 9.6. The Monte Carlo simulations are described in Section 9.7. The chapter ends with conclusions in Section 9.8.

9.2 The Plate Reactor

The Alfa Laval Plate Reactor, [Alfa Laval AB, 2006], originates from a plate heat exchanger, but re-designed to improve micro-mixing. It consists of a number of reactor plates, where the reactants mix and react. On each side of a reactor plate there is a cooling plate, through which cold water is circulated.

Between the inlet and the outlet, special inserts inside the reactor form flow channels of alternating directions that gives turbulent flows and good mixing of the reactants. The concept relies on a flexible reactor configuration. The type of inserts and the number of rows inside the reactor, which determine the residence time, can be adjusted, based on the type and rate of the chosen reaction. Multiple injections and independent cooling zones enable the reactor to be tailor-made for any complex reaction, e.g. multistage reactions. Temperature sensors can be arbitrarily mounted inside the reactor, specifically after each injection point.

A second order temperature-dependent exothermic reaction is considered, where C is the product of interest (9.1). Product D has a constant

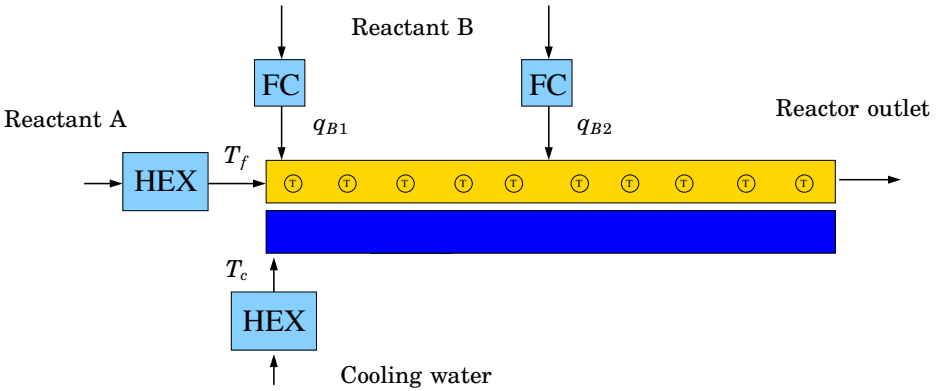


Figure 9.2 The reactor shown as a schematic tubular reactor. There are four inflows to the process and there is one manipulated variable for each inflow; q_{B1}, q_{B2}, T_f and T_c. Each inflow has an actuator subsystem that provides flow control (FC) or temperature control through heat exchangers (HEX). The circles with T represent internal temperature sensors.

stoichiometric relation with C, thus in the sequel $c_C \equiv c_D$.

$$A + B \rightarrow C + D + \text{heat} \tag{9.1}$$

In Figure 9.2, the plate reactor is schematically illustrated as a tubular reactor. The primary reactant A enters from the left. The secondary reactant B is injected at the inlet and in the mid-section of the reactor. With multiple injection points, the heat release from the exothermic reaction can be distributed along the reactor, thus improving the productivity despite constraints on the reactor temperature. A heat exchanger pre-heats the reactant feed A and another heat exchanger cools the cooling water to desired temperatures. Two flow control loops ensure that the desired amount of B is fed into the reactor.

Modeling

A model of the plate reactor can be derived from first principles for heat transfer, reaction kinetics, mass, energy and chemical balances, see e.g. [Froment and Bischoff, 1990]. The reaction kinetics can be approximated with the Arrhenius law. The plate reactor can, from a modelling perspective, be approximated as a continuous tubular reactor with multiple inlet ports of reactant B along the reactor. The distributed nature of the process leads to five partial differential equations (PDEs) for the reactor temperature T_r, the cooling water temperature T_w and the concentrations for the reactants and products, c_A, c_B, and c_C, see e.g. [Hangos and Cameron,

2001],

$$\frac{\partial T_r}{\partial t} = \mathcal{D}_e \frac{\partial^2 T}{\partial z^2} - v_r \frac{\partial T_r}{\partial z} - \frac{4h}{d_r \rho c_p}(T_r - T_w) + \frac{\Delta H_r}{\rho_r c_p} r \tag{9.2}$$

$$\frac{\partial T_w}{\partial t} = - v_w \frac{\partial T_w}{\partial z} + \frac{4h}{d_w \rho c_p}(T_r - T_w) \tag{9.3}$$

$$\frac{\partial c_A}{\partial t} = \mathcal{D}_m \frac{\partial^2 c_A}{\partial z^2} - v_r \frac{\partial c_A}{\partial z} - r \tag{9.4}$$

$$\frac{\partial c_B}{\partial t} = \mathcal{D}_m \frac{\partial^2 c_B}{\partial z^2} - v_r \frac{\partial c_B}{\partial z} - r \tag{9.5}$$

$$\frac{\partial c_C}{\partial t} = \mathcal{D}_m \frac{\partial^2 c_C}{\partial z^2} - v_r \frac{\partial c_C}{\partial z} + r \tag{9.6}$$

$$r = k_0 e^{\frac{E_a}{RT_r}} c_A c_B. \tag{9.7}$$

The reaction rate r is described by the Arrhenius law and depends on the pre-exponential factor k_0, the activation energy E_a, the universal gas constant R, the reactor temperature T_r and the two reactant concentrations c_A and c_B. The variable z is the position along the reactor flow channel, d_r is the reactor tube diameter, d_w is the cooling jacket tube diameter and h is the heat transfer coefficient between reactor fluid and water. The variables v_r and v_w represent the fluid velocity through the reactor and the cooling jacket, respectively. ΔH is the reaction energy term. The density and the specific heat capacity are denoted by ρ and c_p. $\mathcal{D}_e$ and $\mathcal{D}_m$ are the energy and mass diffusion coefficients, respectively. The very high flow rate of the cooling water means that the diffusion term for T_w can be neglected.

The PDEs are approximated using the Method-of-Lines, [Schiesser, 1991], and the finite volume method. The spatial derivatives are approximated with a first order backward difference method yielding a system of ordinary differential equations (ODEs). The dispersion of the reactor is approximated by the tanks-in-series model, see [Levenspiel, 1999]. Each PDE is discretized using $N = 30$ control volumes, which is a compromise between accuracy and computational complexity, and gives adequate representation of the dispersion. The equations for the first control volume

of the discretized model can be written as:

$$\frac{dT_{r,1}}{dt} = \frac{q_{feed,A}}{V_r}T_{feed,A} + \frac{q_{B1}}{V_r}T_{feed,B} - \frac{q_{r,1}}{V_r}T_{r,1} \tag{9.8}$$

$$+\frac{hA_{heat}}{\rho c_p V_r}(T_{w,1} - T_{r,1}) + \frac{\Delta H_r}{\rho c_p}r_1 \tag{9.9}$$

$$\frac{dT_{w,1}}{dt} = \frac{q_{cool}}{V_c}(T_c - T_{w,1}) + \frac{hA_{heat}}{\rho c_p V_c}(T_{w,1} - T_{r,1}) \tag{9.10}$$

$$\frac{dc_{B,1}}{dt} = \frac{q_{B1}}{V_r}c_{feed,B} - \frac{q_{r,1}}{V_r}c_{B,1} - r_1 \tag{9.11}$$

$$r_1 = k_0 e^{\frac{E_a}{RT_{r,1}}} c_{A,1}c_{B,1} \tag{9.12}$$

where V_r is the volume of each reactor element, V_c is the volume of each cooling element. A_{heat} and h are the heat transfer area and coefficient, respectively, between the water and the reactor flow. $T_{feed,A}$ is the feed temperature of reactant A. T_c is the inlet temperature of the cooling water. The variable q_r is the flow rate inside the reactor, $q_{feed,A}$ is the feed flow rate of reactant A, q_{B1} is the flow rate of B injected at the first injection point and q_{cool} is the cooling flow rate. There are similar ODEs for the other chemical components c_A and c_C. Thus T_r, T_w, c_A, c_B and c_C are all vectors of size N. The full state vector is defined as $x = [T_r^T \quad T_w^T \quad c_A^T \quad c_B^T \quad c_C^T]^T$ of size $5N$. In addition, the process model includes four states representing the actuator dynamics of the four control inputs presented below.

Inputs and Outputs

Consider a reactor configuration with two injection points for reactant B and one single cooling flow, that is, the same water cools the entire reactor. The two injection points are located at the reactor inlet and mid section, respectively.

Four control variables are used as manipulated variables in the start-up optimization problem, see Figure 9.2.

- The feed flow rates of reactant B, q_{B1} and q_{B2}, at the two injection points. In the sequel, we will use the scaled control variables $u_{B1} = q_{B1}/q_{feed,B}$ and $u_{B2} = q_{B2}/q_{feed,B}$, where the scaling factor $q_{feed,B}$ is the total feed of B at stoichiometric conditions. Thus when $u_{B1} + u_{B2} = 1$, stoichiometric amounts of A and B are being fed into the reactor.
- The inlet temperature of the cooling water T_c. By manipulating the cooling temperature rather than the cooling flow rate, the temperature control will be more effective and include less nonlinearities. The drawback is the additional process equipment needed.

- The feed temperature $T_{feed,A}$ of the reactant A. In the sequel, the shorter form T_f will be used. The feed flow of reactant A constitutes roughly 80% of the total reactor flow. Thus the feed temperature of reactant B has only minor influence on the reactor temperature.

There is one manipulated variable for each input flow of the reactor system, see Figure 9.2. Each manipulated variable has a corresponding actuator system, i.e. each control signal is in fact a set-point to the actuator system. Therefore, in the sequel, $u = [u_{B1,sp}\ u_{B2,sp}\ T_{f,sp}\ T_{c,sp}]^T$ will be used, where the subscript $_{sp}$ denotes set-point. Each actuator system includes a low-level controller ensuring good set-point tracking. The time constants of the actuator dynamics are roughly 1 second for the injection flow control variables u_{B1} and u_{B2}, 2 seconds for the feed temperature T_f and 4 seconds for the cooling temperature T_c. However, the feed temperature and cooling temperature systems are designed for steady-state operation, which means that during the start-up transients these control variables will be less effective due to valve limitations. These limits for the low-level control signals can be approximated by rate limits on T_f and T_c. To summarize, the absolute and rate limits on the control variables are

$$0 \leq u_{B1} \leq 0.8 \tag{9.13}$$
$$0 \leq u_{B2} \leq 0.8 \tag{9.14}$$
$$-2^\circ\text{C/s} \leq \dot{T}_f \leq 3^\circ\text{C/s} \tag{9.15}$$
$$20^\circ\text{C} \leq T_f \leq 90^\circ\text{C} \tag{9.16}$$
$$-2^\circ\text{C/s} \leq \dot{T}_c \leq 1^\circ\text{C/s} \tag{9.17}$$
$$15^\circ\text{C} \leq T_c \leq 90^\circ\text{C} \tag{9.18}$$

Two reactor temperature measurements T_1 and T_2, located after the first and second injection point, are used for feedback control.

Uncertainties

The process model (9.9)-(9.11) is subject to parametric uncertainty, see Table 9.1. The uncertainties associated with the model may lead to drastic changes in the predicted heat release and the shape of the predicted temperature profiles compared to the real process. The process model can be reformulated as

$$\dot{x} = f(x,u,p), \quad p \in \mathcal{P} \tag{9.19}$$

where x are the states, u are the control inputs, p are the uncertain parameters and $\mathcal{P}$ is the uncertainty set associated with the model parameters.

The main uncertainty is associated with the reaction kinetics. A reasonable model for the reaction kinetics is available, however, it is often

Table 9.1 Bounded uncertainties for the plate reactor

Parameter	Nominal value	uncertainty
Activation energy, E_a	76000 J/mol	± 2%
Pre-exponential factor, k_0	2e7 $m^3/(mol\ s)$	± 5%
Heat of reaction, ΔH	1.17e6 J/mol	± 5%
Heat transfer coefficient, h	3000 $W/(m^2\ K)$	± 10%

only validated for the desired operating point. Therefore, especially during start-up, there may be considerable uncertainty concerning the reaction kinetics.

9.3 Problem Formulation

The overall objective is to find control inputs that transfer the state of the process safely from an initial point, where the reactor is cold and no reactant B is fed, to an optimal operating point with maximum reactant conversion. The objectives can be formulated as

1. The main design objective is safety, which means that the temperature T_r throughout the reactor should at all times stay below a maximum limit, $T_{max} = 160°C$.

2. Maximize the reactant conversions at the reactor outlet, γ_A and γ_B, defined as
$$\gamma_A = \frac{c_C^{out}}{c_C^{out} + c_A^{out}}, \qquad \gamma_B = \frac{c_C^{out}}{c_C^{out} + c_B^{out}}, \tag{9.20}$$
where the superscript *out* denotes the concentrations at the reactor outlet. With the chemical reaction (9.1), this is equivalent to minimizing the amount of unreacted A and B in the reactor outflow.

3. The time to reach the optimal operating point should be as short as the primary objective permits.

There are many interesting challenges associated with start-up control of temperature sensitive exothermic reactions; nonlinear dynamics, actuator limitations and process uncertainty, see e.g. [Haugwitz and Hagander, 2006]. The nonlinear dynamics leads to multiple equilibrium profiles for a given set of control signals. One equilibrium profile corresponds to the

situation when no reaction occurs due to too low reactor temperature. Another equilibrium profile occurs when almost all reactants have converted at high reactor temperature, which is the desired operating point. Finally, in between these points, there is an equilibrium profile, which is unstable, due to the fast temperature rise caused by the exothermic reaction. See e.g. [Laabissi *et al.*, 2002] for an analysis of when there exist multiple equilibrium profiles.

The combination of limitations in the control inputs and unstable nonlinear dynamics require the start-up transition to be carefully optimized to avoid regions in the state-space where maximum control actions are not sufficient to prevent temperature transients above T_{max}.

Dynamic optimization can be used to generate feasible start-up trajectories in the face of control limitations and nonlinear dynamics. However, the process uncertainty adds another dimension to the complexity of the start-up problem. Therefore, robustness is the key focus in this problem formulation. Robustness is often associated with the ability for a feedback controller to compensate for disturbances and uncertainties. In this chapter, however, the feedback controller is considered to be fixed. Instead, we focus on the robustness of the off-line computed optimal trajectories, that is, the sensitivity of the optimal solution to model uncertainty. As will be described in the next section, optimal start-up trajectories may be arbitrarily sensitive to uncertainty, if the issue of robustness is not addressed in the formulation of the optimization problem.

Solving the start-up problem for a process with uncertain parameters is a challenging problem. Clearly, control trajectories computed based on a nominal model is not likely to reproduce the predicted output profiles for all models within the uncertainty set. For the plate reactor start-up, open loop application of the control trajectories may lead to degraded performance, and more importantly, violation of the temperature constraint. Introduction of a feedback control system, which is designed to track the predicted temperature profiles, significantly decreases the effects of modeling errors. However, for large parameter variations, the limitations of the actuator systems and the non-linear characteristics of the process may lead to violation of the safety requirements also in the presence of a well-designed feedback system. However, the ability of the feedback system to enforce the safety requirements is strongly dependent on the properties of the pre-computed control profiles. Start-up trajectories that have large sensitivity to parameter variations can be expected to be more difficult for the control system, which has limited authority, than trajectories for which the parametric uncertainty is small.

To summarize, parametric uncertainty have profound consequences for the start-up problem. In particular, it is not sufficient to meet the three objectives listed above for a nominal parameter set. To meet the objectives

for *all* parameters sets, a complimentary objective can be stated as:

Formulate the optimization problem so that the optimal control input $u(t)$ gives nominal state trajectories $x^{nom}(t)$ that have low sensitivity to parameter uncertainty.

9.4 The Optimization Problem

Definition of a dynamic optimization problem is an iterative procedure. The problem specification given in this section is the result of such a procedure, where successive refinement of objectives and constraints have resulted in the final formulation.

Specifications of the Optimization Problem

The state x of the reactor should be transferred from a cold stable equilibrium where no reaction takes place, to a stable equilibrium at high reactor temperature, where almost all of the reactants A and B are converted to C. By minimizing the concentration of the reactants c_A and c_B at the outlet of the reactor, ignition of the reactor, and transfer of the state, can be achieved.

As discussed in the previous section, it is very important to consider the robustness properties of the optimal solution. The robustness properties of a particular solution may be analyzed by calculating the state sensitivity with respect to parametric uncertainty, $\frac{\partial x}{\partial p}$. It is clear that a bang-bang solution, resulting e.g. from solving a minimum time problem, would not be robust, since the success of such a strategy is based on *timing*. In the presence of model uncertainty, the timing of the bang-bang sequence might not match the actual state of the system, with degraded performance as a result.

Two specifications that will improve the robustness of the optimal solution have been introduced; *i)* a penalty on the high frequency use of the control inputs, *ii)* a constraint on the accumulated amount of reactant B at each injection point.

High frequency penalties on control signals High frequency penalties on the inputs are introduced in the optimization, since it is impossible to implement arbitrarily fast control trajectories. In addition, high frequency penalties improve the numerical solution of the optimization as the problem becomes less singular.

In the optimization problem, there exist rate limits on the feed temperature and cooling temperature $\dot{T}_f^{sp}$ and $\dot{T}_c^{sp}$. These derivatives are penalized in the cost function as one kind of high frequency penalty. However,

penalties on the derivatives of the injection flow rates, u_{B1}^{sp} and u_{B2}^{sp}, were not sufficient. Instead a more general high-pass filter was introduced to increase the flexibility in the optimization formulation. By varying the cut-off frequency ω_c^f of the filter, it is possible to vary the frequency at which the HF-filter starts penalizing the control signal.

The high frequency penalties are important for the nominal solution, but they should also be considered in the context of the closed loop system. The optimal control profiles are implemented as feed forward signals in the closed loop feedback control system, see Figure 9.1. The feedback controller should be able to compensate for effects of the model mismatch. Clearly, the feedback system cannot be expected to suppress effects from model mismatch at frequencies higher than its bandwidth.

It is then convenient to design the high-pass filter cut-off frequency in terms of the frequency domain for the closed loop system. The frequency content of the off-line computed control variables should be such that high frequencies are not injected into the system. For the plate reactor, the bandwidth of the closed loop system is close to 0.5 rad/s at the final steady-state operating point, see also Figure 9.12. The limited bandwidth arises from actuator dynamics and limited control inputs. Accordingly, the bandwidth of the filter, ω_c^f was chosen to 0.5 rad/s. For comparison, the case of $\omega_c^f = 5$ rad/s was evaluated. The filter was implemented as a third order Butterworth high-pass filter.

Accumulation of reactant B For safety reasons, it is undesirable to have large amounts of reactant B accumulated in the reactor during start-up. This may lead to sudden ignitions and thermal runaways. This can be formally analyzed by investigating the state sensitivity with respect to the parametric uncertainties, see Section 9.4. The analysis shows that high concentrations and high temperatures give extremely high sensitivity for the given uncertainties. Therefore, it is required that the concentration of B should not exceed a specified maximum level. The constraints are chosen based on the steady-state values at optimal operation for the nominal model. In this application the concentration constraints were set to 200 mol/m^3 at the first injection point and 400 mol/m^3 at the second injection point.

The constraints on c_B have also another interesting physical interpretation. By limiting the amount of reactant B inside the reactor, the reaction rate r is limited, see equation (9.7). As a consequence, the rate of the change of the temperature, $\dot{T}_r$, is also limited, see equation (9.9). This is a more natural way of constraining the temperature derivative than introducing explicit constraints on the derivative in the optimization problem.

Constraints on reactor temperature The reactor temperature, T_r, should not exceed the specified maximum temperature anywhere along the reactor length, in order not to damage the reactor. The maximum temperature should be chosen somewhat conservative, in order to allow for temperature fluctuations due to disturbances and parameter uncertainty. The maximum temperature allowed in the reactor is $T_{max} = 160°\text{C}$, while the corresponding temperature bound in the optimization problem was set to 155°C.

Absolute and rate limitations of the control inputs There is a complicated inter-play between the feedforward trajectories and the closed loop system, which must be considered in the presence of model uncertainty. Enough control authority must be allocated to the feedback control system to enable it to compensate for any model mismatch. This is done by enforcing more conservative constraints in the optimization procedure than is required by the physical plant, see (9.13)-(9.18). In the optimization formulation, the more restrictive bounds were enforced: $0 \leq u_{B1}^{sp} \leq 0.7,\ 0 \leq u_{B2}^{sp} \leq 0.7,\ 30°\text{C} \leq T_f^{sp} \leq 80°\text{C},\ -1.5°\text{C/s} \leq \dot{T}_f^{sp} \leq 2°\text{C/s},\ 20°\text{C} \leq T_c^{sp} \leq 80°\text{C},\ -1.5°\text{C/s} \leq \dot{T}_c^{sp} \leq 0.7°\text{C/s}$.

The Optimal Control Problem

Given the specifications presented in the previous section, the optimization problem may now be formulated as

$$\begin{aligned}
&\min_u \int_0^{t_f} \alpha_A c_{A,N}^2 + \alpha_B c_{B,N}^2 + \alpha_{u_{B1}} (u_{B1,f}^{sp})^2 + \alpha_{u_{B2}} (u_{B2,f}^{sp})^2 \\
&\qquad + \alpha_{T_f} (\dot{T}_f^{sp})^2 + \alpha_{T_c} (\dot{T}_c^{sp})^2 \, dt \\
&\text{subject to} \\
&\qquad \dot{x} = f(x,u) \\
&\qquad T_{r,i} \leq 155, \quad i = 1..N \quad c_{B,1} \leq 200, \quad c_{B,2} \leq 400 \\
&\qquad 0 \leq u_{B1}^{sp} \leq 0.7, \quad 0 \leq u_{B2}^{sp} \leq 0.7 \\
&\qquad -1.5 \leq \dot{T}_f^{sp} \leq 2, \quad -1.5 \leq \dot{T}_c^{sp} \leq 0.7 \\
&\qquad 30 \leq T_f^{sp} \leq 80, \quad 20 \leq T_c^{sp} \leq 80
\end{aligned} \tag{9.21}$$

where $c_{A,N}$ and $c_{B,N}$ are the concentrations of A and B at the reactor outlet. $u_{B1,f}^{sp}$ and $u_{B2,}^{sp}$ are the high-pass filtered control variables corresponding to injection of reactant B. The weighting coefficients are denoted α_j. $T_{r,i}$ are the reactor temperatures in the N control volumes. The terms $c_{B,1}$ and $c_{B,2}$ are the concentrations at the first and the second injection point, respectively. Note that at time $t = 0$, the only term that is non-zero, is

the first term in the cost function $c_{A,N}$, since reactant A flows through the reactor. The remaining five terms in the cost function are zero, as they all are directly or indirectly associated with actions of the control inputs. In steady-state, at the end of the optimization time t_f, all terms except the two concentrations are zero. Therefore, the values of α_A and α_B determine the steady-state optimal operating point. To achieve high conversion of both reactants, these weights are chosen so that stoichiometric relations are achieved, thus maximizing the conversion. The corresponding Optimica description is shown in Listing 9.1, and the corresponding Modelica plate reactor model is given in Appendix E. For brevity, the specification of the transcription scheme has been excluded from the Optimica description.

The problem was transcribed and solved using a direct collocation method, as described in Chapter 3. For this purpose, the Optimica compiler presented in Chapter 7 was used. The resulting AMPL code was solved numerically by IPOPT, see Chapter 3. The input and state variables were discretized over a time horizon of 150 s using a grid of 450 points, which resulted in a large-scale optimization problem with approximately 145 000 variables. The execution time for solving the optimization problem was 1-2 hours on a Intel Core Duo 2.13 GHz system.

Scaling and Initial Guess

Scaling proved to be important in order for the numerical algorithm to converge. Therefore, all states and controls were scaled to the same order of magnitude. In addition, the automatic scaling facilities of IPOPT were utilized. Further, the convergence as well as the execution time of the optimization algorithm is dependent on the initial guess supplied to the NLP solver. Therefore, a square problem, with fixed inputs was solved initially, to generate initial guesses for all variables. Then the actual optimization problem could be solved with satisfactory convergence rate.

Optimization Results

In this section the effects of the given specifications on the optimal solution are presented and analyzed.

Overview of the characteristics The optimization results are plotted in Figures 9.3 and 9.4. The main characteristic is the need for heating to achieve ignition. By increasing the feed temperature T_f the reactor temperature increases and after ignition the reaction becomes self-accelerating and T_f can return to its initial value. Similarly, the cooling temperature T_c is increased to promote ignition at the second injection point. The maximum conversion occurs when the reactor temperatures around the two injection points are at the maximum limit of 155°C.

```
optimization PlateReactorOptimization (objective=cost(finalTime),
                                       startTime=0,
                                       finalTime=150)

  PlateReactor pr(u_T_cool_setpoint(free=true),
                  u_TfeedA_setpoint(free=true),
                  u_B1_setpoint(free=true),
                  u_B2_setpoint(free=true));

  parameter Real sc_u = 670/50 "Scaling factor";
  parameter Real sc_c = 2392/50 "Scaling factor";

  Real cost(start=0);
equation
  der(cost) = 0.1*pr.cA[30]^2*sc_c^2 +
              0.025*pr.cB[30]^2*sc_c^2 +
              1*pr.u_B1_setpoint_f^2 +
              1*pr.u_B2_setpoint_f^2 +
              1*der(pr.u_T_cool_setpoint)^2*sc_u^2 +
              1*der(pr.u_TfeedA_setpoint)^2*sc_u^2;

constraint
  pr.Tr/u_sc<=(155+273)*ones(30);

  pr.cB[1]<=200/sc_c;
  pr.cB[16]<=400/sc_c;

  pr.u_B1_setpoint>=0;
  pr.u_B1_setpoint<=0.7;
  pr.u_B2_setpoint>=0;
  pr.u_B2_setpoint<=0.7;

  pr.u_T_cool_setpoint>=(15+273)/sc_u;
  pr.u_T_cool_setpoint<=(80+273)/sc_u;
  pr.u_TfeedA_setpoint>=(30+273)/sc_u;
  pr.u_TfeedA_setpoint<=(80+273)/sc_u;

  der(pr.u_T_cool_setpoint)>=-1.5/sc_u;
  der(pr.u_T_cool_setpoint)<=0.7/sc_u;
  der(pr.u_TfeedA_setpoint)>=-1.5/sc_u;
  der(pr.u_TfeedA_setpoint)<=2/sc_u;
end PlateReactorOptimization;
```

Listing 9.1 An Optimica description corresponding to the optimal control problem (9.21).

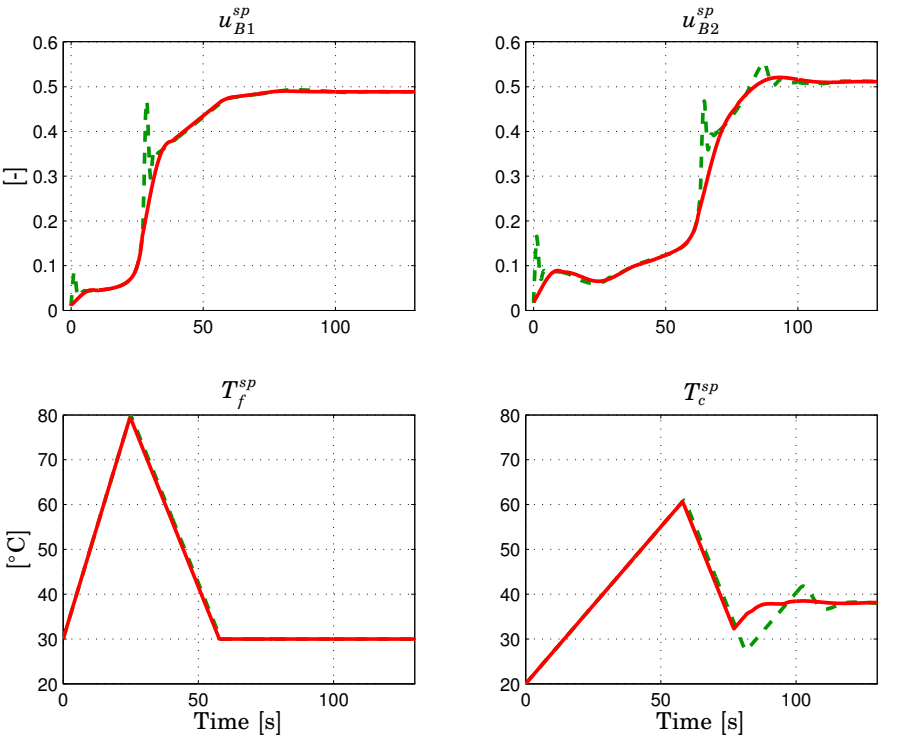


Figure 9.3 Optimal control profiles. The dashed curves correspond to the case $\omega_c^f = 5.0$ rad/s and the solid curves corresponds to $\omega_c^f = 0.5$ rad/s.

In the optimization formulation, the reactant B concentration around the two injection points is limited, see Figure 9.4. The constraints on c_B ensure that there is only a limited accumulation of unreacted chemicals in the reactor. The injection flow rates, u_{B1} and u_{B2}, are initially low to comply with the c_B-constraints. Before more B can be injected, the temperature at the injection points needs to be increased by T_f and T_c. The higher reactor temperature increases the reaction rate, i.e. more of the injected B is consumed. First then is it possible to increase the injection of B and still comply with the constraint in c_B. The constraints in c_B reduce the risk of uncontrolled ignition and increases the robustness of the optimal trajectories.

Results when varying the high frequency penalty on u_{B1} ***and*** u_{B2}
Two cases have been considered, $\omega_c^f = 0.5$ and $\omega_c^f = 5.0$ rad/s. The optimal control profiles for both cases are shown in Figure 9.3. When $\omega_c^f = 5.0$,

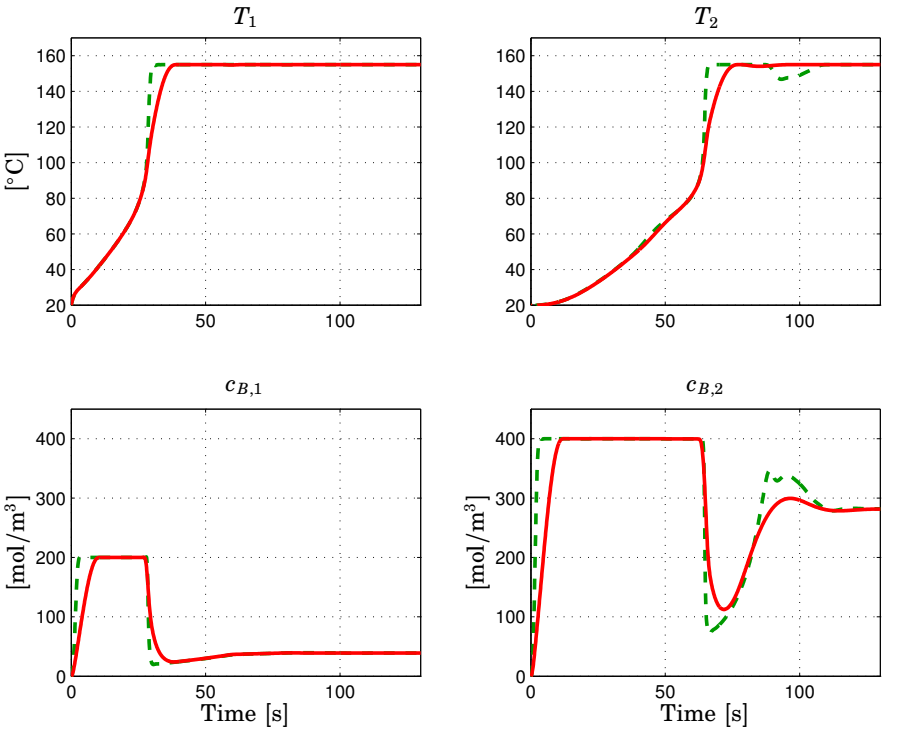


Figure 9.4 Optimal profiles profiles for reactor temperature and concentration of substance B. The left plots correspond to the first injection point, whereas the right plots correspond to the second injection point. The dashed curves correspond to the case $\omega_c^f = 5.0$ whereas solid curves corresponds to $\omega_c^f = 0.5$.

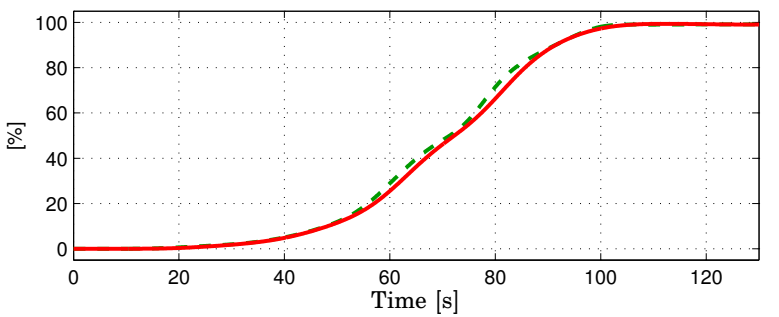


Figure 9.5 Conversion of reactant A at the outlet. The dashed curve correspond to the case $\omega_c^f = 5.0$ and the solid curve correspond to $\omega_c^f = 0.5$.

there is clearly more high frequency content of the injection control inputs. This allows the start-up to be somewhat faster as the control actions can be more aggressive, when the temperature reaches the maximum value, see Figure 9.4.

When $\omega_c^f = 0.5$, the optimization gives a slower transient to the maximum temperature value, since the injection control is penalized for lower frequencies than in the previous case. Notice that the temperature constraints are active for both cases at the optimal steady-state operation point.

The high frequency content of the case when $\omega_c^f = 5.0$ leads to control input trajectories that require exact timing to satisfy the temperature constraints. This optimal solution will be much more sensitive to parametric uncertainty, as will be demonstrated in Section 9.7.

However, there is almost no difference in the settling time of the conversion, γ_A, see Figure 9.5. With time to reach maximum conversion being the primary performance measure, there seems to be almost no performance loss for increasing robustness of the optimal solution in this case.

Results for varying the bounds on c_B In Figures 9.6, 9.7 and 9.8 the optimal start-up trajectories are shown for three cases of different concentration bounds on reactant B. The high frequency penalty on the injection control inputs was fixed to $\omega_c^f = 0.5$ rad/s for all three cases.

With tighter constraints on c_B, the reactor temperature needs to be higher before more reactant can be injected. This is clearly shown in the lower plots of Figure 9.6, where T_f^{sp} and T_c^{sp} are increased to raise the reactor temperature before more injection can occur. Figure 9.8 shows the slower transient to the final operating point in terms of conversion, when tighter constraints are used. However, when higher concentrations are permitted, the start-up trajectories enter regions in the state-space where the state sensitivity for parametric uncertainty is very high, see Section 9.4.

Sensitivity Analysis

In the presence of uncertainty, there will be deviations in the actual state trajectories from the off-line computed optimal trajectories. Sensitivity analysis is performed to quantify the impact of the optimization specifications on the parametric sensitivity of the optimal trajectories. In addition, we will study and compare the sensitivity of the open loop optimal trajectories for the uncertain parameters from Table 9.1.

The state sensitivity to parameter changes, $\frac{\partial x}{\partial p}$, gives an indication of how parametric uncertainty affects the behavior of the process. The higher sensitivity, the higher is the impact of the model mismatch. For

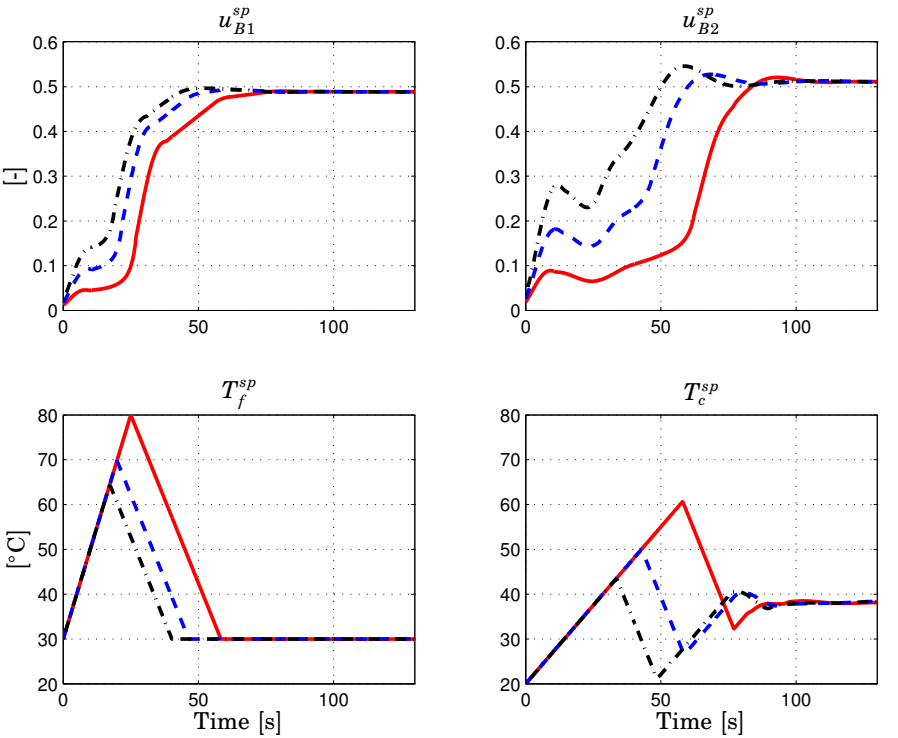


Figure 9.6 Control signals for optimal start-up for various c_B-constraints. $c_{B,1} \leq 200$, $c_{B,2} \leq 400$ (solid), $c_{B,1} \leq 400$, $c_{B,2} \leq 800$ (dashed), $c_{B,1} \leq 600$, $c_{B,2} \leq 1200$ (dash-dot)

small parameter changes Δ, the state trajectories are given by

$$x = x^{nom} + \frac{\partial x}{\partial p}\Delta p^{nom}, \quad \text{where} \quad \Delta = \frac{p - p^{nom}}{p^{nom}} \tag{9.22}$$

is the dimensionless deviation factor of the parameter. In the start-up of the plate reactor, temperature is the most important safety concern. Therefore, the following analysis is focused on the sensitivity of the temperature at the second injection point T_2 to parameter variations.

Table 9.2 summarizes the maximum temperature sensitivity during start-up time to parameter changes for various c_B-constraints and cut-off frequencies ω_c^f. The value in each entry of the table is the maximum temperature deviation due to a 0.1% increase in the specific parameter,

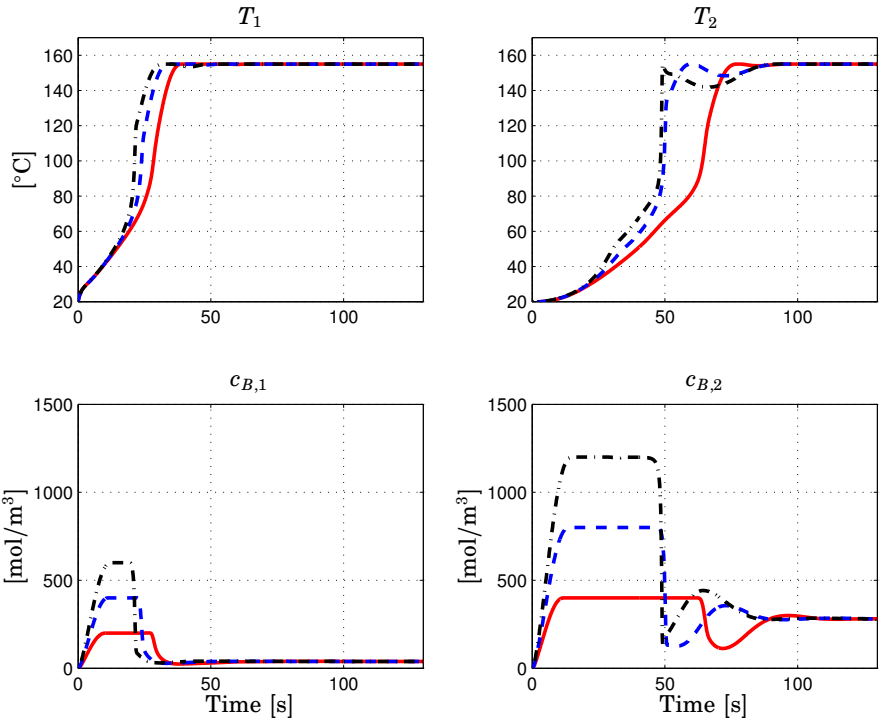


Figure 9.7 Temperatures and concentrations during optimal start-up for various c_B-constraints defined in Figure 9.6.

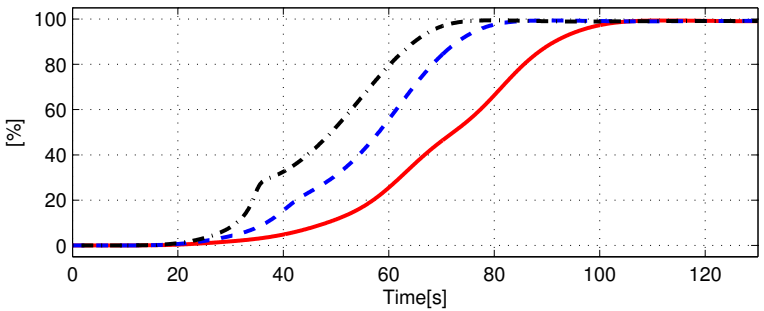


Figure 9.8 Conversion of reactant A at reactor outlet during optimal start-up for various c_B-constraints defined in Figure 9.6.

for example

$$\max_{t\in[0,t_f]} T_2(t) - T_2^{nom}(t) = \max_{t\in[0,t_f]} \frac{\partial T_2(t)}{\partial E_a} 0.001 \cdot E_a^{nom} \tag{9.23}$$

This linear approximation is only valid for very small parameter changes, due to the severe nonlinearities of the system. For the wider range of uncertainties described earlier in Table 9.1, the nonlinear effects have to be considered in order to analyze the state sensitivity, see the next section.

For tighter c_B-constraints, the effect of the model mismatch on the state trajectories is significantly smaller, up to an order of magnitude. The maximum sensitivity in each case occurs when the reaction ignites and the temperature increases very quickly. Start-up trajectories that have large sensitivity to parameter variations can be expected to be more difficult for the control system, which has limited authority, than trajectories for which the parametric uncertainty is small. However, the decreased sensitivity comes at the price of somewhat longer start-up time, see Figure 9.8.

The sensitivity of the optimal trajectories when $\omega_c^f = 5$ rad/s is roughly 50% higher than for $\omega_c^f = 0.5$ rad/s, but it is in turn much smaller than the sensitivity for higher constraints on c_B. The choice of cut-off frequency for the high-pass filter has a smaller, but still significant, impact compared to the c_B-constraints on the sensitivity of the optimal solutions. From Table 9.2 it can also be noted that the reactor temperature is most sensitive to changes in the activation energy E_a. It can be explained by the fact that the reaction rate r depends exponentially on E_a, but only linearly on for example k_0.

The sensitivities discussed in this section were computed using the numerical solver DASPK, see [Maly and Petzold, 1996]. This algorithm implements a BDF method for solving index-1 DAE systems, and can also integrate the sensitivity equations.

Effects of Model Mismatch

In the previous section, linear sensitivity analysis showed that tighter concentration constraints in the optimization formulation gave optimal trajectories that have significantly reduced sensitivity to parameter uncertainty. That analysis is limited to small parameter variations, so in this section, we will use open loop simulations with the nonlinear model to include the nonlinear and multiparametric effects in the sensitivity analysis.

Up to now, optimal start-up trajectories have been presented for the nominal model. Here we will take a first look at the effect of model mismatch. The uncertainty of the model parameters were described in Table 9.1. One specific case of model mismatch is now studied to provide some

Table 9.2 Maximum deviation in T_2 due to parametric sensitivity for different start-up specifications.

	$\omega_c^f = 0.5$ $c_{B,2} \leq 400$	$\omega_c^f = 0.5$ $c_{B,2} \leq 800$	$\omega_c^f = 0.5$ $c_{B,2} \leq 1200$	$\omega_c^f = 5$ $c_{B,2} \leq 400$
$\frac{\partial T_2}{\partial E_a} \cdot 0.001 E_a^{nom}$	-1.5 °C	-14.9 °C	-51.8 °C	-2.2 °C
$\frac{\partial T_2}{\partial k_0} \cdot 0.001 k_0^{nom}$	0.062 °C	0.71 °C	8.6 °C	0.091 °C
$\frac{\partial T_2}{\partial \Delta H} \cdot 0.001 \Delta H^{nom}$	0.24 °C	2.0 °C	15.8 °C	0.35 °C
$\frac{\partial T_2}{\partial h} \cdot 0.001 h^{nom}$	-0.14 °C	-1.4 °C	-15.4 °C	-0.18 °C

insights. The parameter errors are the following; the heat transfer coefficient h 10% lower, the heat of reaction ΔH 5% higher, the pre-exponential coefficient k_0 5% lower and the activation energy E_a 2% higher than in the nominal model. This model mismatch is selected, since it is one of the most difficult cases of model mismatch for the feedback controller to handle, according to the Monte Carlo simulations that will be presented in Section 9.7.

Figure 9.9 plots the difference $\Delta T = T_1 - T_1^{nom}$ between the actual temperature and the nominal temperature at the first injection. The higher value in E_a and lower value in k_0 will reduce the reaction rate and the subsequent heat release. Thus, the actual temperature will be lower than the nominal temperature before ignition occurs. After the ignition, the higher value in ΔH leads to more heat being released, thus the actual temperature will be higher than the nominal. A reduced heat transfer coefficient h will also lead to the actual temperature being higher than the nominal. The combination of these four parameter errors form a challenging model mismatch for a feedback controller to handle. First the mismatch leads to lower temperatures, but after ignition has occurred, the effect of the mismatch is the directly opposite.

Figure 9.9 shows that the effect of the model mismatch is significantly smaller when tighter concentration constraints in B are enforced or when lower cut-off frequency ω_c^f is used. This supports the results of Table 9.2 even when the nonlinear effects and uncertainty in multiple parameters are considered simultaneously. In the next sections, we will extend the robustness analysis to the closed loop system.

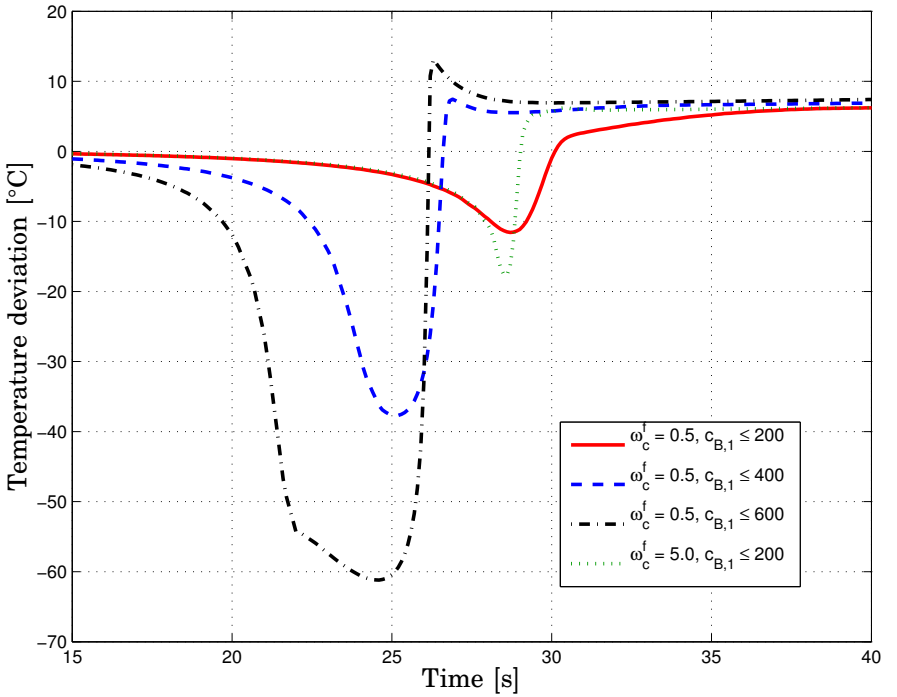


Figure 9.9 Difference in temperature at first injection point T_1 for the nominal trajectory and the actual trajectory caused by model mismatch for various c_B-constraints.

9.5 Feedback Control

The dynamic optimization algorithm calculates open loop trajectories for the four manipulated variables. Feedback control is necessary, however, due to process uncertainties and disturbances. Only temperature feedback is available. In the feedback control structure, the optimization results are used as reference and feed forward trajectories,

$$T_1^{ref} = T_1^{opt}, \quad T_2^{ref} = T_2^{opt}, \tag{9.24}$$

$$T_f^{ff} = T_f^{sp,opt}, \quad T_c^{ff} = T_c^{sp,opt}, \quad u_{B1}^{ff} = u_{B1}^{sp,opt}, \quad u_{B2}^{ff} = u_{B2}^{sp,opt}, \tag{9.25}$$

where the superscript ^{opt} denotes the results from the optimization in (9.21).

Figure 9.10 shows the Bode diagrams of the four most dominant open loop transfer functions from the control inputs to the reactor temperatures T_1 and T_2. The transfer functions are linearizations of the process model at

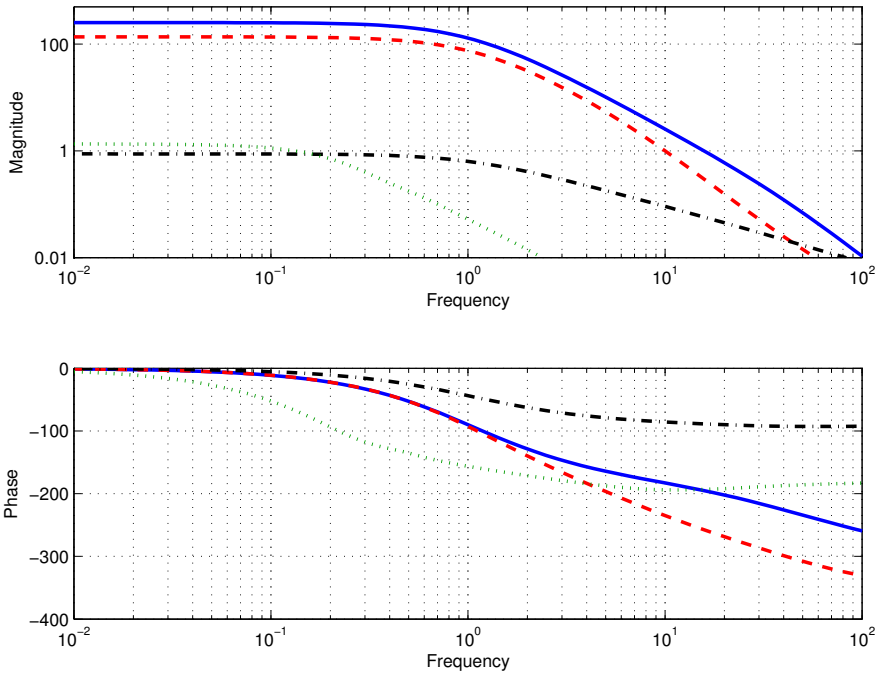


Figure 9.10 Bode diagrams; $u_{B1}^{sp} \rightarrow T_1$ (solid), $u_{B2}^{sp} \rightarrow T_2$ (dashed), $T_f^{sp} \rightarrow T_1$ (dash-dot) and $T_c^{sp} \rightarrow T_2$ (dotted).

steady-state after the start-up. In the Bode diagram, we can see that the injection flow rates of reactant B, u_{B1}^{sp} and u_{B2}^{sp}, have larger process gain and faster impact on T_1 and T_2 than T_f^{sp} and T_c^{sp} have. However, there are several nonlinear effects that should be considered when choosing control signals for feedback. For example, the injection flow rates may affect the stoichiometric balance and should thus be used with care in steady-state. Clearly, the variables T_f^{sp} and T_c^{sp} also affect the reactor temperatures, but their input dynamics and rate limits will prevent achieving a desirable bandwidth for the closed loop system using these two inputs only.

Therefore, a mid-ranging control structure, see e.g. [Åström and Hägglund, 2005], shown in Figure 9.11 is introduced. The idea of mid-ranging is to use control variables with fast impact, in this case, u_{B1}^{sp} and u_{B2}^{sp}, to account for high frequency variations. This is realized by the controllers C_1 and C_2 in Figure 9.11. Meanwhile, variables with slower impact, in this case, T_f^{sp} and T_c^{sp}, are used to compensate for low frequency variations or effects of model mismatch, using controllers C_3 and C_4. The actions of T_f^{sp} and T_c^{sp} on the process, enable the two injection flow rates u_{B1}^{sp} and u_{B2}^{sp} to

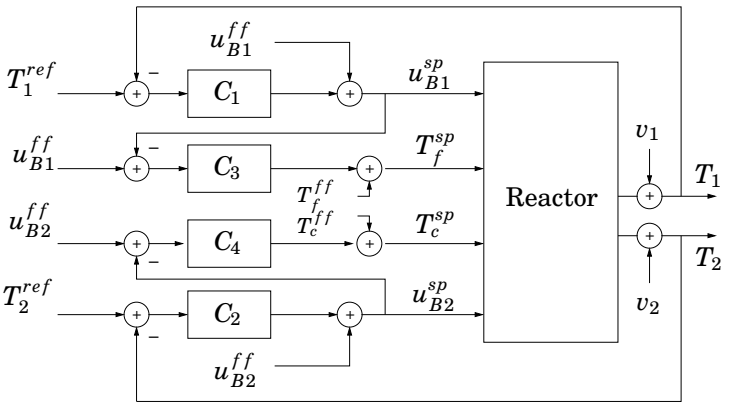


Figure 9.11 Block diagram for the mid-ranging feedback control system.

return to their optimal values, thus achieving the correct stoichiometric conditions between A and B in steady state. To reduce the interaction between the fast and the slow control variables, the slow control loops are designed to have a closed loop bandwidth that is an order of magnitude smaller.

Each controller C_1, C_2, C_3 and C_4 in Figure 9.11 is implemented as a PID-controller. The tuning of the controller parameters is based on the AMIGO method, which involves robust loop-shaping and optimization of the integral gain, see [Åström and Hägglund, 2005].

Finally, to analyze the resulting closed loop system, the singular values of the sensitivity function S is plotted in Figure 9.12. S is the transfer function from the disturbance signals v_1 and v_2 to the temperature signals T_1 and T_2 and it is defined as

$$S = [I + PC]^{-1}, \tag{9.26}$$

where P is the linearization of the nominal reactor model at steady-state and C represents the mid-ranging control structure depicted in Figure 9.11. The closed loop system has good attenuation of constant and low frequency disturbances and model-mismatch effects. The maximum singular value for any frequency is 1.17 at 1.7 rad/s, which implies that the feedback control gives a good robustness. However, $\sigma_{max} \geq 1$ for $\omega \geq 0.9$ rad/s. This indicates that the feedback controller will have difficulties in attenuating model-mismatch effects with higher frequencies.

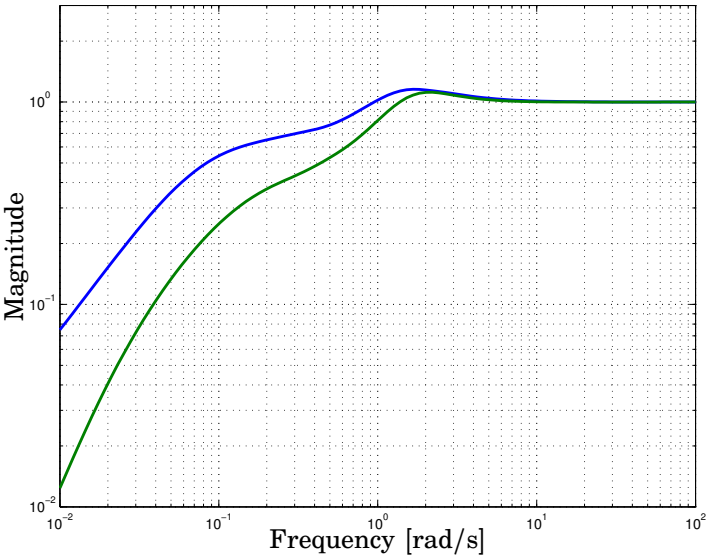


Figure 9.12 Plot of maximum and minimum singular values of the sensitivity function S

9.6 Simulation with Feedback Control

In this section, the closed loop system with feedback control is simulated and analyzed, see Figure 9.1. The result of the feedback is compared to the optimal solution given that the true values of the uncertain parameters had been known.

In Figures 9.13 and 9.14, the start-up trajectories for three cases are plotted; the optimal solution for the nominal plant, the feedback solution when there is model mismatch (as defined in Section 9.4) and, finally, the optimal solution knowing the exact model mismatch. The specifications in the optimization problem are $\omega_c^f = 0.5$ rad/s and $c_{B,1} \leq 200$ and $c_{B,2} \leq 400$ mol/m^3. The model mismatch and its effect were described in Section 9.4.

The feedback controller takes the temperature trajectories T_1^{ref} and T_2^{ref} from the optimal solution as references. The controller manipulates primarily u_{B1} and u_{B2} to achieve this reference tracking, despite tracking errors resulting from model mismatch. As described in Section 9.4, this model mismatch leads initially to lower temperatures than in the reference. Thus, the controller have to increase the injection flow rates of B to compensate. After the ignition, the model mismatch quickly gives higher temperatures than for the nominal model, see e.g. Figure 9.9. The

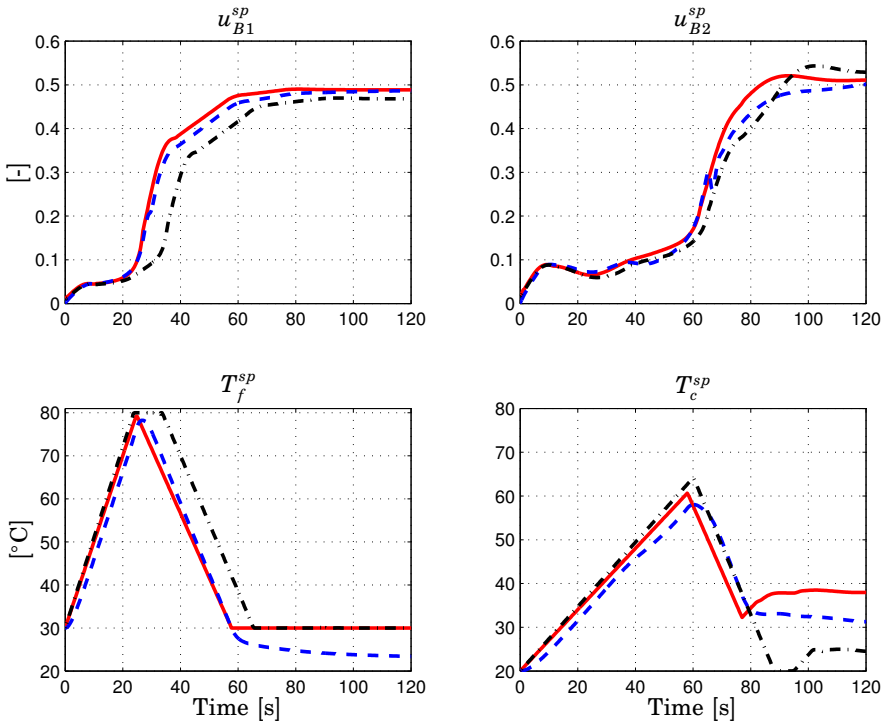


Figure 9.13 The control signals. Comparison of optimal solution with nominal model (solid), feedback control with model mismatch (dashed) and the optimal solution when the exact model mismatch is given (dash-dot).

controller quickly lower the flow rates again.

Meanwhile, to achieve stoichiometric conditions in steady-state, the mid-ranging control reduces the pre-heating T_f and cooling T_c, to allow the injection flow rates u_{B1} and u_{B2} to return to their pre-defined optimal trajectories. In other words, during the transients the higher heat release is compensated for by u_{B1} and u_{B2}, but in stationarity by T_f and T_c. In this way, mid-ranging allows each control input to be used at its best depending on its limitations, dynamics and available bandwidth. The control limitations for T_f and T_c were more restrictive in the optimization formulation, to allocate some additional flexibility to the feedback controller. The feedback controller is bound by the original limitations defined in (9.13)-(9.18).

The feedback control tracks the optimal temperature trajectories and preserves the optimal injection flow rates with the mid-ranging control. However, these optimal trajectories are computed for the nominal model

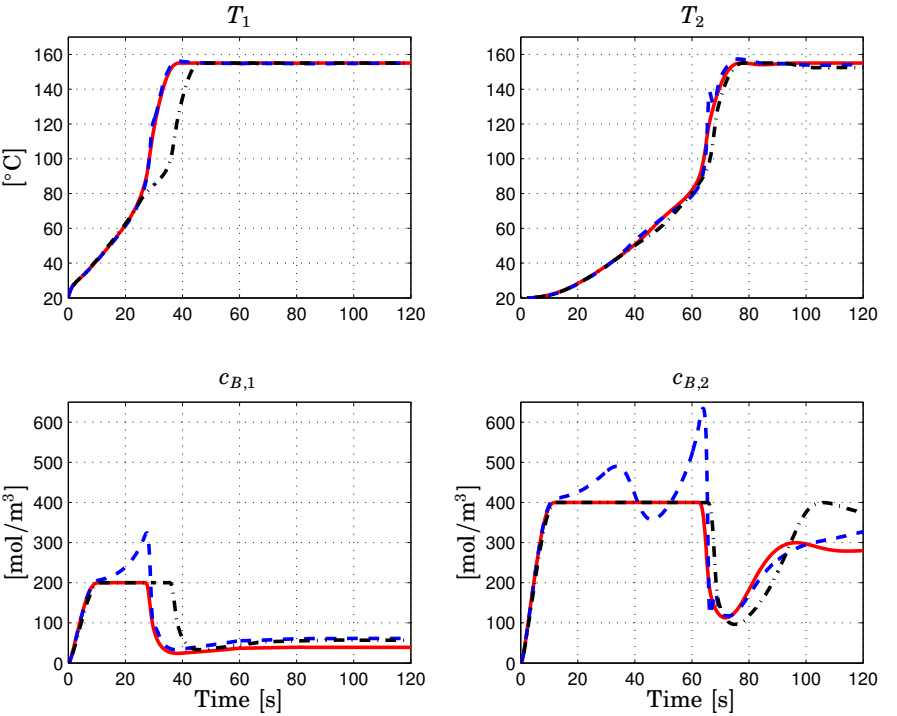


Figure 9.14 The state variables. Comparison of optimal solution with nominal model (solid), feedback control with model mismatch (dashed) and the optimal solution when the exact model mismatch is given (dash-dot).

and may not be optimal due to model mismatch. It may then be interesting to see how the feedback solution compares with an optimal start-up if the exact model mismatch had been known, see the dash-dot lines in Figures 9.13 and 9.14. As the activation energy is higher, the reactor flow needs to be heated more, before injection can be increased further, thus avoiding excessive accumulation of B. The higher heat release leads to less reactant being injected at the first point, since T_f^{sp} is already at its lower limit of 30°C. Thus, the injection of B is slightly redistributed from the first to the second injection point. To adjust for the increased heat release, the cooling temperature T_c^{sp} is lowered in stationarity.

The feedback controller succeeds in tracking the temperature references, but the concentration of reactant B increases temporarily due to the model mismatch, see the lower plots in Figure 9.14. This may be avoided if concentration feedback is available. The resulting operating point will be different for the feedback solution and the optimal solution knowing the

model mismatch. However, the conversion of reactant A is 98.8% for both operating points, thus the feedback control does not loose any efficiency for this particular model mismatch.

9.7 Monte Carlo Simulations

In this section. the robustness of the closed loop system is evaluated by means of Monte Carlo simulations. To reduce the state sensitivity of the optimal trajectories to parametric uncertainties, we have introduced two key specifications in the optimization problem; i) high frequency penalties on u_{B1}^{sp} and u_{B2}^{sp} defined by a cut-off frequency ω_c^f and ii) constraints on c_B at the two injection points. The temperatures at the first and second injection points were evaluated and compared for five cases:

Case 1: Feedback control disabled, optimal trajectories computed for $\omega_c^f = 0.5$ rad/s, $c_{B,1} \leq 200$ mol/m^3 and $c_{B,2} \leq 400$ mol/m^3

Case 2: Closed loop control, optimal trajectories computed for $\omega_c^f = 0.5$ rad/s, $c_{B,1} \leq 600$ mol/m^3 and $c_{B,2} \leq 1200$ mol/m^3

Case 3: Closed loop control, optimal trajectories computed for $\omega_c^f = 0.5$ rad/s, $c_{B,1} \leq 400$ mol/m^3 and $c_{B,2} \leq 800$ mol/m^3

Case 4: Closed loop control, optimal trajectories computed for $\omega_c^f = 0.5$ rad/s, $c_{B,1} \leq 200$ mol/m^3 and $c_{B,2} \leq 400$ mol/m^3

Case 5: Closed loop control, optimal trajectories computed for $\omega_c^f = 5.0$ rad/s, $c_{B,1} \leq 200$ mol/m^3 and $c_{B,2} \leq 400$ mol/m^3

The first case is open loop control, the remaining four are closed loop control. Cases 2, 3 and 4 displays the effect of tighter c_B-constraints. Case 5 considers less high frequency penalties on the injection control inputs.

For each case, 5000 simulations were carried out. In each simulation, the values of the model parameters $E_a, \Delta H, k_0$ and h were randomly generated from a uniform distribution, based on the uncertainties in Table 9.1. The closed loop system is simulated and the reactor temperatures at the injection points are recorded. Then a new sample of the uncertain parameters is generated and the whole procedure is repeated.

To visualize the sensitivity of the five cases to the uncertainties, envelope curves were constructed, see Figure 9.15. They show the minimum and maximum temperature among the 5000 simulations for each sample time t. The horizontal dashed line indicates the safety limit of 160 °C. Any temperature above this may lead to safety shut-down to avoid damage to the reactor. A larger area between the minimum and maximum

temperature indicates a higher variation in reactor temperature due to insufficient robustness to the model mismatch.

In the first case, without feedback control, the reactor temperature at the first injection point spans an interval from 145 to 164°C at steady state. At the second injection point, there are some situations where the reaction does not even ignite directly after injection, but instead ignition occurs further downstream in the reactor. This gives an even larger temperature interval. The remaining cases include feedback control.

The second case has the least restrictive c_B-constraints. The optimal solution based on the nominal model is then extremely sensitive to uncertainties, see Section 9.4. Therefore, the Monte Carlo simulations for this case show the widest range between the minimum and maximum reactor temperature. In fact, the time for ignition of the reaction at T_2 varies from 45 to 87 seconds, due to the effects of different model mismatch.

In the third case, the c_B-constraints are somewhat stricter than in the second case. The parameter uncertainty leads to model mismatch that the feedback controller can not handle fast enough. In fact, the feedback controller actually worsen the situation, since the transient results are even worse than without feedback. In steady-state, however, the temperatures are back to the nominal values, due to the integral action.

The most robust start-up is achieved in the fourth case, where tighter c_B-constraints and a low ω_c^f yield optimal solutions with very low sensitivity to uncertainty. Therefore, the effects of the model mismatch is small and the feedback controller succeeds in keeping the temperature below the safety limit.

In the fifth and final case, the same c_B-constraints are enforced, but there is less penalty on the high frequency components of the injection control inputs u_{B1} and u_{B2}. Due to the very fast transient, the maximum temperature limit for the nominal model, the optimal solution is sensitive to model mismatch. There are some parameter values within the uncertainty region, for which the feedback controller can not keep the reactor temperatures below the safety limit.

To summarize, the first case shows that feedback is necessary. However, due to limited bandwidth in the feedback controller, the optimal start-up trajectories cannot be computed based on an arbitrary optimization specification. To avoid unsafe start-up, the optimal start-up trajectories need to have low sensitivity to parameter uncertainty. This is achieved by introducing high-frequency penalties on the control signals and enforcing concentration constraints on reactant B.

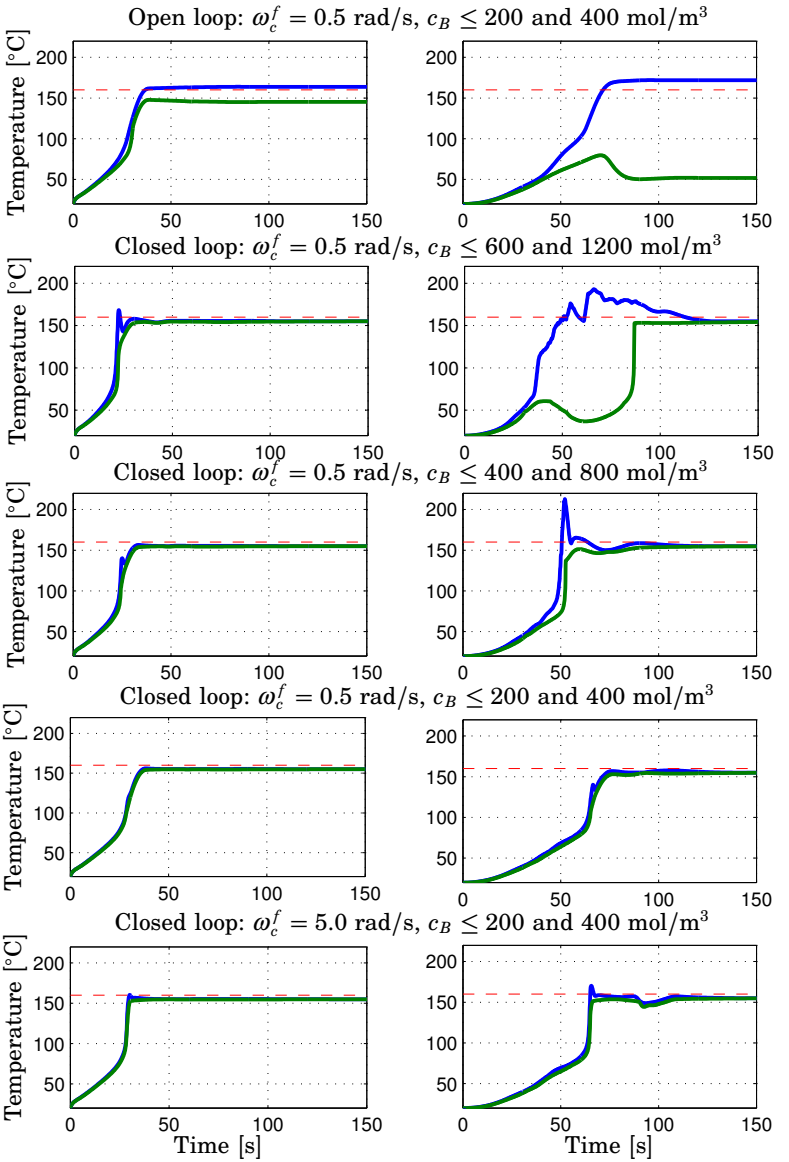


Figure 9.15 Maximum and minimum reactor temperature at the first (left column) injection point and the second injection point (right column) at each time instant out of 5000 sample simulations

9.8 Summary and Conclusions

In this chapter, it has been shown how dynamic optimization can be used to generate trajectories for start-up of a plate reactor. The complex interplay between the formulation of the optimization problem and the implementation of its solution in a closed loop setting has been discussed. With model mismatch, an optimal solution may lead to transients that the feedback controller can not handle due to limited bandwidth. Therefore the specifications of the optimization problem include concentration constraints on the injected reactant B and a high frequency penalties on the control inputs. This results in an optimal solution with a significantly reduced sensitivity to uncertainties compared to solutions closer to the time-optimal. Temperature feedback control ensures that the optimal temperature trajectories are tracked. A mid-ranging control structure is used to take advantage of all four available control inputs. The proposed optimization specifications has been evaluated in Monte Carlo simulations, under the assumption of uncertain parameter values, with satisfactory result.

The design procedure has been supported by automatic code generation tools, where the model description has been expressed in Modelica. The availability of automatic tools has enabled focus to be shifted from the details of *encoding* the problem towards *formulation* of the actual optimization problem. As a result, the iterative process of formulating a dynamic optimization problem is supported. A natural extension of this work is to consider multi-parametric optimization, where the uncertainty in the parameters are included into the optimization problem.

Control of Pendula

IV

IV

10

Safe Manual Control of an Inverted Pendulum

10.1 Introduction

In many control applications, a system is controlled by a combination of manual and automatic control. Typical examples are air crafts, where stability augmentation systems are used to assist the pilot. The combination of manual and automatic control is particularly crucial for unstable systems with actuator constraints, because the system can be driven to such a state unintentionally by manual control. The problem is similar to the one encountered when controlling unstable air crafts such as the Saab Gripen, where in some flight conditions the unstable mode is so fast that a pilot cannot stabilize the system. In this case, the aircraft dynamics is non-linear and, also, the actuator rate is bounded, see [Rundqwist *et al.*, 1997; Patcher and Miller, 1998]. The pendulum problem can, however, serve as a simple prototype for an interesting class of real problems.

The essence of the problem can be captured in the following formulation. Consider an unstable system with actuator saturation. Find a control strategy that stabilizes the system and provides facilities for manual control. The strategy should be such that the system can be controlled manually without driving it unstable.

There is an extensive literature on stabilizing a dynamical system subject to input or state constraints. For linear systems, the problem is well understood. For stable systems there are strong results stating that there always exist a controller that stabilizes the system globally. The result was proven for a chain of integrators in [Teel, 1992] and for the general case in [Sussmann *et al.*, 1994]. For unstable systems, the situation is more involved. A key concept for control of unstable systems is the notion

of *Reachability Sets*, which contain all points of the state space such that there exists a *feasible* control trajectory that brings the system to the origin. The problem is closely associated with that of minimum time optimal control. It can be shown that the reachability set of a linear exponentially unstable system is bounded in the directions of the unstable modes. Consequently, only semi-global stability may be achieved. An elegant result for calculation of reachability sets for exponentially unstable systems as well as a method of semi-global stabilization are given in [Hu *et al.*, 2001].

For non-linear systems, the situation is different. Fewer results are available on stabilization with bounded controls, [Teel, 1996] being a notable exception. The problem of calculating reachability sets is significantly harder for non-linear systems.

Another branch of the theory deals with the problem of *anti-windup*. In this setting, a local *performance controller* is designed without taking the saturation nonlinearity into account. The problem is then to find an anti-windup modification of the controller that leaves the behavior of the local controller unaffected when there is no saturation, and limits the effects of saturation if it occurs, see for example [Rönnbäck, 1993]. In [Teel and Kapoor, 1997], the problem was given a rigorous definition and solved for the case of stable linear systems. In [Teel, 1999] the anti-windup problem for exponentially unstable linear systems is addressed.

In this chapter, the inverted pendulum, representing a non-linear unstable system is studied. The aim of the controller is to enable velocity tracking of the pivot point of the pendulum while ensuring stability. The reachability set of the system is explicitly characterized, and a controller based on this set is proposed. The chapter is an extension of [Åkesson and Åström, 2001], where a linearized pendulum system was studied.

10.2 Equations of Motion

Consider the inverted pendulum on a cart in Figure 10.1. Let the position of the cart be x, and the angle of the pendulum θ. Let l denote the distance from the pivot point to the center of mass of the pendulum, m_p the mass of the pendulum and J_p its moment of inertia w.r.t. the pivot point. Further, let m_c denote the mass of the cart, F the force acting on the cart and g the acceleration due to gravity. The equations of motion of the inverted pendulum may be written as

$$\begin{aligned} J_p\ddot{\theta} - m_p l\ddot{x}\cos\theta - m_p g l\sin\theta &= 0 \\ -m_p l\ddot{\theta}\cos\theta + (m_c + m_p)\ddot{x} - m_p l\dot{\theta}^2\sin\theta &= F. \end{aligned} \tag{10.1}$$

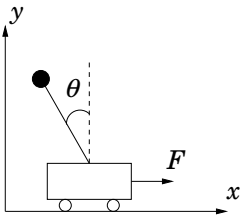


Figure 10.1 A schematic picture of the inverted pendulum on cart.

By introducing the input transformation

$$\begin{aligned} F = &\frac{1}{J_p}\left[v(m_c J_p + m_p J_p' + m_p^2 l^2 \sin^2\theta)\right. \\ &\left. -m_p^2 g l^2 \sin\theta\cos\theta + J_p m_p l \dot{\theta}^2 \sin\theta\right] \end{aligned} \tag{10.2}$$

where J_p' is the moment of inertia of the pendulum with respect to its center of mass, the control input to the system is transformed to the acceleration of the cart, v, rather than the acting force F. Notice that the transformation can be done globally in the state space since $J_p \leq m_p l^2$. Introducing the normalizations

$$\begin{aligned} x_1 &= \theta \quad x_2 = \sqrt{\frac{J_p}{m_p g l}}\,\dot{\theta} \quad x_3 = \sqrt{\frac{m_p l}{J_p g}}\,\dot{x} \\ u &= \frac{v}{g} \quad \tau = \sqrt{\frac{m_p g l}{J_p}}\,t \end{aligned} \tag{10.3}$$

the dynamics of the system may be written as

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= \sin x_1 + u\cos x_1 \\ \dot{x}_3 &= u. \end{aligned} \tag{10.4}$$

Notice that the state x has been excluded, because the aim of the control system is to enable velocity control of the cart.

The equilibria of the pendulum are $x_1 = 0$ and $x_1 = \pi$ which represents a saddle (unstable) and a center (stable) respectively. Linearization of the model (10.4) with respect to the unstable equilibrium point $x = (0,0,0)$ is given by

$$\dot{x} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} u. \tag{10.5}$$

10.3 Reachability Set Analysis

The reachability set plays an important role for design of controllers for unstable systems subject to input saturation, because stability is lost if the state leaves this set. A point in the state space belongs to the reachability set if there exists a feasible control signal such that the state of the system is brought to the origin. The set of all such points constitutes the reachability set. The role of reachability sets was discussed also in Section 1.5.

In the following it will be assumed that the control input of the system (10.4) is subject to the following standard saturation

$$\text{sat}_{u_0}(u) = \begin{cases} u_0 & u \geq u_0 \\ u & -u_0 \leq u \leq u_0 \\ -u_0 & u \leq -u_0. \end{cases} \tag{10.6}$$

The reachability set of the planar pendulum was studied in [Brufani, 1997], where the reachability set for $|x_1| \leq \frac{\pi}{2}$ was calculated. For completeness, this derivation is given below, as well as its extension to the case when $|x_1| \leq \pi$.

We first notice that for a constant acceleration u_0 there is an equilibrium at $x_1^0 = \arctan u_0$ and $x_2^0 = 0$. When the acceleration has the constant value u_0, the equation of motion of the pendulum can be integrated to give

$$\begin{aligned} \frac{1}{2}x_2^2 &= -\cos x + u_0 \sin x_1 + C \\ \frac{1}{2}x_2^2 &= \frac{-\cos x_1 \cos x_1^0 + \sin x_1 \sin x_1^0}{\cos x_0} + C \\ &= -\frac{\cos(x_1 + x_1^0)}{\cos x_1^0} + C. \end{aligned} \tag{10.7}$$

The reachability set is essentially given by (10.7). To explore the details we will consider two cases.

Case 1: $|x_1| < \pi/2$

This case correspond to the situation where the pendulum is never allowed to pass the horizontal plane through the pivot. In this case, the boundaries of the reachability set is given by the trajectories through the unstable equilibria $(x_1^0, 0)$ and $(-x_1^0, 0)$. Linearization around the equilibria shows that they are saddles. The trajectories are the stable solutions of (10.7)

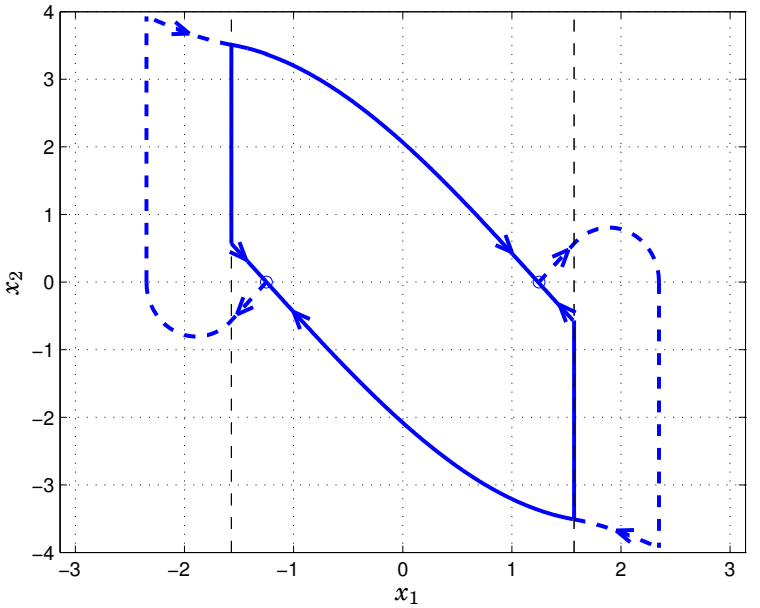


Figure 10.2 Boundaries of the reachability regions for $|x| < \pi/2$ in full lines and for $|x_1| \leq \alpha = 2.35$ in dashed lines. The circles show the unstable equilibria and the arrows show the direction of the vector field. In this case $u_0 = 3$.

through the equilibria. This gives $C = \pm 1/\cos x_1^0$ and the expressions

$$f^+_{\pi/2}(x_1) = \begin{cases} \sqrt{2\dfrac{1-\cos(x_1 - x_1^0)}{\cos x_1^0}}, & -\dfrac{\pi}{2} \leq x_1 \leq x_1^0 \\ -\sqrt{2\dfrac{1-\cos(x_1 - x_1^0)}{\cos x_1^0}}, & x_1^0 \leq x_1 \leq \dfrac{\pi}{2} \end{cases} \tag{10.8}$$

for the upper boundary of the reachability region. Because of symmetry, the lower boundary is the mirror of the upper boundary, hence

$$f^-_{\pi/2}(x_1) = -f^+_{\pi/2}(-x_1). \tag{10.9}$$

Figure 10.2 shows the reachability region for this case with $u_0 = 3$ in solid curves.

Case 2 $|x_1| < \pi$

It follows from the analysis in [Åström and Furuta, 2000] that if the acceleration is larger than $4/3$ it is possible to have a reachability set which allows the pendulum to go below the horizontal plane through the pivot. Assume that the angle is restricted to $-\alpha \leq x_1 \leq \alpha$. This requires that the acceleration of the pendulum is sufficiently large to swing up a pendulum at rest from the angle α. The energy analysis in [Åström and Furuta, 2000] gives the following relation between α and u_0.

$$\alpha = \pi - \arctan u_0 + \arccos\Big(\frac{2u_0}{\sqrt{1+u_0^2}-1}\Big) \tag{10.10}$$

It is somewhat counter-intuitive that the smallest acceleration $u_0 = 4/3$ is obtained for the largest α, i.e. $\alpha = \pi$. Smaller values of α require larger acceleration.

To find the reachability set we first observe that the boundary of the reachability region goes through the point $x_1 = \pm\alpha$, $x_2 = 0$. In the case of $\alpha > 0$, the acceleration is positive for $|x_1| > \pi/2$ and negative for $|x_1| < \pi/2$. Using the energy equation (10.7) and matching the parameter C to the boundary conditions we obtain the following expression for the upper boundary $x_2 = f^+(x_1)$ of the reachability region.

$$f_\alpha^+(x_1) = \begin{cases} \sqrt{-2\dfrac{\cos(x_1 + x_1^0)}{\cos x_1^0} + C_1} & \text{if } \dfrac{\pi}{2} \leq x_1 \leq \alpha \\[2ex] \sqrt{-2\dfrac{\cos(x_1 - x_1^0)}{\cos x_1^0} + C_2} & \text{if } -\dfrac{\pi}{2} < x_1 < \dfrac{\pi}{2} \\[2ex] \sqrt{-2\dfrac{\cos(x_1 + x_1^0)}{\cos x_1^0} + C_3} & \text{if } -\alpha \leq x_1 \leq -\dfrac{\pi}{2} \end{cases} \tag{10.11}$$

where

$$C_1 = 2\frac{\cos(\alpha + x_1^0)}{\cos x_1^0} \tag{10.12}$$

$$C_2 = 2\frac{\cos(\pi/2 - x_1^0)}{\cos x_1^0} - 2\frac{\cos(\pi/2 + x_1^0)}{\cos x_1^0} + C_1 \tag{10.13}$$

$$C_3 = 2\frac{\cos(-\pi/2 + x_1^0)}{\cos x_1^0} - 2\frac{\cos(-\pi/2 - x_1^0)}{\cos x_1^0} + C_2 \tag{10.14}$$

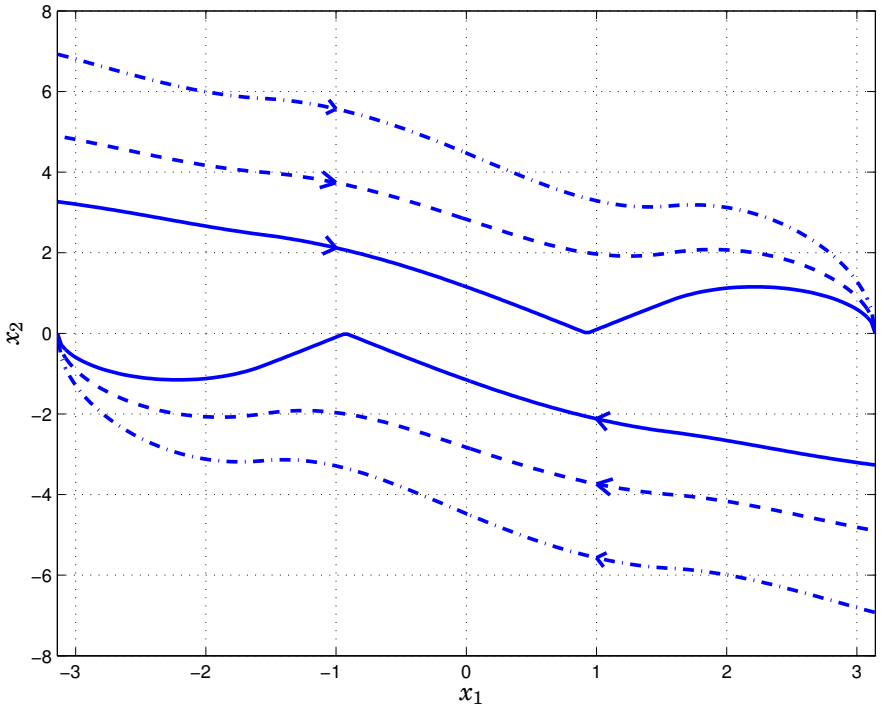


Figure 10.3 Boundaries of the reachability regions for $|x_1| \leq \pi$ and $u_0 = 4/3$ (solid), $u_0 = 3$ (dashed) and $u_0 = 6$ (dash-dotted).

The lower boundary of the reachability region, $f_\alpha^-(x_1)$, is defined as in Equation (10.9). In Figure 10.2, the reachability region in the case of $u_0 = 3$ and α given by (10.10) is shown in dashed curves. Figure 10.3 shows reachability regions for $|x_1| \leq \pi$. Notice that the region grows for larger values of u_0. The size of the reachability set depends on the saturation limit, u_0, and on the permissible range of x_1. The entire state space is the reachability set if there are no restrictions on x_1.

10.4 A Stabilizing Controller

As a first step towards the design of a controller enabling tracking of reference commands for the cart velocity, a stabilizing controller for the pendulum states x_1 and x_2 will be developed. It is clear from the previous analysis that such a controller may only stabilize the system in (a subset

of) the reachable region, which is *a priori* known.

In the following, the case of $|x_1| \leq \pi/2$ will be considered. A simple but effective way to design such a controller is to use a linear design method based on the linearized model (10.5), resulting in a linear control law

$$u = \text{sat}_{u_0}(-l_1 x_1 - l_2 x_2), \tag{10.15}$$

which locally stabilizes also the non-linear system (10.4). It is not clear that such a controller also achieves semi-global stabilization. Using an LQ design, however, it is possible to prove semi-global stability, given that the controller fulfills the following two sufficient conditions: Firstly, the region of the state space where the controller operates linearly must be entirely contained in the reachability set. Secondly, the solution of the algebraic Riccati equation, P, should produce a Lyapunov function candidate, $V(x) = x^T P x$, such that there is a sufficiently large region defined by $V(x) \leq c$ in which $\dot{V}(x) < 0$. From the Lyapunov stability theorem it follows that $\{x : x^T P x \leq c \,|\, x \to 0\}$. The first condition is to make sure that close to the boundaries of the reachability set, the controller is saturated. In this situation, trajectories will approach the center of the reachability set and the linear region. The second condition is to ensure that all trajectories starting outside of the ellipse defined by $x^T P x \leq c$ will actually enter it. It is not difficult to find a controller that fulfills the requirements. A typical situation is shown in Figure 10.4. As can be seen, an ellipse defined by $x^T P x \leq c$ (bold) can be fitted inside the region in which $\dot{V} \leq 0$ (dash-dotted bold). Further, the controller operates in linear mode in the region defined by the non-bold dash-dotted lines. It hence follows that all trajectories starting inside of the reachability region (dashed bold) will inevitably enter the ellipse. Semi-global stability follows. By tuning the weights in the LQR-design, it is possible to shape the local behavior of the controller, and also obtain an ellipse that is better aligned with the reachability region. However, stability and the region of attraction of the controller will be unaffected as long as the two requirements stated above are fulfilled.

10.5 Tracking

In this section we will design a controller that permits manual control of the cart velocity while stabilizing the pendulum. Consider the control law

$$u = \text{sat}_{u_0}(-l_1 x_1 - l_2 x_2 + m), \tag{10.16}$$

where m represents the tracking term which will be defined below. First assume that m is constant. The equilibria of the perturbed system are

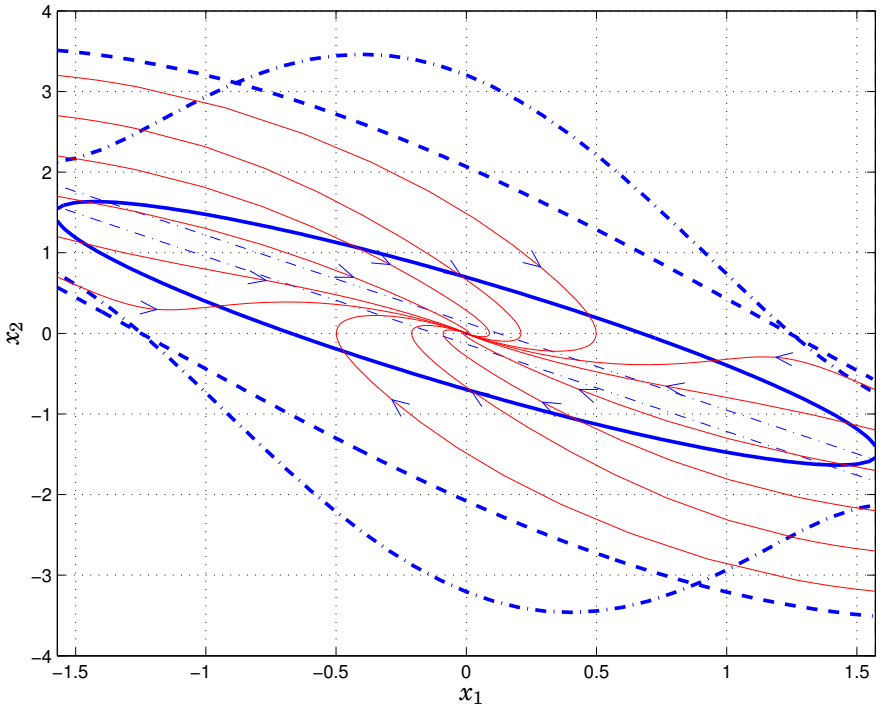


Figure 10.4 The region of attraction of the linear saturated controller.

then given by the equation

$$\tan x_1 = \text{sat}_{u_0}(-l_1 x_1 + m). \tag{10.17}$$

The curve representing the saturation is shifted horizontally when the manual control is changed. The number of equilibria then depends on m. In Figure 10.5, there are three equilibria marked by circles. The middle equilibrium is (controlled) stable and the others are unstable. For large positive or negative values of m there is only one equilibrium which is unstable. A necessary condition for semi-global stability is that the system has three equilibria for a constant m. To maintain stability it is necessary that the manual control actions are limited. From the point of view of performance it is desirable that the limits on the authority of manual control are as wide as possible. From the previous analysis, it is clear that a constant angle x_1^0, corresponds to a constant acceleration u^0. To enable fast tracking, i.e., large acceleration towards the reference velocity,

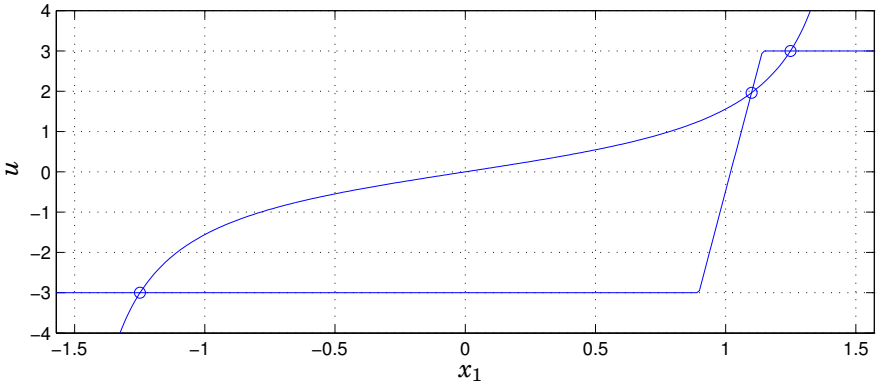


Figure 10.5 Equilibria for the system subject to saturated control and constant reference tracking term m.

it is thus desirable to allow for large values of m. By selecting the tracking term m as

$$m = \text{sat}_a(l_3(r - x_3)) \tag{10.18}$$

where r is the reference value of the cart velocity state x_3, it is possible to capture the trade-off between stability and performance. The feedback gain l_3 is conveniently calculated using LQR-design, that gives the desired local behavior. The choice of the saturation limit a is guided by the following lemma:

LEMMA 10.1
Consider

$$a = \begin{cases} a^+(x_1) = l_2(f^-_{\pi/2}(x_1) + d) - u_0 + l_1 x_1 \\ a^-(x_1) = -l_2(f^+_{\pi/2}(x_1) - d) - u_0 - l_1 x_1 \end{cases} \tag{10.19}$$

where $a^+(x_1)$ and $a^-(x_1)$ are the positive and negative saturation limits of (10.18) and $0 \leq d \leq d_{max}$. Then the region bounded by $f^+_{\pi/2}(x_1) - d$ and $f^-_{\pi/2}(x_1) + d$ is positively invariant, i.e., trajectories starting in this region will remain in it regardless of the reference value r.
Proof: The proof is a straight forward application of Nagumos theorem, stating that for a closed set $\mathcal{S} \in R^n$, $\mathcal{S}$ is positively invariant for the system $\dot{x} = f(x)$, if and only if the field $f(x)$ points to the interior of $\mathcal{S}$ for all $x \in \partial\mathcal{S}$. See [Blanchini, 1999] for details.

The controller (10.16) with saturation limits defined by (10.19), operates in saturated mode whenever $x_2 \geq f^+_{pi/2}(x_1) - d$ or $x_2 \leq f^-_{pi/2}(x_1) + d$. It then follows from a phase plane argument that trajectories starting at

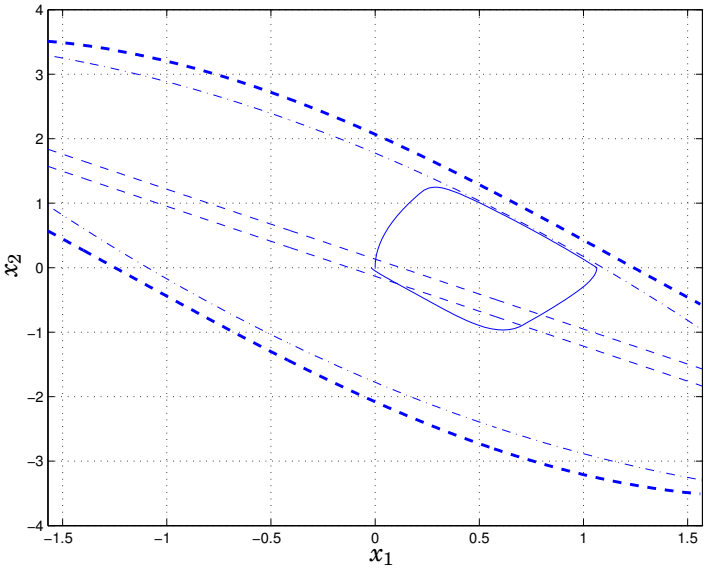


Figure 10.6 Phase portrait of the system (10.4) subject to tracking control.

the boundary curves $f^{+}_{pi/2}(x_1) - d$ and $f^{-}_{pi/2}(x_1) + d$ will approach the interior of the region. The same argument can be applied for the vertical line segments bounding the region at $x_1 = \pm\pi/2$. □

REMARK 10.1
To avoid an overly conservative design, the saturation limit a is dependent on the angle x_1. □

REMARK 10.2
The value of d is used to control the size of the invariant region, yielding a safety margin for robustness. However, the region does not exist if d is too large. □

REMARK 10.3
The boundary functions $f^{+}_{\pi/2}(x_1)$ and $f^{-}_{\pi/2}(x_1)$ used in (10.19) can be approximated by simpler expressions, as long as the condition of Nagumos theorem are fulfilled. □

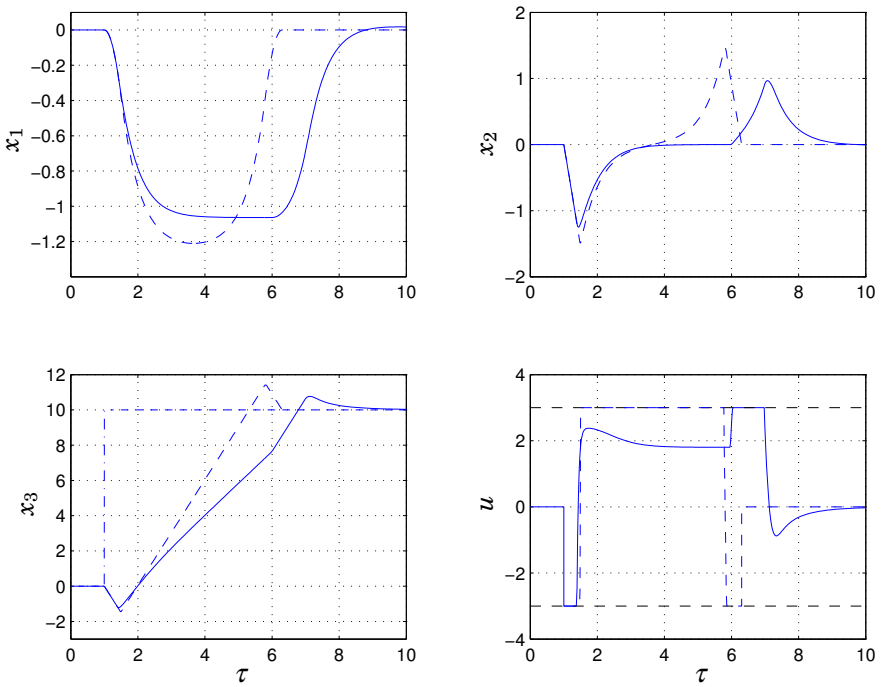


Figure 10.7 Trajectories for a step reference change for the proposed controller (10.16) (solid) and the time-optimal solution (dashed).

Figure 10.6 shows the phase portrait resulting when a step reference sequence is applied to the system (10.4), controlled by the controller (10.16)-(10.19). As can be seen, the state remains in the specified invariant set. Notice that the linear region marked by dashed lines is small compared to the invariant region. A strategy that avoids saturation is thus very conservative. The tracking behavior of x_3 in Figure 10.7 is reasonable as shown by a comparison with minimum time trajectories. The time optimal trajectories give a faster response for large set point changes, but lacks the robustness of the proposed feedback controller.

10.6 Extensions

The analysis above is valid for the system (10.4), where limited acceleration of the pivot was assumed. The true problem, however, is to device a controller for the system (10.1), assuming input saturation on F, i.e.,

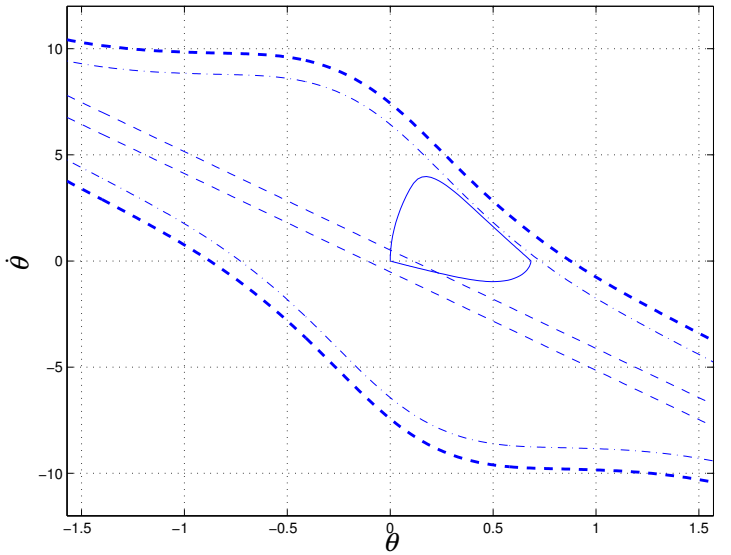


Figure 10.8 Phase portrait of the system (10.1) subject to tracking control.

limited force. This problem can be solved using insight gained from the analysis in the previous sections.

The reachability set of (10.1) subject to the input nonlinearity (10.6) can be found numerically through simulation. The set is indicated in Figure 10.8 in bold curves. Using this set, the controller (10.16) with saturation limits (10.19) can be employed. As previously, the controller renders the region defined by the boundary functions $f^+_{\pi/2}$ and $f^-_{\pi/2}$ and the parameter d invariant. Notice that the invariance argument holds also for approximations of the boundary functions, see Remark 10.3.

Figure 10.8 shows a typical phase portrait. The pendulum states do not leave the invariance region marked by dash dotted curves. Figure 10.9 shows the step response of the system. The minimum time solution is shown in dashed curves. Notice the different time scales in Figures 10.7 and 10.9, which are due to scaling. The following numerical values of the parameters of the system (10.1) were used in the simulations: $m_p = 0.3$ kg, $l = 0.5$ m, $m_c = 0.2$ kg, $g = 9.81\text{m/s}^2$ and $J_p = m_p l^2$.

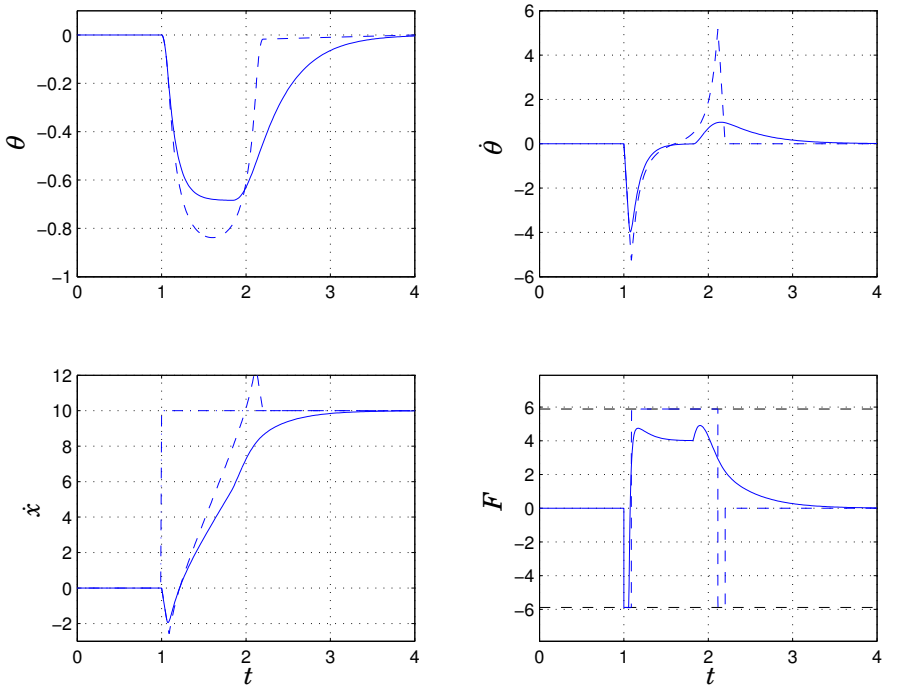


Figure 10.9 Trajectories for a step reference change for the proposed controller (solid) and the time-optimal solution (dashed).

10.7 Conclusions

An explicit characterization of the reachability set for an inverted pendulum on a cart subject to limited acceleration of the pivot has been given. A controller enabling tracking of constant pivot velocity references while stabilizing the pendulum has been proposed. A single parameter, d, is used to trade performance and robustness of the controller. The controller has also been generalized to the case of the actual pendulum system subject to a limited force acting on the cart.

11

Design and Control of YAIP

11.1 Introduction

Inverted pendula have been the subject of numerous studies in automatic control, from the 1940s and onwards. In this chapter, a variation on the theme inspired by the well known Segway robot, [Segway Inc., 2006], is described.

Inverted pendula mounted on two wheels have been reported in several papers during the last years, see e.g. [Grasser *et al.*, 2002]. Also, there are a number of commercial robots on the market. This chapter describes the design and control of a prototype Segway-type robot, intended as a platform for research and teaching. The design problem is challenging, considering that a primary objective has been to use components available at a reasonable price, while maintaining acceptable performance. Key design issues include selection of drives and sensors as well as electronics and choice of microprocessors for signal processing and control.

The resulting robot offers several interesting features regarding sensors, control design, distributed control systems and implementation. The robot is equipped with two drives for actuation, a rate gyro and an accelerometer for measuring the angle and angular velocity of the pendulum body, and encoders for measuring the angle of the wheels. Signal processing and control algorithms are distributed amongst three microprocessors; one for each of the drives and one responsible for stabilizing control. This layout enables hierarchical control design, but also complicates implementation, since processor communication must be considered.

The purpose of this chapter is to describe the robot design and to report experiences from the design process. The chapter gives two main contributions. Firstly, a design description of the robot is given. Secondly, a novel approach to angular velocity estimation based on analog differentiation of

encoder signals is presented.

The chapter is organized as follows. In Section 11.2, the robot design is briefly described. Sections 11.3 and 11.4 treat selection of sensors and associated algorithms. In Section 11.5, a dynamic model of the system is presented. In Section 11.6 control strategies are described and in Section 11.7 implementational issues are covered. Section 11.8 describes results from experiments, and finally, in Section 11.9, conclusions and final remarks are given.

11.2 System Design

A schematic picture of the robot is shown in Figure 11.4. The robot consists of the pendulum body, which is attached to axes at which the two wheels are mounted. The pendulum body incorporates two DC-motor drives, transmission, sensors and several circuit boards hosting the micro-processors and sensor related electronics such as filters and amplifiers.

Mechanical Design

The actual pendulum body is built using FAC system's meccano/errecter set [FAC-system, 2006], complemented by some custom made aluminium parts, e.g. the parts used for mounting the drives. The robot is depicted in Figure 11.1.

Sensors and Actuators

In order to enable stabilizing control, the robot must be equipped with appropriate sensors and actuators. In fact, this issue constitutes perhaps the most challenging task in the robot design process. The choices concerning sensors and actuators, and associated algorithms, will inevitably impose constraints on achievable performance.

A pair of DC-drives were used to actuate the robot. Basically, the main trade-off is that between weight and torque, where higher torque comes at the price of more expensive and heavier drives. In order to increase torque, drives equipped with gear-boxes are attractive choices. A standard solution, which is also widely available, is then planetary gear-boxes. However, such devices introduce back-lash, which severely degrades control performance. This was confirmed by early designs in the project, as well as in [Grasser *et al.*, 2002]. Therefore, a gear-box was constructed using timing belts (ratio 4.1:1), which was found to effectively eliminate the back-lash. The drives used were a pair of Faulhaber 3863012C, which produced the maximum torque 0.45 Nm each, when connected to the gear-box. The choice of drives was guided by a preliminary simulation study.

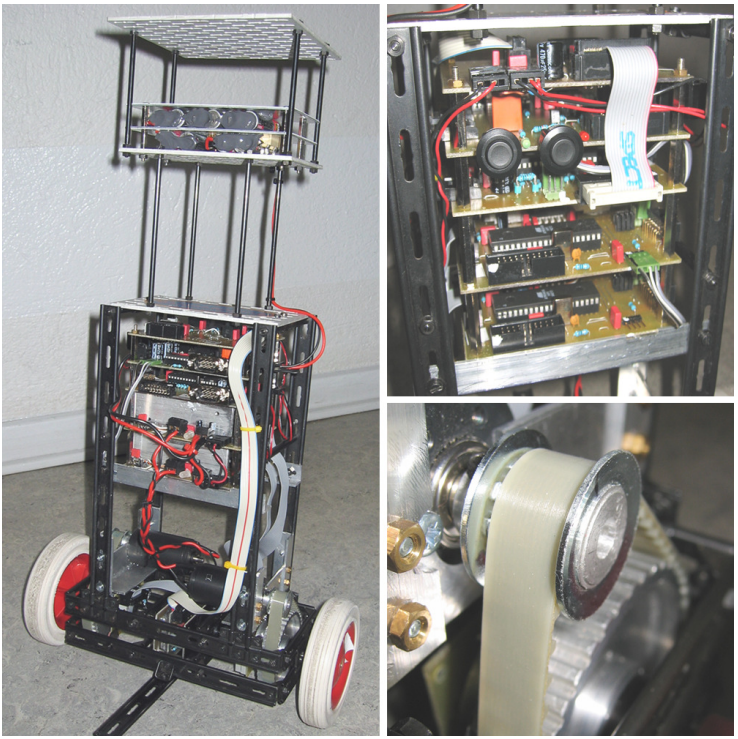


Figure 11.1 YAIP – the inverted pendulum robot.

The use of a rate gyro in combination with an accelerometer is a standard configuration for inertial sensing, which is needed in this application to estimate the angle and angular velocity of the pendulum body, see e.g. [Grewal *et al.*, 1991]. The selected accelerometer was an ADX202 and the rate gyro was an ADXRS300, both from Analog Devices. The ADX202 is a two axis accelerometer, although only one channel is needed in this application.

In order to measure the wheel angles, encoders were used. Since good angle and angular velocity information is crucial for stabilization of the system, the analog encoder signals were sampled at a frequency high enough to enable angle estimation based on the analog wave forms produced by the encoders. In addition, in order to further increase the accuracy of the angular velocity measurements, the derivative of the encoder signals were produced using analog filters. The differentiated signals were then sampled together with the original encoder signals. The idea of using

analog differentiated encoder signals for velocity estimation is not new, see e.g. [Gabor, 1974]. However, the approach presented in this chapter, which is based on tabulated mappings calculated from the encoder wave-forms, has not, to the best knowledge of the authors, been reported previously.

Micro-processors

The micro-processor Atmel MEGA16 was chosen for signal processing and control algorithms. This processor is a convenient choice since it offers several important features on one chip, such as A/D conversion, RS232 communication, PWM-signal generation for motor control and a protocol for inter-processor communication, I2C. In addition, there is a free C-compiler for the AVR architecture, `avr-gcc`. The robot was equipped with three Atmel MEGA16 processors; one for each drive and one for stabilizing control and coordination.

11.3 Encoder Processing

Angle Estimation

The use of encoders is a standard method for estimation of angle and angular velocity. The idea is simple. By attaching a disc with alternating transparent and solid fields to the axis of rotation, and mounting a light source, commonly a LED, on one side of the disc and a photocell on the other, the latter component will produce a periodic waveform as the disc rotates with constant velocity. If two such pairs of a LED and a photocell are placed a quarter of a period apart, the angle of rotation can be calculated from the resulting waveforms. Alternatively, the LED and photocell pair may be placed on the same side of the disc, if a solid disc with light and dark fields is used. This configuration was used on the robot presented here. Commonly, the encoder signals are decoded digitally as logic high or low, which yields a resolution of twice the number of fields per disc revolution. This approach works well for high angular velocities, but generally poorly for low velocities. This is because in the latter case, few fields per time unit are passed which gives rise to severe quantization effects.

A different approach, described in [Venema, 1994] and references therein, is to sample the analog encoder signals. Using this technique, the shape of the waveforms can be used to increase the resolution of the angle measurement significantly. It is a common misconception that encoder signals

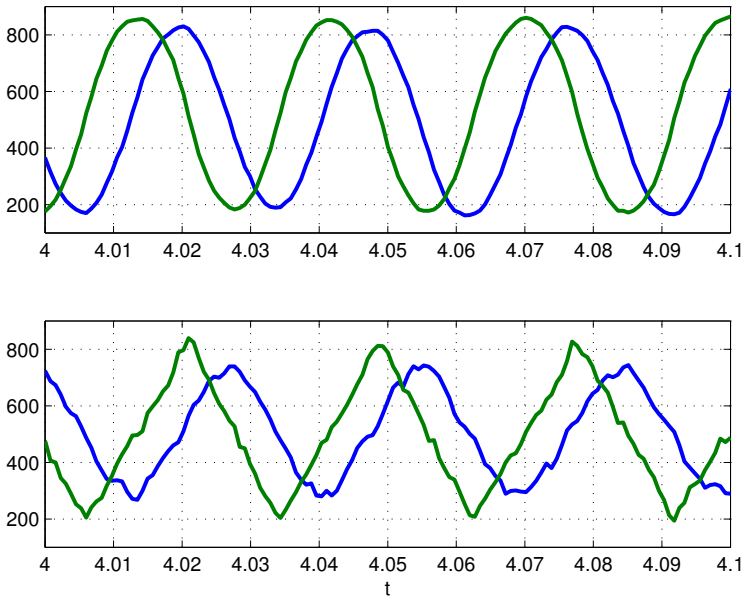


Figure 11.2 Typical encoder waveforms. The upper plot shows the original encoder signals and the lower plot shows the signals obtained through analog differentiation.

are given by

$$\begin{aligned} \bar{z}_x(\varphi) &= V_x \cos(\varphi N) \\ \bar{z}_y(\varphi) &= V_y \sin(\varphi N) \end{aligned} \tag{11.1}$$

where φ is the disc angle, $\bar{z}_x$ and $\bar{z}_y$ are the encoder signals and N is the number of solid (or transparent) fields of the disc. Mainly, there are two reasons for the encoder signals to be non-ideal. Firstly, the encoder signals are usually not shifted exactly by $\pi/2/N$ rad. Secondly, the waveforms often deviates from the ideal sinusoidal shape.

In order to increase the position estimation accuracy, these phenomena could be compensated for, e.g. using methods presented in [Zimmerman *et al.*, 2006] and [Venema, 1994]. We have used a method that explores the ideas presented in [Venema, 1994], and the method is described in summary in the following. We assume that the encoder waveforms are given by

$$\begin{aligned} z_x(\phi) &= g_x(\varphi N) = g_x(\phi),\ g_x(\phi) = g_x(\phi + 2\pi) \\ z_y(\phi) &= g_y(\varphi N) = g_y(\phi),\ g_y(\phi) = g_y(\phi + 2\pi) \end{aligned} \tag{11.2}$$

where ϕ is the encoder angle, related to the disc angle as $\phi = \varphi N$. Further, z_x corresponds to the cosine-like waveform and z_y corresponds to the sine-like waveform. See Figure 11.2 for typical waveforms.

Given measurements of z_x and z_y we would like to calculate an estimate of the angle, ϕ. In the ideal case, the expression

$$\hat{\phi} = \arctan\left(\frac{V_x \bar{z}_y}{V_y \bar{z}_x}\right) \tag{11.3}$$

gives the correct result. In the non-ideal case, however, the arctan-function will not produce a correct result. Now, instead of using the arctan-function, an equivalent mapping function may be used, given by

$$\hat{\phi} = q\left(\frac{z_y}{z_x}\right) \tag{11.4}$$

where q is calculated using the actual encoder waveforms rather than the ideal ones.

In order to calculate the function q, the actual encoder waveforms need to be estimated. This may be done using signal data recorded while rotating the encoder disc at constant velocity. By calculating the average of all recorded encoder periods, estimates $\hat{g}_x(\phi)$ and $\hat{g}_y(\phi)$, $0 \leq \phi \leq 2\pi$ may be obtained. The mapping function q is then calculated from

$$q^{-1}(\phi) = \frac{\hat{g}_y(\phi)}{\hat{g}_x(\phi)} \tag{11.5}$$

where $q^{-1}(\phi)$ denotes the inverse of $q(\phi)$. In Figure 11.3, the function $q(\phi)$ is shown in solid for a particular encoder configuration. The dashed curves shows the angle estimation given by expression (11.3). As can be seen, there is a significant difference between the angle estimates obtained if ideal curves are assumed, as compared to the case when the actual encoder waveforms are used as a basis for the estimation. Notice that in order for the mapping from z_y/z_x to $\hat{\phi}$ to be unique, the current quadrant, i.e. the signs of the encoder signals, must be considered.

Angular Velocity Estimation

In addition to accurate angle estimates, it is desirable to have accurate angular velocity information. The standard approach to this problem is to apply a discrete-time differentiation filter to the sequence of angle estimates. While this approach works well for high velocities, the performance for low velocities is poor. The main reason for this is that the temporal discretization, resulting from sampling, is difficult to compensate for using

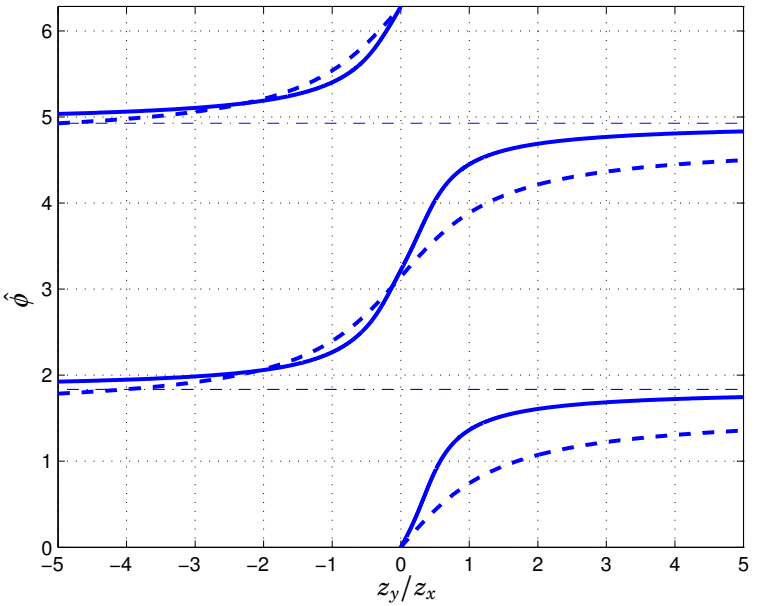


Figure 11.3 The solid curves show the function $q(\phi)$. For comparison, the dashed curves shows the angle estimation given by expression (11.3).

digital filters. Therefore we propose derivation of the encoder signals using analog filters. The analog derivative signals may then be sampled and used to improve the accuracy of the angular velocity estimate. The main advantage of this approach is that the need for differentiation by means of digital filters is eliminated. Instead, this approach enables velocity estimation by means of mapping-based methods similar to that presented above for angle estimation.

Certainly, there are many ways to use the encoder signals in combination with the derivative signals to obtain angle and angular velocity estimates. In this section, a method producing promising results will be described. The main advantage of the proposed method is its simplicity and low computational complexity, which has been a key issue in this project. A different approach could be to extend the method presented in [Zimmerman *et al.*, 2006], which is based on an Extended Kalman Filter, to also include analog encoder derivative measurements.

Given the encoder waveform expressions (11.2), the time derivative

signals may be written

$$\begin{aligned} dz_x(\phi,\dot{\phi}) &= \alpha_x \frac{dg_x}{d\phi}(\phi)\dot{\phi} = \alpha_x g_x'(\phi)\dot{\phi} \\ dz_y(\phi,\dot{\phi}) &= \alpha_y \frac{dg_y}{d\phi}(\phi)\dot{\phi} = \alpha_y g_y'(\phi)\dot{\phi} \end{aligned} \tag{11.6}$$

were α_x and α_y represents the amplification of the analog derivation filters. Using these expressions, estimates of the angular velocity may be calculated from

$$\begin{aligned} \hat{\dot{\phi}}_x &= \frac{dz_x}{\alpha_x g_x'(\phi)} \\ \hat{\dot{\phi}}_y &= \frac{dz_y}{\alpha_y g_y'(\phi)}. \end{aligned} \tag{11.7}$$

Clearly, $\hat{\dot{\phi}}_x$ and $\hat{\dot{\phi}}_x$ can be expected to be reasonable velocity estimates when, respectively, $g_x'(\phi)$ and $g_y'(\phi)$ are not close to zero. A natural way of combining the estimates is then

$$\begin{aligned} \hat{\dot{\phi}} &= \frac{g_x'(\phi)^2}{g_x'(\phi)^2 + g_y'(\phi)^2} \frac{dz_x}{\alpha_x g_x'(\phi)} \\ &\quad + \frac{g_y'(\phi)^2}{g_x'(\phi)^2 + g_y'(\phi)^2} \frac{dz_y}{\alpha_y g_y'(\phi)} \\ &= \frac{g_x'(\phi)}{g_x'(\phi)^2 + g_y'(\phi)^2} \frac{dz_x}{\alpha_x} + \frac{g_y'(\phi)}{g_x'(\phi)^2 + g_y'(\phi)^2} \frac{dz_y}{\alpha_y} \\ &= w_x(\phi) dz_x + w_y(\phi) dz_y \end{aligned} \tag{11.8}$$

where the weights $w_x(\phi)$ and $w_y(\phi)$ are readily calculated using the average waveforms $\hat{g}_x(\phi)$ and $\hat{g}_y(\phi)$. In the expression (11.8), the angle ϕ is assumed to be known, which is not the case in the real application. Instead, the estimate of the angle is used, which yields

$$\hat{\dot{\phi}} = w_x(\hat{\phi}) dz_x + w_y(\hat{\phi}) dz_y. \tag{11.9}$$

Filtering

The estimates obtained from the algorithms described above are subject to noise. There are several sources of noise, including noise at the original encoder signals which is propagated to the estimated variables and noise introduced by the arithmetic operations. The latter source of noise may result if the algorithms are implemented using fixed-point arithmetic, which is common when computing power is scarce. Notice, for example,

that a noisy angle estimate will propagate to the angular velocity estimate through the weights w_x and w_y.

In order to reduce the effects of noise, the estimated variables may be filtered. Assuming that a Kalman filter is used, there are two main different approaches for construction of the filter, differentiated by the structure of the underlying dynamic model. It could be argued that the use of a full dynamic model of the robot is advantageous, since it captures the full behavior of the system. However, such a model may be very complex, and it may also contain severe non-linearities, e.g. friction, that makes it difficult to use as a basis for a Kalman filter. Instead, a model expressing the kinematic relationships of the variables has been used. In [Zimmerman *et al.*, 2006], the following stochastic model is proposed

$$dx = Adt + dw = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -\alpha \end{pmatrix} dt + dw \tag{11.10}$$

where $x = (\phi, \dot{\phi}, \ddot{\phi})^T$, α is the inverse decorrelation time of the acceleration and dw is vector of zero-mean Wiener processes with incremental correlation matrix $R_{1c} = diag(0, 0, \sigma^2)$. The continuous time model (11.10) is readily sampled using the sampling interval $h = 0.00146$ ms, yielding a discrete time equivalent

$$\begin{aligned} x(kh + h) &= \Phi x(kh) + w_d(kh) \\ y(kh) &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} x(kh) + v(kh) \end{aligned} \tag{11.11}$$

where $\Phi = e^{Ah}$ and the covariance matrix of w_d, R_1, is given by

$$R_1 = \int_0^h e^{A\tau} R_{1c} e^{A^T \tau} d\tau$$

The sampling expressions can be found e.g. in [Åström and Wittenmark, 1997]. Also, the measurement noise process v has been introduced. Assuming that the covariance matrix of v, R_2, can be estimated from data, the Kalman filter has two tuning knobs; α and σ^2. These parameters can be used to set the bandwidth of the filter – lower values of σ^2 and/or α, yields a filter with lower bandwidth. The calculation and implementation of the Kalman filter is described in [Åström and Wittenmark, 1997]. The Kalman filter parameters α and σ^2 were set to 50 and 5×10^5.

A preliminary evaluation of the improvement of the angular velocity estimation produced by the proposed scheme showed improved precision

compared to an estimate produced by a digital filter. However, in order to fully evaluate the properties of the method, more studies need to be performed. This, however, is beyond the scope of this chapter.

11.4 Gyro and Accelerometer Processing

The problem of obtaining accurate estimates of the angle and angular velocity of the pendulum body, θ and $\dot{\theta}$, is important in order to enable stabilizing control. The accelerometer, which is mounted on the pendulum body, takes into account the acceleration due to gravity, and may be used as an indicator of the angle of the body. This signal is, however, corrupted by high frequency disturbances. The rate gyro, on the other hand, produces a signal proportional to the angular velocity. While this signal has good high frequency properties, it suffers from drift. The two sensors in combination, however, may be used to obtain angle and angular velocity estimates.

A method developed for the combination of an inclinator and a rate gyro based on filters designed by shaping of frequency responses is presented in [Baerveldt and Klang, 1997]. Alternatively, the problem may be addressed by postulating a Kalman filter based on the system

$$\begin{aligned} \dot{x} &= \begin{pmatrix} 0 & -1 \\ 0 & 0 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix} y_{gyro} \\ y_{acc} &= \begin{pmatrix} 1 & 0 \end{pmatrix} \end{aligned} \tag{11.12}$$

where x_1 and x_2 ($x = (x_1, x_2)^T$) represents the body angle, θ, and the gyro signal offset respectively. The bandwidth of the Kalman filter is tuned by adjusting the measurement and process noise covariances.

The choice of method does not seem to be critical – both methods work well in practice. However, the Kalman filter method was used since it gives an explicit estimate of the gyro offset.

11.5 Dynamic System Model

Consider the schematic picture of the pendulum robot shown in Figure 11.4. Let the angle of the pendulum body relative to the vertical plane be θ, and let the angle of rotation of the wheels be φ. The applied torque is denoted by τ. Derivation of the equations of motion of the system by

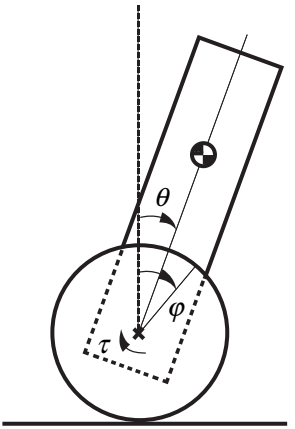


Figure 11.4 A schematic picture of the pendulum robot.

means of the Euler-Lagrange equations gives the system:

$$\begin{pmatrix} J_p & m_p l r_w \cos\theta \\ m_p l r_w \cos\theta & J_w + m_w r_w^2 + m_p r_w^2 \end{pmatrix} \begin{pmatrix} \ddot{\theta} \\ \ddot{\varphi} \end{pmatrix} + \begin{pmatrix} -m_p g l \sin\theta \\ -m_p l r_w \dot{\theta}^2 \sin\theta \end{pmatrix} = \begin{pmatrix} -\tau \\ \tau \end{pmatrix} \tag{11.13}$$

where $J_p = 0.16$ Nm is the moment of inertia of the body w.r.t. the pivot, $m_p = 2.94$ kg is the mass of the pendulum, $l = 0.168$ m is the distance from the pivot to the center of mass of the body, $m_w = 0.46$ kg, $r_w = 0.0515$ m and $J_w = 0.00045$ Nm are the mass, radius and moment of inertia of the wheel assembly respectively and $g = 9.81$ m/s^2 is the acceleration due to gravity.

The stabilizing control strategy described in Section (11.6) requires a linear state space model. Introducing the state vector $x = (\theta, \dot{\theta}, \varphi, \dot{\varphi})^T$, and linearizing the model (11.13) around $x^0 = (0,0,0,0)^T$ gives

$$\dot{x} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ \frac{\alpha\delta}{\alpha\beta-\gamma^2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{\gamma\delta}{\alpha\beta-\gamma^2} & 0 & 0 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ -\frac{\alpha+\gamma}{\alpha\beta-\gamma^2} \\ 0 \\ \frac{\beta+\gamma}{\alpha\beta-\gamma^2} \end{pmatrix} \tau \tag{11.14}$$

where $\alpha = J_w + m_w r_w^2 + m_p r_w^2$, $\beta = J_p$, $\gamma = m_p l r_w$ and $\delta = m_p g l$.

The input of the model (11.13) is the torque applied by the drives, τ. However, the actual control signal is the commanded voltage to the drives. A simple motor model relating torque, voltage and angular velocity of the motor was then introduced. Assuming that the current dynamics is fast compared to the rest of the system, the following relation holds

$$\tau = -\beta_1(\dot{\varphi} - \dot{\theta}) + \beta_2 u \tag{11.15}$$

where $\beta_1 = 0.015$ and $\beta_2 = 0.27$ are parameters calculated from the datasheet of the drives, and u is the applied voltage. The negative term results from the back EMF produced by a DC-motor in motion. Notice that it is the *relative* angular velocity between the body and the wheels that gives rise to the back EMF.

11.6 Stabilizing Control

Since estimates of all states of the robot are available, state feedback is conveniently used to stabilize the system. The control law is then given by

$$u = -Lx. \tag{11.16}$$

where the feedback vector L was calculated using LQR design, see e.g. [Åström and Wittenmark, 1997].

In addition to the stabilizing controller (11.16), it was necessary to introduce an additional controller responsible for control of the robot heading. If the heading is not controlled, the robot will inevitable start rotating, resulting in lost stability. While the heading dynamics is not included in the model (11.13), the dynamics relating the difference in commanded voltages to the drives and difference in wheel velocities is well approximated by an integrator. A simple, yet effective, strategy is then to introduce a proportional controller acting on the difference in the angular velocities between the wheels,

$$u_d = -K(\dot{\varphi}_a - \dot{\varphi}_b) \tag{11.17}$$

where K is the controller gain and $\dot{\varphi}_a$ and $\dot{\varphi}_b$ are the angular velocity estimates of the two wheels respectively.

The control signals for the two motors are then given by the relations

$$\begin{aligned} u_a &= \frac{1}{2}(u + u_d) \\ u_b &= \frac{1}{2}(u - u_d). \end{aligned} \tag{11.18}$$

Friction Compensation

As most mechanical systems, the robot suffers from friction. In order to successfully stabilize the system, friction compensation proved necessary. There are several methods to compensate for friction, see, e.g. [Olsson *et al.*, 1998]. In this work, a friction compensation scheme based on Coulomb friction was used. The friction force may be approximated by the following model

$$F_f(\omega, u) = \begin{cases} F_c^+ & \dot{\omega} > 0 \\ F_c^+ & \dot{\omega} = 0, \quad u > F_c^+ \\ u & \dot{\omega} = 0, \quad F_c^- < u < F_c^+ \\ F_c^- & \dot{\omega} = 0, \quad u < F_c^- \\ F_c^- & \dot{\omega} < 0 \end{cases} \tag{11.19}$$

where $\omega = \dot{\varphi} - \dot{\theta}$ is the relative angular velocity between the body and the wheels. The coefficients F_c^+ and F_c^- are coefficients that may be calculated from simple experiments. An estimate of the friction force is then calculated using this model and added to the control signal computed from the control law (11.16). The control signal is then

$$v = -Lx + \hat{F}_f \tag{11.20}$$

11.7 Implementational Issues

The signal processing and control algorithms described in previous sections were implemented in the Atmel AVR micro-processors mounted on the robot. The software was implemented in C, compiled using `avr-gcc`, and downloaded to the processors.

Since the computing power of the chosen AVR model is limited, floating point implementation of algorithms was infeasible, which rendered fixed point implementations necessary. This introduces additional overhead for the programmer, and may degrade algorithm performance. However, all algorithms were successfully implemented using fixed point arithmetic, with little performance loss. The most sensitive algorithms proved to be the encoder angle and angular velocity estimation, where a slight decrease in accuracy was noted.

Since the control system was distributed amongst three microprocessors, communication and synchronization was necessary. For this matter, the I2C protocol, which is supported by the Atmel AVR, was used. I2C is a serial bus-based communication protocol, which uses only two

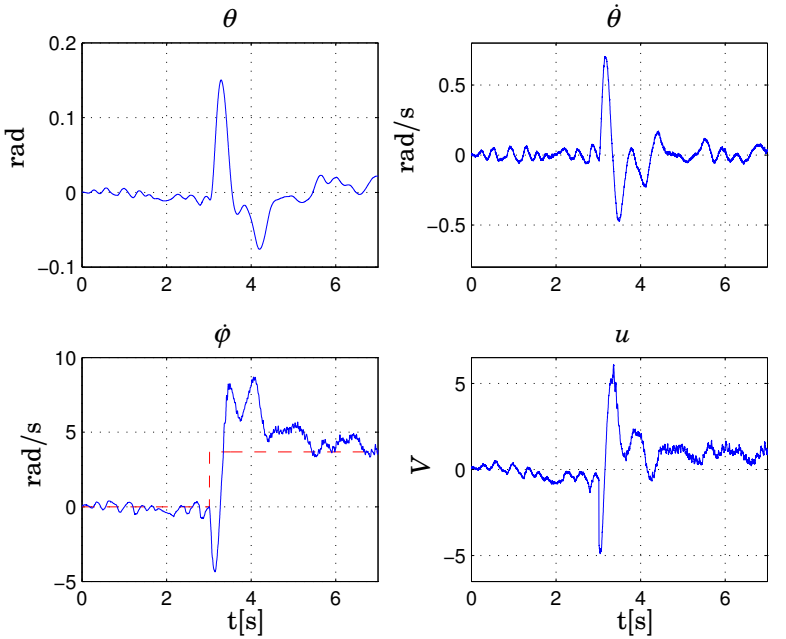


Figure 11.5 Stabilizing control and system response for a start command.

signal wires. Several units may communicate on the same bus, acting as masters and/or slaves. In this application, the processor responsible for stabilizing control acted as master and the two processors managing the encoders acted as slaves. During run-time, the master issues requests for measurement readings from, and transmits the commanded voltages to the slaves.

11.8 Experimental Results

Using the control law (11.20), the system may be stabilized. As can be seen in Figure 11.5, up to 3 s, the robot is indeed successfully stabilized, although there is a small, but visible, remaining limit cycle in the wheel velocity, $\dot{\varphi}$. This limit cycle is due to friction, which is not fully compensated for by the friction compensation scheme.

In order to evaluate the system response to start and stop maneuvers,

the modified control law

$$v = -l_1\theta - l_2\dot{\theta} - l_3(\dot{\varphi} - r) + \hat{F}_f \tag{11.21}$$

was introduced, where r represents the reference wheel velocity. At run time, the control law (11.20) is used when $r = 0$ and (11.21) otherwise. Using this strategy, drift is avoided in the case of $r = 0$.

In Figure 11.5, a start command, represented by a step in the wheel velocity reference r (dashed), is initiated at 3 s. As can be seen, after an initial overshoot, the wheel velocity settles at the desired value. While the transient response of the system controlled by the controller (11.21) may be improved, it should be noted that introduction of reference signals requires special attention, in particular for unstable systems, see [Åkesson and Åström, 2005]. This, however, is beyond the scope of this chapter.

11.9 Conclusions and Suggested Improvements

In this chapter, design and control of an inverted pendulum robot on two wheels has been described. The aim of this project has been to build a flexible platform suitable for teaching and research using low cost components. During the course of the project, several preliminary designs were evaluated. The two most important improvements motivated by experiences from early attempts were the use of drives without planetary gear-boxes (which eliminated back-lash) and high resolution wheel angle measurements using analog encoder signals.

A main challenge facing the control system designer is that of obtaining accurate estimates of the states of the system, while designing a stabilizing controller is comparatively straight forward. This is reflected by the fact that the main part of this chapter is devoted to state estimation algorithms.

The current design offers several opportunities for further improvements. Firstly, the encoder hardware design may be improved. The current implementation, based on reflection, is sensitive to irregularities of the reflecting surface. Also, if the distance between the LED/photo cell pair varies slightly, the resulting variation in the encoder signal offsets degrade the performance of the algorithms. Secondly, current control of the drives would be desirable, since this would effectively eliminate the uncertainties associated with these components. The resulting system would have a torque reference as input, which also gives a convenient structure of the control system. Another interesting feature would be a remote control facility, enabling a “driver” to maneuver the robot.

Finally, more sophisticated control structures targeting manual control aspects, would improve control performance. It was shown in Chapter 10 that reachability sets play a key role when controlling unstable systems subject to bounded inputs. The dynamics of the pendulum robot presented here is very similar to that of the planar pendulum analyzed in Chapter 10. Accordingly, those results are applicable also on the pendulum robot. An interesting extension of the control system for the pendulum robot would therefore be to apply the controller proposed in Chapter 10.

Conclusions

V

V

12

Conclusions and Future Work

In this thesis, contributions have been given in three areas, namely languages and tools for dynamic optimization, cases studies, and control of pendula. In this chapter, the main results are summarized, and some future research directions are discussed.

12.1 Languages and Tools for Dynamic Optimization

In Part II, a prototype Modelica compiler, the JModelica compiler, has been presented. The compiler is developed using the compiler construction tool JastAdd. It was shown how the mechanisms available in JastAdd, in particular reference attributes and rewrites can be used to perform name and type analysis, as well as flattening of Modelica models. Also, some of the strategies used in the JModelica compiler have been adopted from the JastAddJ Java compiler, but were adapted to the context of the Modelica language.

The JModelica compiler supports extensible compiler construction, both at the language level and at the implementation level. The concept of modular extensibility is very powerful, since it enables development of a core compiler supporting pure Modelica, which is independent of potential extensions. In addition, extended compilers based on the core compiler immediately benefit from improvements in the latter.

An extension of Modelica, Optimica, targeted at dynamic optimization, has been proposed. Optimica enables formulation of a large class of dynamic and static optimization problems, ranging from optimal control problems to multi-case formulations and parameter optimization. Some of the key features of Optimica are inequality and equality point and path constraints, free and fixed optimization interval and the possibility to for-

mulate problems with free initial conditions for states. In addition, Optimica supports annotations for a particular transcription method based on interpolation polynomials.

An Optimica compiler is under development. A prototype version of the Optimica compiler has been used in research, notably for the formulation and solution of the start-up optimization problem for the plate reactor in Chapter 9, in two master's thesis projects and in teaching. The Optimica compiler supports automatic transcription of dynamic variables by means of a direct collocation method, and generates AMPL code.

Future Work

There are several possible extensions of the work presented in Part II. One direction is to continue the development of the JModelica compiler into a more complete modeling, simulation, and optimization environment. It then remains to develop support for the constructs which are currently unsupported by the JModelica compiler. Most notably, these constructs include improved support for parametrized classes, dynamic name lookup, functions and algorithms, and full support for arrays. The JModelica compiler currently lacks the algorithms needed to transform a flat Modelica representation into efficient simulation code. These algorithms include equation sorting, index reduction and tearing. In addition, in order to accommodate simulation of Modelica models, a simulation back-end has to be developed.

There are two main benefits resulting from continued development of the JModelica compiler. Firstly, since the compiler is generated from a JastAdd description, it consists of pure Java code. Such a compiler can be easily integrated and adapted in a wide range of tools and environment, independently of platform. Secondly, the JModelica compiler is modularly extensible, which makes it particularly interesting for experimental compiler development and for development of new language extensions of Modelica.

In Chapter 7, the Optimica extension was presented. While the current version of Optimica can be used to express a wide range of dynamic and static optimization problems, there are several potential extensions. Firstly, the syntax and semantics of Optimica may be extended to accommodate more general optimization problems. For example, a natural extension would be to support multi-stage problems. Another interesting extension would be to allow for `optimization` classes to be instantiated. This, in turn, would enable solution of optimization problems during simulation. For example, it would then be straightforward to implement model predictive controllers. Secondly, the Optimica compiler could be extended to support additional back-ends. One of the main arguments for development of the JModelica and Optimica compilers is to enable the user to

choose the most appropriate algorithm for a particular problem. Accordingly, code generation to different algorithms should then be supported. Two natural candidates for algorithms to implement support for would be a multiple shooting algorithm and a global search method. The latter would be useful in order to enable non-gradient based optimization of systems where gradients are difficult or impossible to obtain.

In order to assist the user in formulating the transcription scheme, it would be useful to develop a library containing some of the most common schemes. In the context of direct simultaneous methods, these include, for example, collocation methods based on Lagrange polynomials as well as monomial basis formulations of different orders.

12.2 Case studies

A Modelica library, DryLib, for modeling of paper machine dryer sections has been developed. The library is based on the physical model presented in [Slätteke, 2006], and contains base level components that encode the physical behavior, as well as components for structured hierarchical modeling of dryer sections. DryLib has been used to formulate three different design problems, namely model reduction, parameter optimization, and an NMPC scheme for paper moisture. These problems were solved by integrating different software packages, and in one case, the model reduction problem, complete re-encoding of the model was required. The large and problem-specific coding efforts needed to solve these problems also serve as a motivation for the development of high-level support for dynamic optimization.

In the second case study, dealing with start-up optimization of a plate reactor, the Optimica compiler was used to formulate and solve the resulting dynamic optimization problems. In this case, much of the tedious and error-prone coding work was eliminated, with increased focus on formulation of the optimization problem as a result. It was also shown how the cost function and constraints can be designed to achieve improved robustness to parameter variations. The improved robustness was demonstrated by means of Monte-Carlo simulations.

Future Work

DryLib currently contains classes that capture the behavior of the actual drying process. In order to model a complete paper machine, however, additional elements need to be added to the library. In particular, the wet end of a paper machine, including wire and press sections, and the steam and condensate systems need to be included. An extension of DryLib to

include components related to the condensate system was made in [Windahl, 2006].

Regarding the design problems that were formulated based on DryLib, it would be interesting to re-formulate the problems in Optimica. This is, however, not currently possible, since the JModelica compiler does not yet support dynamic name lookup (inner/outer constructs) which is extensively used in DryLib.

In the formulation of the start-up optimization problem for the plate reactor, the parameter uncertainties were taken into account implicitly. In particular, the relation between the properties of the cost function and the constraints on one hand, and the robustness properties of the resulting start-up trajectories was not quantified directly. Instead, it was shown by means of Monte-Carlo simulations that a particular configuration of cost function and constraints fulfilled the robustness requirements. An interesting extension of this work would be to take the parameter uncertainties into account explicitly in the optimization formulation. This, in turn, would lead to very large optimization problems, which would most likely call for extensive parallelization of the computations in order to be feasible. In this context, the techniques developed in [Zavala *et al.*, 2007] would be an interesting alternative.

12.3 Control of Pendula

In Chapter 10, an explicit characterization of the reachability set of a planar inverted pendulum was presented. Based on this set, a controller that guarantees semi-global stability when subject to manual control was developed. The original analysis was performed for a normalized pendulum system with acceleration of the pivot as input. In addition, an equivalent controller for the untransformed pendulum system, with the force acting on the pivot as input, was presented.

In Chapter 11 a pendulum on two wheels robot, YAIP, was presented. One of the main objectives in the construction of the robot was to show how low cost components can be used to construct high-precision control systems. In this respect, the inverted pendulum robot is a challenging subject of study, since common artifacts of mechanical actuators, such as friction and back-lash severely degrade control performance. A large part of the project was also devoted to accurate estimation of the wheel angles. For this purpose, analog encoder measurements were used. Also, since the computing resources of the on-board CPUs are scarce, the signal processing and control algorithms were implemented using fixed point arithmetic. The design presented in this thesis is the result of several prototypes, where different mechanical, electrical and software designs

were evaluated.

Future Work

In the case of manual controllers based on reachability sets, an interesting extension would be to develop such a controller for the Furuta pendulum. The dynamics of the Furuta pendulum, however, is significantly more challenging than that of a planar pendulum, and there seems to be no immediately available analytical characterization of the reachability set. Instead, a feasible alternative may be to use numerical level-set methods, for which there is efficient software available, see [Mitchell, 2007].

The inverted pendulum on two wheels robot, YAIP, would benefit from improved support for user interaction. Work in this direction has been performed as a student project in the course "Projects in Automatic Control", where the robot was set up to receive commands from a wireless joystick. Another interesting extension would be to implement a more sophisticated manual control scheme, that offers guaranteed stability regardless of reference commands from the user. For example, the results presented in Chapter 10 would be applicable in this context.

13

Bibliography

Abadi, M. and L. Cardelli (1996): *A Theory of Objects*. Springer Verlag.

Åkesson, J. (2003): "Operator interaction and optimization in control systems." Licentiate Thesis ISRN LUTFD2/TFRT--3234--SE. Department of Automatic Control, Lund University, Sweden.

Åkesson, J. and K. J. Åström (2001): "Safe manual control of the Furuta pendulum." In *Proceedings 2001 IEEE International Conference on Control Applications (CCA'01)*, pp. 890–895. Mexico City, Mexico.

Åkesson, J. and K. J. Åström (2005): "Manual control and stabilization of an inverted pendulum." In *Proc. 16th IFAC World Congress*. Prague, Czech Republic.

Åkesson, J., T. Ekman, and G. Hedin (2007): "Development of a Modelica compiler using JastAdd." In *Seventh Workshop on Language Descriptions, Tools and Applications*. Braga, Portugal.

Alfa Laval AB (2006): "Alfa Laval Reactor Technology." `http://www.alfalaval.com`.

Anderson, B. and J. Moore (1979): *Optimal Filtering*. Prentice-Hall, New Jersy.

Andersson, M. (1994): *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD thesis ISRN LUTFD2/TFRT--1043--SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Appel, A. (2002): *Modern compiler implementation in Java — Second edition*. Cambridge University Press.

Ascher, U. M. and L. R. Petzold (1998): *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics.

ASTOS Solutions GmbH (2006): "Optimization, Guidance and Control." http://www.astos.de.

Åström, K. J. (2007): In conversation.

Åström, K. J., H. Elmqvist, and S. E. Mattsson (1998): "Evolution of continuous-time modeling and simulation." In *Proceedings of the 12th European Simulation Multiconference, ESM'98*, pp. 9–18. Society for Computer Simulation International, Manchester, UK.

Åström, K. J. and K. Furuta (2000): "Swinging up a pendulum by energy control." *Automatica*, **36**, February, pp. 278–285.

Åström, K. J. and T. Hägglund (2005): *Advanced PID Control*. ISA - The Instrumentation, Systems, and Automation Society, Research Triangle Park, NC 27709.

Åström, K. J. and B. Wittenmark (1997): *Computer-Controlled Systems*. Prentice Hall.

Bader, G. and U. M. Ascher (1987): "A new basis implementation for mixed order boundary value ode solver." *SIAM J. Sci. Comput.*, pp. 483–500.

Baerveldt, A. J. and R. Klang (1997): "A low-cost and low-weight attitude estimation system for an autonomous helicopter." *Intelligent Engineering Systems, 1997. INES '97. Proceedings., 1997 IEEE International Conference on*, pp. 391–395.

Baron, S., D. Kleinman, and W. Levinson (1970): "An optimal control model of human response - part ii: Prediction of human performance in a complex task." *Automatica*, **6**, pp. 371–383.

Barton, P. and C. K. Lee (2002): "Modeling, simulation, sensitivity analysis, and optimization of hybrid systems." *ACM Transactions on Modeling and Computer Simulation*, **12:4**.

Bauman, E., A. Varma, J. Lorusso, M. Dente, and M. Morbidelli (1990): "Parametric sensitivity in tubular reactors with co-current external cooling." *Chemical Engineering Science*, **45**, pp. 1301–1307.

Beaver Project (2007): "Beaver - a LALR Parser Generator." `http://beaver.sourceforge.net/`.

Bellman, R. (1957): *Dynamic Programming*. Princeton University Press, Princeton, N.J.

Bertsekas, D. P. (2000a): *Dynamic Programming and Optimal Control, vol 1*. Athena Scientific.

Bertsekas, D. P. (2000b): *Dynamic Programming and Optimal Control, vol 2*. Athena Scientific.

Betts, J. T. (2001): *Practical Methods for Optimal Control Using Nonlinear Programming*. Society for Industrial and Applied Mathematics.

Biegler, L., A. Cervantes, and A. Wächter (2002): "Advances in simultaneous strategies for dynamic optimization." *Chemical Engineering Science*, **57**, pp. 575–593.

Blanchini, F. (1999): "Set invariance in control." *Automatica*, **35:11**, pp. 1747–1767.

Boeing (2007): "Sparse Optimal Control Software (SOCS)." `http://www.boeing.com/phantom/socs/`.

Bohlin, T. and A. J. Isaksson (2003): "Grey-box model calibrator and validator." In *13th IFAC Symposium on System Identification*. Rotterdam, The Netherlands.

Brenan, K., S. Campbell, and L. Petzold (1996): *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics.

Broman, D. and P. Fritzson (2007): "Abstract syntax can make the definition of Modelica less abstract." In *1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, Linköping Electronic Conference Proceedings. Linköping University Electronic Press, Linköping, Sweden.

Broman, D., P. Fritzson, and S. Furic (2006): "Types in the Modelica language." In *Proceedings of the 5th International Modelica Conference*.

Brufani, S. (1997): "Manual control of unstable systems." Master's Thesis ISRN LUTFD2/TFRT--5576--SE. Department of Automatic Control, Lund University, Sweden.

Bryson, A. E. and Y.-C. Ho (1975): *Applied optimal control*. Hemisphere Publishing Corporation.

Bush, V. (1931): "The differential analyzer: A new machine for solving differential equations." *Journal of the Franklin Institute*.

Cao, Y., S. Li, L. R. Petzold, and R. Serban (2003): "Adjoint sensitivity analysis for differential-algebraic equations: The adjoint dae system and its numerical solution." *SIAM J. Scientific Computing*, pp. 1076–1089.

Cellier, F. (1991): *Continuous System Modeling*. Springer-Verlag, New York, USA.

Cuthrell, J. E. (1986): *On the Optimization of Differential-Algebraic Systems of Equations in Chemical Engineering*. PhD thesis, Carnegie Mellon University.

Danielsson, H. (2007): "Vehicle path optimisation." Master's Thesis ISRN LUTFD2/TFRT--5797--SE. Department of Automatic Control, Lund University, Sweden.

Diehl, M., H. Bock, and E. Kostina (2006): "An approximation technique for robust nonlinear optimization." *Mathematical Programming*, **107**, pp. 213–230.

Diehl, M., D. Leineweber, and A. Schäfer (2001): *MUSCOD-II Users' Manual*. Interdisciplinary Center for Scientific Computing (IWR), University of Heidelberg, Germany.

Dynasim AB (2007): "Dynasim AB Home Page." `http://www.dynasim.se`.

Ekman, T. (2006): *Extensible Compiler Construction*. PhD thesis, Lund University, Sweden.

Ekman, T. and G. Hedin (2004): "Rewritable Reference Attributed Grammars." In *Proceedings of ECOOP 2004*, vol. 3086 of *LNCS*, pp. 144–169. Springer-Verlag.

Ekman, T. and G. Hedin (2006): "Modular name analysis for Java using JastAdd." In *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005*, vol. 4143 of *LNCS*. Springer-Verlag.

Ekman, T. and G. Hedin (2007): "The jastadd extensible java compiler." In *Proceedings of OOPSLA 2007*.

Ekman, T., G. Hedin, and E. Magnusson (2006): "JastAdd." http://jastadd.cs.lth.se/web/.

Ekvall, J. (2004): "Dryer section control in paper machines during web breaks." Licentiate Thesis ISRN LUTFD2/TFRT--3236--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Elmqvist, H. (1975): "Simnon — an interactive simulation program for nonlinaear systems — user's manual." Technical Report TFRT-7502. Department of Automatic Control, Lund University, Sweden.

Elmqvist, H. (1978): *A Structured Model Language for Large Continuous Systems*. PhD thesis TFRT-1015, Department of Automatic Control, Lund University, Sweden.

Elmqvist, H., S. E. Mattsson, H. Olsson, J. Andreasson, M. Otter, C. Schweiger, and D. Brück (2004): "Real-time simulation of detailed vehicle and powertrain dynamics." In *Electronics Simulation and Optimization (SAE 2004 World Congress)*. SAE International, Detroit.

Elmqvist, H., S. E. Mattsson, and M. Otter (1998): "Modelica—An international effort to design an object-oriented modeling language." In *Proceedings of the 1998 Summer Simulation Conference*, pp. 333–339. Society for Computer Simulation International, Reno, Nevada.

FAC-system (2006): "FAC Home Page." `http://www.facsystem.se/index.asp?lang=eng`.

Farrow, R. (1982): "Linguist-86: Yet another translator writing system based on attribute grammars." In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pp. 160–171. ACM Press, New York, NY, USA.

Farrow, R. (1986): "Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars." In *Proceedings of the SIGPLAN symposium on Compiler contruction*, pp. 85–98. ACM Press.

Fourer, R., D. Gay, and B. Kernighan (2003): *AMPL – A Modeling Language for Mathematical Programming*. Brooks/Cole — Thomson Learning.

Franke, R. (2007): "HQP: a solver for sparse nonlinear optimization." `http://sourceforge.net/projects/hqp/`.

Franke, R., M. Rode, and K. Krüger (2003): "On-line optimization of drum boiler startup." In *Proceedings of Modelica'2003 conference*.

Fritzson, P. (2004): *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. John Wiley & Sons.

Froment, G. and K. Bischoff (1990): *Chemical reactor analysis and design*. Wiley.

Gabor, A. (1974): "Apparatus measuring relative velocity of movable members including means to detect velocity from the position encoder." Patent. Pat. no. US3839665.

Gaines, B. R. (1969): "Linear and nonlinear models of the human controller." *International Journal of Man-Machine Studies*, **1**, pp. 330–360.

Gamma, E. H., R. Johnson, and J. R. Vlissides (1995): *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley.

Gerdin, M., T. Schön, T. Glad, F. Gustafsson, and L. Ljung (2007): "On parameter and state estimation for linear differential-algebraic equations." *Automatica*, **43:3**, pp. 416–425.

Gerwin Klein (2007): "JFlex - The Fast Scanner Generator for Java." `http://jflex.de/`.

Grasser, F., A. D'Arrigo, and S. Colombi (2002): "Joe: A mobile, inverted pendulum." *IEEE Transactions on Industrial Electronics*, **49:1**, pp. 107–115.

Grewal, M., V. Henderson, and R. Miyasako (1991): "Application of kalman filtering to the calibration and alignment of inertial navigation systems." *Automatic Control, IEEE Transactions on*, **36:1**, pp. 3–13.

Gustafsson, K. (1994): "Traps and pitfalls in simulation." In *SIMS (Scandinavian Simulation Society) Simulation Conference*.

Hägglund, T. (1991): *Process Control in Practice*. Chartwell-Bratt Ltd, Bromley, UK.

Hahn, D., L. Fan, and C. Hwang (1971): "Optimal startup control of a jacketed tubular reactor." *AIChE Journal*, **17**, pp. 1394–1401.

Hall, I. A. M. (1963): "Study of the human pilot as a servo element." *J. Royal Aeronatutic Soc.*, **67**.

Hangos, K. and I. Cameron (2001): *Process Modelling and Model Analysis*. Academic Press.

Haugwitz, S. and P. Hagander (2006): "Challenges in start-up control of a heat exchange reactor with exothermic reactions; a hybrid Approach." In *Proceedings of the 2nd IFAC Conference on Analysis and Design of Hybrid Systems*. Alghero, Italy.

Haugwitz, S., P. Hagander, and T. Norén (2007): "Modeling and control of a novel heat exchange reactor, the open plate reactor." *Control Engineering Practice*, **15:7**, pp. 779–792.

Hedin, G. (2000): "Reference Attributed Grammars." In *Informatica (Slovenia)*, 24(3), pp. 301–317.

Hedin, G. and E. Magnusson (2003): "JastAdd: an aspect-oriented compiler construction system." *Science of Computer Programming*, **47:1**, pp. 37–58.

Heikkilä, P. (1993): *A study on the drying process of pigment coated paper webs*. PhD thesis, Department of Chemical Engineering, Åbo Akademi, Åbo, Finland.

Henriksson, D., A. Cervin, and K.-E. Årzén (2003): "TrueTime: Real-time control system simulation with MATLAB/Simulink." In *Proceedings of the Nordic MATLAB Conference*. Copenhagen, Denmark.

Hovland, P. and A. Carle (2007): "ADIFOR 2.0 automatic differentiation of fortran." `http://www-unix.mcs.anl.gov/autodiff/ADIFOR/`.

Hu, T., Z. Lin, and L. Qiu (2001): "Stabilization of exponentially unstable linear systems with saturating actuators." *IEEE Transactions on Automatic Control*, **46:6**, pp. 973–979.

Hultgren, H. and H. Jonasson (2007): "Automatic calibration of vehicle models." Master's Thesis ISRN LUTFD2/TFRT--5794--SE. Department of Automatic Control, Lund University, Sweden.

IEEE (1997): "Standard vhdl analog and mixed-singnal extensions." Technical Report. IEEE.

Ingalls, D. H. H. (1986): "A simple technique for handling multiple polymorphism." In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pp. 347–349. ACM Press, New York, NY, USA.

INRIA (2007): "SciLab Home Page." `http://www.scilab.org/`.

ITI GmbH (2007): "ITI GmbH Home Page." `http://www.iti.de/`.

Johansson, K. H. (1997): *Relay Feedback and Multivariable Control*. PhD thesis ISRN LUTFD2/TFRT--1048--SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Kågedal, D. and P. Fritzson (1998): "Generating a Modelica compiler from natural semantics specifications." In *Proceedings of the Summer Computer Simulation Conference*.

Kameswaran, S. and L. Biegler (2006): "Convergence rates for direct transcription of optimal control problems with final-time equality constraints using collocation at radau points." In *American Control Conference, 2006*.

Karlsson, M., Ed. (2000): *Paper Making part 2, drying*. Tappi Press.

Karlsson, M. (2005): *Static and Dynamic Modelling of the Drying Section of a Paper Machine*. PhD thesis, Department of Chemical Engineering, Lund Institute of Technology, Lund, Sweden.

Karnopp, D. and R. Rosenberg (1968): *Analysis and simulation of multiport systems — The bond graph approach to physical system dynamics*. MIT Press, Cambridge, MA, USA.

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold (2001): "An overview of AspectJ." *LNCS*, **2072**, pp. 327–355.

Kleinman, D., S. Baron, and W. Levinson (1970): "An optimal control model of human response - part i: Theory and validation." *Automatica*, **6**, pp. 357–369.

Knuth, D. E. (1968): "Semantics of context-free languages." *Mathematical Systems Theory*, **2:2**, pp. 127–145. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

Laabissi, M., M. Achhab, J. Winkin, and D. Dochain (2002): "Equilibrium profiles of tubular reactor nonlinear models." *In Proceedings of 15th Int. Symposium on Mathematical Theory of Networks and Systems*.

Levenspiel, O. (1999): *Chemical Reaction Engineering*. Wiley.

Li, S. and L. Petzold (2002): "Description of daspkadjoint: An adjoint sensitivity solver for differential-algebraic equations." Technical Report. Department of Computer Science, University of California Santa Barbara, USA.

Lincoln, B. (2003): *Dynamic Programming and Time-Varying Delay Systems*. PhD thesis ISRN LUTFD2/TFRT--1067--SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Maciejowski, J. M. (2002): *Predictive Control with Constraints*. Pearson Education.

Magnusson, E. and G. Hedin (2003): "Circular Reference Attributed Grammars - Their Evaluation and Applications." *Electr. Notes Theor. Comput. Sci.*, **82:3**.

Maly, T. and L. R. Petzold (1996): "Numerical methods and software for sensitivity analysis of differential-algebraic systems." *Applied Numerical Mathematics*, **20:1-2**, pp. 57–82.

MathCore Engineering AB (2007): "MathCore Engineering AB Home Page." `http://www.mathcore.com/`.

Mayne, D. Q., J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert (2000): "Constrained model predictive control: Stability and optimality." *Automatica*, **36:6**, pp. 789–814.

McConnell, R. (1980): "A literature review of drying research in the pulp and paper industry." In Mujumdar, Ed., *Drying '80*. Hemisphere Publishing, NY.

McRuer, D., D. Graham, E. S. Krendel, and W. Reisener (1965): "Human pilot dynamics in compensatory systems." Report. AFFDL-TR-65-15.

McRuer, D. and E. S. Krendel (1959): "The human operator as a servo element." *J. Franklin Institute*, **267**, pp. 381–403, 511–536.

Miall, R., D. Weir, D. Wolpert, and J. Stein (1993): "Is the cerebellum a smith predictor." *Journal of Motor Behavior*, **25**, pp. 203–216.

Mitchell, E. E. L. and J. S. Gauthier (1976): "Advanced continuous simulation language (ACSL)." *Simulation*, **26:3**.

Mitchell, I. M. (2007): *A Toolbox of Level Set Methods (Version 1.1)*. Department of Computer Science, University of British Columbia, Canada.

Mitchell, I. M. and J. Templeton (2005): "A toolbox of Hamilton-Jacobi solvers for analysis of nondeterministic continuous and hybrid systems." *Lecture Notes in Computer Science*, **3414**, pp. 480–494.

Moore, B. (1981): "Principal component analysis in linear systems: controllability, observability, and model reduction." *IEEE Transactions on Automatic Control*, **26:1**, pp. 17–32.

Moray, N., W. B. Ferrell, and W. B. Rouse (1990): *Robotics Control and Society*. Taylor & Francis, London.

MSC Software (2007): "ADAMS Home Page." `http://www.mscsoftware.com/products/adams.cfm`.

Nagel, L. and D. O. Pederson (1973): "Simulation program with integrated circuit emphasis (spice)." Technical Report. Electronics Research Laboratory, College of Engineering, University of California Berkeley, CA, USA. Memorandum ERL-M382.

Nilsson, B. (1993): *Object-Oriented Modeling of Chemical Processes*. PhD thesis ISRN LUTFD2/TFRT--1041--SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Nissan, A. and W. Kaye (1955): "An analytical approach to the problem of drying thin fibrous sheets on multicylinder machines." *Tappi Journal*, **38:7**, pp. 385–398.

Numerica Technolgy (2007): "Jacobian." `http://www.numericatech.com/jacobian.htm`.

Nytsch-Geusen, C. (2007): "MosiLab Home Page." `http://www.mosilab.de/`.

Object Management Group (2007): "Unified Modeling Language." `http://www.uml.org/`.

Olsson, H., K. J. Åström, C. C. de Wit, M. Gäfvert, and P. Lischinsky (1998): "Friction models and friction compensation." *European Journal of Control*, January.

Patcher, M. and R. Miller (1998): "Manual flight control with saturating actuators." *IEEE Control Systems*, February, pp. pp. 10–19.

PELAB (2007): "The OpenModelica Project." `http://www.ida.liu.se/~pelab/modelica/OpenModelica.html`.

Pettersson, J., U. Persson, T. Lindberg, L. Ledung, and Z. Xiaojing (2005): "On-line pulp mill production optimization." In *16th IFAC World congress*.

Pettersson, M. and S. Stenström (2000): "Experimental evaluation of electric infrared dryers." *Tappi Journal*, **83:8**.

Petzold, L. (1986): "Order results for implicit runge-kutta methods applied to defferential/algebraic systems." *SIAM Journal on Numerical Analysis*, **23:4**, pp. 837–852.

Pontryagin, L. S., V. G. Boltyanskii, R. V. Gamkrelidze, and E. F. Mishchenko (1962): *The Mathematical Theory of Optimal Processes*. John Wiley & Sons Inc.

Pop, A. and P. Fritzson (2006): "MetaModelica: A unified equation-based semantical and mathematical modeling language." In *Joint Modular Languages Conference 2006 (JMLC2006)*. Oxford, England.

Prat, L., A. Devatine, P. Cognet, M. Cabassud, C. Gourdon, S. Elgue, and F. Chopard (2005): "Performance evaluation of a novel concept "open plate reactor" applied to highly exothermic reactions." *Chemical Engineering Technology*, **28**, pp. 1028–1034.

Process Systems Enterprise (2007): "gPROMS Home Page." `http://www.psenterprise.com/gproms/index.html`.

Pulkowski, J. H. and G. L. Wedel (1988): "The effect of spoiler bars on dryer heat transfer." *Pulp and Paper Magazine of Canada*, **89:8**, pp. 61–66.

Qin, S. J. and T. A. Badgwell (2003): "A survey of industrial model predictive control technology." *Control Engineering Practice*, **11**, pp. 733–764.

Rao, C. V., J. B. Rawlings, and D. Q. Mayne (2003): "Constrained state estimation for nonlinear discrete-time systems: Stability and moving horizon approximations." *IEEE Transactions on Automatic Control*, **48:2**, pp. 246–259.

Rao, M., Q. Xia, and Y. Ying (1994): *Modeling and Advanced Control for Process Industries – Applications to Paper Making Processes*. Springer-Verlag, New York.

Rausmussen, J. (1983): "Skill, rules, and knowledge; signals, signs, and symbols, and other distinctions in human performance models." *IEEE Trans. on Systems, Man, and Cybernetics*, **3**, pp. 257–266.

Rosen, O. and R. Luus (1991): "Evaluation of gradients for piecewise constant optimal control." *Comput. chem. Engng.*, **15:4**, pp. 273–281.

Rundqwist, L., K. Stål-Gunnarsson, and J. Enhagen (1997): "Rate limiters with phase compensation in JAS 39 Gripen." In *Proc. European Control Conference*. Saab Military Aircraft, Linköping, Sweden.

Rönnbäck, S. (1993): *Linear Control of Systems with Actuator Constraint*. PhD thesis ISRN LUTFD2/TFRT--0348–8373--SE, Luleå University of Technology.

Schiesser, W. (1991): *The Numerical Method of Lines: Integration of Partial Differential Equations*. Academic Press.

Schmidt, E. (1969): *Properties of water and steam in SI-units*. Springer Verlag, Berlin, Germany.

Scokaert, P. O. M., D. Q. Mayne, and J. B. Rawlings (1999): "Suboptimal model predictive control (feasibility implies stability)." *IEEE Transactions of Automatic Control*, **44:3**, pp. 648–654.

Segway Inc. (2006): "Segway Home Page." `http://www.segway.com/`.

Sheridan, T. B. and T. V. Lunteren (1997): *Perspectives on the Human Controller - Essays in Honor of Henk G. Stassen*. Lawrence Erlbaum Associates, Mahwah, NJ.

Slätteke, O. (2006): *Modeling and Control of the Paper Machine Drying Section*. PhD thesis ISRN LUTFD2/TFRT--1075--SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Slätteke, O. and K. J. Åström (2005): "Modeling of a steam heated rotating cylinder—A grey-box approach." In *Proc. 2005 American Control Conference*. Portland, Oregon, USA.

Stein, G. (2003): "Respect the unstable." *IEEE Control Systems Magazine*, **23:4**. The first IEEE Bode lecture 1990.

Stengel, R. F. (1994): *Optimal Control and Estimation*. Dover Publications Inc.

Strauss, J. (1967): "The sci continuous system simulation language (CSSL)." *Simulation*, **9:6**.

Sussmann, H., E. Sontag, and Y. Yang (1994): "A general result on the stabilization of linear systems using bounded controls." *IEEE Transactions on Automatic Control*, **39:12**, pp. 2411–2425.

Tarjan, R. (1972): "Depth-first search and linear graph algorithms." *SIAM J. Computing*, **1:2**, pp. 146–160.

Teel, A. (1992): "Global stabilization and restricted tracking for multiple integrators with bounded controls." *System & Control Letters*, **18**, pp. 165–171.

Teel, A. (1996): "A nonlinear small gain theorem for the analysis of control systems with saturation." *IEEE Trans. on Automatic Control*, **41:9**, pp. 1256–1270.

Teel, A. (1999): "Anti-windup for exponentially unstable linear systems." *International Journal of Robust and Nonlinear Control*, **9:10**, pp. 701–716.

Teel, A. and N. Kapoor (1997): "The l_2 anti-windup problem: its definition and solution." In *Proceedings of European Control Conference*.

The Modelica Association (1997): "Modelica – a unified object-oriented language for physical systems modeling, language specification, version 1." Technical Report. Modelica Association.

The Modelica Association (2005): "Modelica – a unified object-oriented language for physical systems modeling, language specification, version 2.2." Technical Report. Modelica Association.

The Modelica Association (2007a): "Modelica – a unified object-oriented language for physical systems modeling, language specification, version 3.0." Technical Report. Modelica Association.

The Modelica Association (2007b): "The Modelica Association Home Page." `http://www.modelica.org`.

The Omuses Team (2007): "Omuses: a tool for the Optimization of Multistage Systems." `http://swik.net/Omuses`.

Thomas, P. (1999): *Simulation of industrial processes - for control engineers*. Butterworth-Heinemann, Oxford, Great Britain.

Tustin, A. (1947): "The nature of the human operators response in manual control and its implication for controller design." *Journal IEE*, **94**, pp. 190–.

Varma, A., M. Morbidelli, and H. Wu (1999): *Parametric sensitivity in chemical systems*. Cambridge University Press.

Vassiliadis, V. (1993): *Computational solution of dynamic optimization problem with general differnetial-algebraic constraints*. PhD thesis, Imerial Collage, London, UK.

Venema, S. (1994): "A kalman filter calibration method for analog quadrature position encoders.". Master's thesis, University of Washington.

Verwijs, J., H. van den Berg, and K. Westerterp (1996): "Startup strategy design and safeguarding of industrial adiabatic tubular reactor systems." *AIChE Journal*, **42**, pp. 503–515.

Vogt, H. H., S. D. Swierstra, and M. F. Kuiper (1989): "Higher order attribute grammars." In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, pp. 131–145. ACM Press.

Wächter, A. and L. T. Biegler (2006): "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming." *Mathematical Programming*, **106:1**, pp. 25–58.

Walther, A. and A. Griewank (2007): "ADOL-C a package for automatic differentiation of algorithms written in C/C++." `http://www.math.tu-dresden.de/~adol-c/`.

Waltz, R. (2005): *KNITRO User's Manual, Version 4.0*. Zienna Optimization Inc.

Wilde, R. W. and J. Westcott (1962): "The characteristics of the human operator engaged in a tracking task." *Automatica*, **1**.

Wilhelmsson, B. (1995): *An experimental and theoretical study of multicylinder paper drying*. PhD thesis, Department of Chemical Engineering, Lund Institute of Technology, Lund, Sweden.

Windahl, J. (2006): "Modelling and parameter estimation of a paper machine drying section using Modelica." Master's Thesis ISRN LUTFD2/TFRT--5783--SE. Department of Automatic Control, Lund University, Sweden.

Zaldívar, J., J. Cano, M. Alós, J. Sempere, R. Nomen, D. Lister, G. Maschio, T. Obertopp, E. Gilles, J. Bosch, and F. Strozzi (2003): "A general criterion to define runaway limits in chemical reactors." *Journal of Loss Prevention in the Process Industries*, **16**, pp. 187–200.

Zavala, V., C. Laird, and L. Biegler (2007): "Interior-point decomposition approaches for parallel solution of large-scale nonlinear parameter estimation problems." *Chem. Eng. Sci.* In Press.

Zimmerman, Y., Y. Oshman, and A. Brandes (2006): "Improving the accuracy of analog encoders via kalman filtering." *Control Engineering Practice*, **14:4**, pp. 337–350.

A

Collocation and Runge-Kutta Methods

There are strong connections between collocation methods and implicit Runge-Kutta methods for solving differential equations. This relation has been noted previously, for example in [Cuthrell, 1986], where a special case is considered. In this Appendix, it is shown, explicitly, how the Butcher tableau of a Runge-Kutta scheme can be constructed given a collocation method.

Consider the ordinary differential equation

$$\frac{dy(t)}{dt} = f(t, y(t)), \quad y(0) = y_0 \tag{A.1}$$

The famous Runge-Kutta scheme is given by

$$y_{n+1} = y_n + h \sum_{i=1}^{N} b_i k_i \tag{A.2}$$

where $y_n = y(t_n)$ denotes the approximate solution of (A.1) at time t_n, h is the step length and the coefficients k_i are given by

$$\begin{aligned} k_1 &= f(t_n + c_1 h, y_n + a_{11} h k_1 + a_{12} h k_2 + \ldots a_{1N} h k_N) \\ k_2 &= f(t_n + c_2 h, y_n + a_{21} h k_1 + a_{22} h k_2 + \ldots a_{2N} h k_N) \\ &\vdots \\ k_N &= f(t_n + c_N h, y_n + a_{N1} h k_1 + a_{N2} h k_2 + \ldots a_{NN} h k_N) \end{aligned} \tag{A.3}$$

Solving the equations (A.3) for the $k_i : s$ gives the approximate solution y_{n+1} using (A.2). Commonly, a Runge-Kutta method is characterized the corresponding Butcher tableau, given by

c	A
	b^T

A collocation scheme for the differential equation (A.1) is characterized by a set of collocation points, $\tau_i, \quad i = 1 \ldots N$, $0 \leq \tau_i \leq 1$. To simplify the notation in the following, the step length is normalized to unity, which gives $t_i = t_0 + h\tau_i$, where t_0 denotes the start time of the step. Consequently, (A.1) is transformed into

$$\frac{dy}{d\tau} = \frac{dy}{dt}\frac{dt}{d\tau} = hf(t,y). \tag{A.4}$$

Assume that the solution of (A.4) is approximated, in the normalized interval $[0,1]$, by the polynomials

$$y^{N+1}(\tau) = \sum_{j=0}^{N} \alpha_j \tau^j, \tag{A.5}$$

where α_j is a vector of appropriate size. The collocation equations may then be written

$$\frac{dy^{N+1}}{d\tau} = \sum_{j=1}^{N} j\alpha_j \tau_i^{j-1} = hf(t_i, y^{N+1}(\tau_i)) \quad i = 1 \ldots N. \tag{A.6}$$

Solving the equations (A.6) for the $\alpha_i : s$, with the additional requirement $\alpha_0 = y_0$ specifies the approximation polynomial (A.5).

We will now proceed to show how the collocation equations can rewritten to a Butcher tableau. Let us write the collocation equations (A.6) on matrix form

$$Q\bar{\alpha} = \begin{bmatrix} 1 & 2\tau_1 & \ldots & N\tau_1^{N-1} \\ \vdots & & & \vdots \\ 1 & 2\tau_N & \ldots & N\tau_N^{N-1} \end{bmatrix} \begin{bmatrix} \alpha_1^T \\ \vdots \\ \alpha_N^T \end{bmatrix} = h \begin{bmatrix} f(t_1, y^{N+1}(\tau_1)^T \\ \vdots \\ f(t_N, y^{N+1}(\tau_N)^T \end{bmatrix}. \tag{A.7}$$

This gives

$$\alpha_j = h\sum_{i=1}^{N} m_{ji} f(t_i, y^{N+1}(\tau_i)) = h\sum_{i=1}^{N} m_{ji} k_i \tag{A.8}$$

where m_{ji} are the elements of the matrix $M = Q^{-1}$ and the variables

$$k_i = f(t_i, y^{N+1}(\tau_i)) \tag{A.9}$$

have been introduced. We proceed by showing that the approximation polynomials y^{N+1} can be expressed as an affine combination of the $k_i : s$.

$$\begin{aligned} y^{N+1}(\tau_i) &= \alpha_0 + \sum_{j=1}^{N} \alpha_j \tau_i^j = \alpha_0 + \sum_{j=1}^{N} \left[h \sum_{l=1}^{N} m_{jl} k_l \right] \tau_i^j \\ &= \alpha_0 + h \sum_{l=1}^{N} \left[\sum_{j=1}^{N} m_{jl} \tau_i^j \right] k_l = y_0 + h \sum_{l=1}^{N} a_{il} k_l \end{aligned} \tag{A.10}$$

The coefficients of the A matrix in the Butcher tableau is thus given by

$$a_{il} = \sum_{j=1}^{N} m_{jl} \tau_i^j. \tag{A.11}$$

In a similar way, the coefficients of b in the Butcher tableau can be derived.

$$y^{N+1}(h) = \alpha_0 + \sum_{j=1}^{N} \alpha_j 1^j = \ldots = y_0 + h \sum_{i=0}^{N} b_i k_i \tag{A.12}$$

where

$$b_i = \sum_{j=1}^{N} m_{ji}. \tag{A.13}$$

The c vector is given by

$$c^T = \left[\begin{array}{c} \tau_1 \ldots \tau_N \end{array} \right]. \tag{A.14}$$

It can be remarked that the Butcher tableau matrices depend only on the location of the collocation points τ_i, which in turn fully specifies the method.

B

PicoModelica Syntax and Abstract Grammar

B.1 Concrete Syntax

In the defininition of the concrete syntax of PicoModelica, `[]` denotes an optional element, `{}` denotes zero, one, or more elements, and `|` denotes that a rule has two or more valid right-hand sides. Keywords are indicated by boldface.

```
model_definition :
  model IDENT
  {extends_clause}
  {model_definition}
  {component_decl}
  [equation_clause]
  end IDENT ";"

extends_clause :
  extends name [class_modification] ";"

component_decl :
  [replaceable] name IDENT [modification] ";"

modification :
  class_modification ["=" expression]
  | "=" expression

class_modification :
  "(" [argument_list] ")"

argument_list :
  argument {"," argument}
```

```
argument :
  redeclare [replaceable] name IDENT [modification]
  | name modification

equation_clause :
  equation
  {equation_}

equation_ :
  expression "=" expression ";"

expression :
  expression "+" expression
  | UNSIGNED_NUMBER
  | name

name :
  IDENT ["." name]
```

B.2 Source Abstract Grammar

```
Root ::= ClassDecl*;
abstract ClassDecl ::= Name:IdDecl;
Model : ClassDecl ::= Super:ExtendsClause*
                      ClassDecl*
                      ComponentDecl*
                      Equation* ParModel*
                      /InstRoot/;
RealClass : ClassDecl;

ExtendsClause ::= Super:Access
                  [Modification];
ComponentDecl ::= [Replaceable]
                  [Parameter]
                  ClassName:Access
                  Name:IdDecl
                  [Modification];
Replaceable;


abstract Modification;
CompositeModification : Modification ::= Modification*;
EquationModification : Modification ::= Exp;
abstract NamedModification : Modification ::= Name:Access;
ComponentRedeclare : NamedModification ::= ComponentDecl;
ComponentModification : NamedModification ::= Modification;
```

```
IdDecl ::= <ID:String>;

abstract Access : Exp ::= <ID:String>;
Dot : Access ::= Left:Access Right:Access;
ClassAccess : Access;
ComponentAccess : Access;
ParseAccess : Access;
AmbiguousAccess : Access;

BoundClassAccess : ClassAccess ::= <ClassDecl:ClassDecl>;
BoundComponentAccess : ComponentAccess ::= <ComponentDecl:ComponentDecl>;

Equation ::= Left:Exp Right:Exp;
abstract Exp;
BinExp : Exp ::= Left:Exp Right:Exp;
AddExp : BinExp;
RealLitExp : Exp ::= <Val:double>;
```

B.3 Instance Abstract Grammar

```
abstract InstNode ::= /InstNode*/
                      /MergedEnvironment:InstModification*/;
InstRoot : InstNode ::= <ClassDecl:ClassDecl>;
abstract InstComponent : InstNode ::= <ComponentDecl:ComponentDecl>;
InstComposite : InstComponent;
InstReal : InstComponent;
InstExtends : InstNode ::= <ExtendsClause:ExtendsClause>;

InstReplacingComponent : InstComponent ::= <OriginalDecl:ComponentDecl>;

InstModification ::= <InstNode:InstNode> <Modification:Modification>;
```

B.4 Flat Abstract Grammar

```
FClass ::= <Name:String> FVariable* FEquation*;
FVariable ::=  Name:FIdDecl [BindingExp:FExp];
FEquation ::= Left:FExp Right:FExp;

abstract FExp;
abstract FBinExp : FExp::= Left:FExp Right:FExp;
FAddExp : FBinExp::=;
FRealLitExp : FExp ::= <Val:double>;
FIdDecl ::=<ID>;
FIdUse : FExp ::= <ID>;
```

C

Complete AST for the Examples in Chapter 5

The complete AST for the model used in Examples 5.1, 5.2 and 5.4 is shown in Figure C.1. Notice that all nodes corresponding to identifiers are of type `ParseAccess`, i.e., the AST in Figure C.1 is the AST resulting from parsing, before the rewrites specified in the name classification framework are performed.

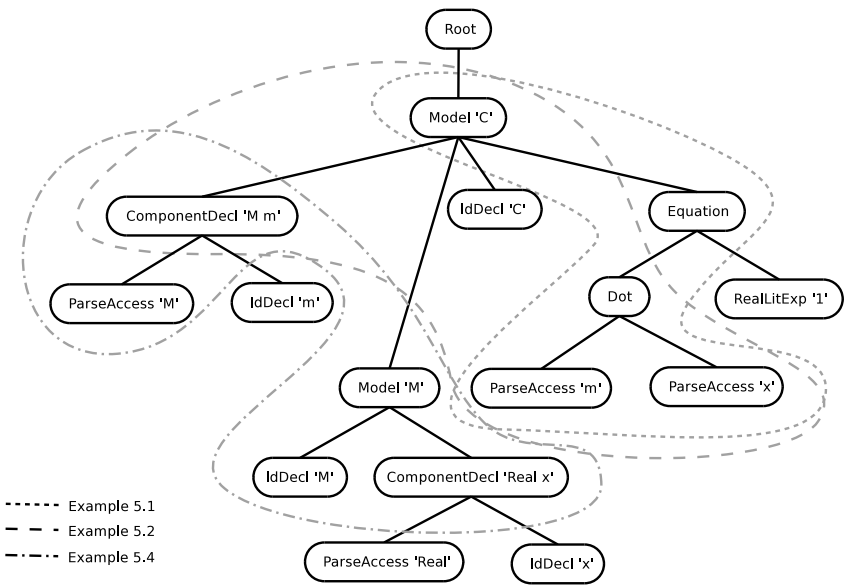


Figure C.1 Complete AST for the model used in the examples in Chapter 5.

D

A Flat Optimica Description for an Optimization Problem

In this appendix, the flat Optimica description corresponding to the parameter optimization problem composed of the Modelica model in Listing 7.3 and the Optimica description in Listing 7.4 is given. The flat Optimica description was generated by the Optimica compiler.

```
optimization OptimicaTests.ServoParameterOptimization
    (objective=cost,startTime=0,finalTime=100)
  parameter Real servo.slidingMass.m
    (quantity="Mass",final unit="kg",min=0)=servo.m1
    "mass of the sliding mass";
  Real servo.slidingMass.v(final quantity="Velocity",final unit="m/s")
    "absolute velocity of component";
  Real servo.slidingMass.a
    (final quantity="Acceleration",final unit="m/s2")
    "absolute acceleration of component";
  Real servo.slidingMass.s
    (start=s0,final quantity="Length",final unit="m")
    "absolute position of center of component
     (s = flange_a.s + L/2 = flange_b.s - L/2)";
  parameter Real servo.slidingMass.L
    (final quantity="Length",final unit="m")=0
    "length of component from left flange to right flange
     (= flange_b.s - flange_a.s)";
  Real servo.slidingMass.flange_a.s
    (final quantity="Length",final unit="m")
    "absolute position of flange";
  Real servo.slidingMass.flange_a.f
    (final quantity="Force",final unit="N")
    "cut force directed into flange";
  Real servo.slidingMass.flange_b.s
    (final quantity="Length",final unit="m")
```

```
    "absolute position of flange";
  Real servo.slidingMass.flange_b.f
    (final quantity="Force",final unit="N")
    "cut force directed into flange";
  Real servo.force.flange_b.s(final quantity="Length",final unit="m")
    "absolute position of flange";
  Real servo.force.flange_b.f(final quantity="Force",final unit="N")
    "cut force directed into flange";
  input Real servo.force.f "driving force as input signal";
  input Real servo.F;
  parameter Real servo.springDamper.s_rel0
    (final quantity="Length",final unit="m")=0
    "unstretched spring length";
  parameter Real servo.springDamper.c
    (final unit="N/m",final min=0)=servo.c "spring constant";
  parameter Real servo.springDamper.d
    (final unit="N/(m/s)",final min=0)=servo.d "damping constant";
  Real servo.springDamper.v_rel
    (final quantity="Velocity",final unit="m/s")
    "relative velocity between flange_a and flange_b";
  Real servo.springDamper.flange_a.s
    (final quantity="Length",final unit="m")
    "absolute position of flange";
  Real servo.springDamper.flange_a.f
    (final quantity="Force",final unit="N")
    "cut force directed into flange";
  Real servo.springDamper.flange_b.s
    (final quantity="Length",final unit="m")
    "absolute position of flange";
  Real servo.springDamper.flange_b.f
    (final quantity="Force",final unit="N")
    "cut force directed into flange";
  Real servo.springDamper.s_rel
    (min=-(2000),final quantity="Length",final unit="m")
    "relative distance (= flange_b.s - flange_a.s)";
  Real servo.springDamper.f
    (final quantity="Force",final unit="N")
    "forcee between flanges (positive in direction of flange axis R)";
  output Real servo.y;
  parameter Real servo.slidingMass1.m
    (quantity="Mass",final unit="kg",min=0)=servo.m2
    "mass of the sliding mass";
  Real servo.slidingMass1.v
    (final quantity="Velocity",final unit="m/s")
    "absolute velocity of component";
  Real servo.slidingMass1.a
    (final quantity="Acceleration",final unit="m/s2")
```

```
  "absolute acceleration of component";
Real servo.slidingMass1.s
  (final quantity="Length",final unit="m")
  "absolute position of center of component
   (s = flange_a.s + L/2 = flange_b.s - L/2)";
parameter Real servo.slidingMass1.L
  (final quantity="Length",final unit="m")=0
  "length of component from left flange to right flange
   (= flange_b.s - flange_a.s)";
Real servo.slidingMass1.flange_a.s
  (final quantity="Length",final unit="m")
  "absolute position of flange";
Real servo.slidingMass1.flange_a.f
  (final quantity="Force",final unit="N")
  "cut force directed into flange";
Real servo.slidingMass1.flange_b.s
  (final quantity="Length",final unit="m")
  "absolute position of flange";
Real servo.slidingMass1.flange_b.f
  (final quantity="Force",final unit="N")
  "cut force directed into flange";
parameter Real servo.m1(free=true,initialGuess=0.5)=1;
parameter Real servo.m2=1;
parameter Real servo.d=0.1;
parameter Real servo.c=0.01;
Real servo.speedSensor.flange_a.s
  (final quantity="Length",final unit="m")
  "absolute position of flange";
Real servo.speedSensor.flange_a.f
  (final quantity="Force",final unit="N")
  "cut force directed into flange";
Real servo.speedSensor.s
  (final quantity="Length",final unit="m")
  "Absolute position of flange";
output Real servo.speedSensor.v
  "Absolute velocity of flange as output signal";
parameter Real sine.amplitude=1 "Amplitude of sine wave";
parameter Real sine.freqHz
  (final quantity="Frequency",final unit="Hz")=0.1
  "Frequency of sine wave";
parameter Real sine.phase
  (final quantity="Angle",final unit="rad",displayUnit="deg")=0
  "Phase of sine wave";
parameter Real sine.offset=0 "Offset of output signal";
parameter Real sine.startTime
  (final quantity="Time",final unit="s")=0
  "Output = offset for time < startTime";
```

```
  constant Real sine.pi=sine.Modelica.Constants.pi;
  output Real sine.y "Connector of Real output signal";
  parameter Real s0(free=true,initialGuess=0.1);
  parameter Integer N=1000;
  parameter Real data_vals[N]={...};
  parameter Real data_times[N]={...};
  Real cost=sum((data_vals[i]-(servo.y(data_times[i])))^2 for i in 1:N);
equation
  servo.slidingMass.v=der(servo.slidingMass.s);
  servo.slidingMass.a=der(servo.slidingMass.v);
  (servo.slidingMass.m)*(servo.slidingMass.a)=
    servo.slidingMass.flange_a.f+servo.slidingMass.flange_b.f;
  servo.slidingMass.flange_a.s=
    servo.slidingMass.s-((servo.slidingMass.L)/(2));
  servo.slidingMass.flange_b.s=
    servo.slidingMass.s+(servo.slidingMass.L)/(2);
  servo.force.flange_b.f=-(servo.force.f);
  servo.springDamper.v_rel=
    der(servo.springDamper.s_rel);
  servo.springDamper.f=
    (servo.springDamper.c)*(servo.springDamper.s_rel-
    (servo.springDamper.s_rel0))+(servo.springDamper.d)*
    (servo.springDamper.v_rel);
  servo.springDamper.s_rel=
    servo.springDamper.flange_b.s-(servo.springDamper.flange_a.s);
  servo.springDamper.flange_b.f=servo.springDamper.f;
  servo.springDamper.flange_a.f=-(servo.springDamper.f);
  servo.slidingMass1.v=der(servo.slidingMass1.s);
  servo.slidingMass1.a=der(servo.slidingMass1.v);
  (servo.slidingMass1.m)*(servo.slidingMass1.a)=
    servo.slidingMass1.flange_a.f+servo.slidingMass1.flange_b.f;
  servo.slidingMass1.flange_a.s=
    servo.slidingMass1.s-((servo.slidingMass1.L)/(2));
  servo.slidingMass1.flange_b.s=
    servo.slidingMass1.s+(servo.slidingMass1.L)/(2);
  servo.speedSensor.s=servo.speedSensor.flange_a.s;
  servo.speedSensor.v=der(servo.speedSensor.s);
  0=servo.speedSensor.flange_a.f;
  sine.y=sine.offset+(if time<sine.startTime then 0
    else (sine.amplitude)*(sin((((2)*(sine.pi))*(sine.freqHz))*
    (time-(sine.startTime))+sine.phase)));
  servo.force.f=sine.y;
  servo.F=servo.force.f;
  servo.slidingMass.flange_a.f+servo.springDamper.flange_b.f=0.0;
  servo.slidingMass.flange_a.s=servo.springDamper.flange_b.s;
  servo.slidingMass1.flange_b.f+servo.springDamper.flange_a.f=0.0;
  servo.slidingMass1.flange_b.s=servo.springDamper.flange_a.s;
```

```
  servo.force.flange_b.f+servo.slidingMass1.flange_a.f=0.0;
  servo.force.flange_b.s=servo.slidingMass1.flange_a.s;
  servo.slidingMass.flange_b.f+servo.speedSensor.flange_a.f=0.0;
  servo.slidingMass.flange_b.s=servo.speedSensor.flange_a.s;
  servo.speedSensor.v=servo.y;
constraint
  servo.m1>=0.5;
  servo.m1<=1.5;
  s0>=-(1);
  s0<=1;
end OptimicaTests.ServoParameterOptimization;
```

E

A Modelica Model of a Plate Reactor

The reactor model in this appendix was developed by Staffan Haugwitz.

```
model PlateReactor "plate reactor with concurrent (parallel) flow"

  // the vector nspecies represents the reactants
  // A and B and the product C
  // the current reaction is then A + B = C

  // n = 30, number of discretized elements

  parameter Integer inj_B_in_elem = 16;

  parameter Integer n=30 "discretization number";
  parameter Real V_reac = 4.1667e-4/n "V per element";
  parameter Real V_sp = 2*2.9e-4/n "V per element";
  parameter Real Aheat = 2*0.52*0.07398/n "A per element";

  parameter Real d=1000 "density";
  parameter Real cp=4180 "specific heat capacity";
  parameter Real k=3000 "convection heat transfer coeff";
  parameter Integer nspecies=3 "number of involving species";
  parameter Integer A=1
    "The reactant A corresponds to the first element
     in the nspecies list";
  parameter Integer B=2
    "The reactant B corresponds to the second element
     in the nspecies list";
  parameter Integer C=3
    "The product C  corresponds to the third element
     in the nspecies list";
  parameter Integer nreac=1 "number of reactions involved";
```

```
parameter Real Hr = 2*5.86e5 "enthalpy of formation";

parameter Integer stoich_coeff_A = -2
  "The stoichometric coefficient from the reaction formula,
   negative for reactants and positive for products";
parameter Integer stoich_coeff_B = -4;
parameter Integer stoich_coeff_C = 1;

parameter Real rateK0 = 2e7 "The reaction constant k0";
parameter Real Ea = 76000 "Activation energy";
parameter Real R=8.31434 "Universal gas constant";
parameter Real small=1e-9;
parameter Real q_reac_limit=1e-7;
parameter Real c_C_limit=1;

parameter Real c_nom = 2391.9/50;
parameter Real Jcf = Hr/(d*cp)*c_nom;
parameter Real aU = k*Aheat/(d*cp);
parameter Real Z = Ea*d*cp/(Hr*R*c_nom);

parameter Real q_B_total = 10e-3/3600;
parameter Real T_feedB = (273+20)/Jcf;
parameter Real cooling_flow = 10.8/3/3600;

parameter Real factor_c = 1.7;
parameter Real c_feedA = 1407*factor_c/c_nom;
parameter Real c_feedB = 11256*factor_c/c_nom;

parameter Real q_A = 40e-3/3600;
parameter Real u_B1_tau = 30
  "time constant of injection flow dynamics";

// Default initial conditions, no injections and everything is cold
Real[n] Tr(start = 303*ones(n)/Jcf,max=(273+155)/Jcf*ones(n));
Real[n] Tc(start = 293*ones(n)/Jcf);
Real[n] cA(start = 2392/c_nom*ones(n)) "Conc A in reactor";
Real[n] cB(start = 0*ones(n)) "Conc B in reactor";
Real T_feedA(start= 303/Jcf);
Real T_cool(start= 293/Jcf);
Real u_B1(start = 0);
Real u_B2(start = 0);
Real ub1f1( start=0);
Real ub1f2( start=0);
Real ub1f3( start=0);

Real[n] q_reac
"The total flow rate in each discretized part of the reactor";
```

```
  Real[n] q_B;

  Real q_cool_in "utility volume flow into pipe 2";
  Real Tr_in "Temperature of fluid at pipe 1 entrance";
  Real Tc_in "Temperature of fluid at pipe 2 entrance";
  Real c_A_in
    "Inlet concentration of reactant A at pipe 1 entrance";
  Real c_feedB_in
    "Injection concentration of reactant B, constant at each
     injection site";

  parameter Real af11 = -0.6465642069286;
  parameter Real af12 = -0.52235971683047;
  parameter Real af13 = 0.20092585654438;
  parameter Real af21 = 0.52235971683047;
  parameter Real af22 = -0.07798518562701;
  parameter Real af23 = 0.13233112279764;
  parameter Real af31 =  0.20092585654438;
  parameter Real af32 = -0.13233112279764;
  parameter Real af33 = -0.27545060744436;

  parameter Real bf1 = -101.272744990762;
  parameter Real bf2 = 23.164289092889;
  parameter Real bf3 = 16.745608714377;

  parameter Real cf1 =  101.272744990762;
  parameter Real cf2 = 23.164289092889;
  parameter Real cf3 = -16.745608714377;

  parameter Real df = 10000;

  Real u_B1_setpoint_f(start=0);
  Real ub2f1( start=0);
  Real ub2f2( start=0);
  Real ub2f3( start=0);
  Real u_B2_setpoint_f(start=0);

  Modelica.Blocks.Interfaces.RealInput u_B1_setpoint;
  Modelica.Blocks.Interfaces.RealInput u_B2_setpoint;
  Modelica.Blocks.Interfaces.RealInput u_TfeedA_setpoint;
  Modelica.Blocks.Interfaces.RealInput u_T_cool_setpoint;
equation
  der(ub1f1) = af11*ub1f1+af12*ub1f2+af13*ub1f3 + bf1*u_B1_setpoint;
  der(ub1f2) = af21*ub1f1+af22*ub1f2+af23*ub1f3 + bf2*u_B1_setpoint;
  der(ub1f3) = af31*ub1f1+af32*ub1f2+af33*ub1f3 + bf3*u_B1_setpoint;
  u_B1_setpoint_f = cf1*ub1f1+cf2*ub1f2+cf3*ub1f3 + df*u_B1_setpoint;
```

```
  der(ub2f1) = af11*ub2f1+af12*ub2f2+af13*ub2f3 + bf1*u_B2_setpoint;
  der(ub2f2) = af21*ub2f1+af22*ub2f2+af23*ub2f3 + bf2*u_B2_setpoint;
  der(ub2f3) = af31*ub2f1+af32*ub2f2+af33*ub2f3 + bf3*u_B2_setpoint;
  u_B2_setpoint_f = cf1*ub2f1+cf2*ub2f2+cf3*ub2f3 + df*u_B2_setpoint;

  der(u_B1) = 1/1*(u_B1_setpoint - u_B1);
  der(u_B2) = 1/1*(u_B2_setpoint - u_B2);
  der(T_feedA) = 1/2*(u_TfeedA_setpoint - T_feedA);
  der(T_cool) = 1/4*(u_T_cool_setpoint - T_cool);
  Tr_in = T_feedA;
  c_A_in = c_feedA;
  q_cool_in = cooling_flow;
  Tc_in = T_cool;
  c_feedB_in = c_feedB;
  q_B[1] = u_B1*q_B_total;
  q_B[2:inj_B_in_elem-1] = zeros(inj_B_in_elem-2);
  q_B[inj_B_in_elem] = u_B2*q_B_total;
  q_B[inj_B_in_elem+1:n] = zeros(n-inj_B_in_elem);
  q_reac[1] = q_A + q_B[1];
  der(Tr[1]) = 1/V_reac*(q_B[1]*T_feedB + q_A*Tr_in - q_reac[1]*Tr[1] +
               aU*(Tc[1] - Tr[1])  + V_reac*rateK0*exp(-Z/Tr[1])*
               cA[1]*cB[1]*c_nom);
  der(Tc[1]) = 1/V_sp*(-q_cool_in*Tc[1] + q_cool_in*Tc_in -
               aU*(Tc[1] - Tr[1]));
  der(cA[1]) = 1/V_reac*(q_A*c_A_in - q_reac[1]*cA[1] +
               stoich_coeff_A*V_reac*rateK0*exp(-Z/Tr[1])*
               cA[1]*cB[1]*c_nom);
  der(cB[1]) = 1/V_reac*(q_B[1]*c_feedB_in - q_reac[1]*cB[1] +
               stoich_coeff_B*V_reac*rateK0*exp(-Z/Tr[1])*
               cA[1]*cB[1]*c_nom);
  for i in 2:n loop
  q_reac[i] = q_A + sum(q_B[1:i]);
  der(Tr[i]) = 1/V_reac*(q_B[i]*T_feedB + q_reac[i-1]*Tr[i-1] -
               q_reac[i]*Tr[i] + aU*(Tc[i] - Tr[i]) +
               V_reac*rateK0*exp(-Z/Tr[i])*cA[i]*cB[i]*c_nom);
  der(Tc[i]) =  1/V_sp*(q_cool_in*(Tc[i-1] - Tc[i]) -
                aU*(Tc[i] - Tr[i]));
  der(cA[i]) = 1/V_reac*(q_reac[i-1]*cA[i-1] - q_reac[i]*cA[i] +
               stoich_coeff_A*V_reac*rateK0*exp(-Z/Tr[i])*
               cA[i]*cB[i]*c_nom);
  der(cB[i]) = 1/V_reac*(q_B[i]*c_feedB_in + q_reac[i-1]*cB[i-1] -
               q_reac[i]*cB[i] + stoich_coeff_B*V_reac*
               rateK0*exp(-Z/Tr[i])* cA[i]*cB[i]*c_nom);
  end for;
end PlateReactor;
```