Aalborg Universitet

A Quantised State Systems Approach Towards Declarative Autonomous Control

Alminde, Lars

Publication date:
2009

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

Link to publication from Aalborg University

Citation for published version (APA):
Alminde, L. (2009). *A Quantised State Systems Approach Towards Declarative Autonomous Control*. Automation and Control, Department of Electronic Engineering, Aalborg University.

General rights
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 -

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

Downloaded from vbn.aau.dk on: September 27, 2024

A Quantised State Systems Approach Towards Declarative Autonomous Control

Ph.D. Thesis

Lars Alminde

Department of Electronic Engineering
Section for Automation and Control
Aalborg University
Fredrik Bajers Vej 7, 9220 Aalborg East, Denmark

January 29, 2009

Thesis title: A Quantised State Systems Approach Towards Declarative Autonomous Control

ISBN 978-87-90664-38-1
January 2009

Copyright 2004–2009 © Lars Alminde

This thesis was typeset using LaTeX 2_ε in `report` document class.

Preface

This thesis is submitted as partly fulfilment of the requirements for the Doctor of Philosophy at the Department of Electronic Systems, Aalborg University, Denmark. The work has been carried out in the period August 2004 to January 2009 under the supervision of Professor Jakob Stoustrup and Associate Professor Jan Dimon Bendtsen.

Aalborg University, January 2009
Lars Alminde

Abstract

This thesis concerns methods for developing control software for autonomous systems with a high level of modularity, striving towards a declarative control paradigm, where the control system is able to solve control and estimations tasks based on system and objective descriptions alone. This is pursued on the basis of Quantised State Systems (QSS) which is a recent formalism for working with systems of ordinary differential equations in an computing environment based on discrete interactions between model entities.

At first; Quantised State Systems are introduced together with the Discrete EVent Systems (DEVS) formalism used to implement them. A comparative study on simulation performance compared to traditional time-discrete methods is given for an autonomous underwater vehicle.

Next, a novel Extended Kalman Filter (EKF) variation is developed which utilises the QSS approach to allow Jacobian free estimation with discrete event inputs. This approach is compared to a traditional EKF implementation on an example concerning attitude determination for a deep-space-probe.

Two different control strategies are thereafter developed based on the QSS approach; an optimising general controller that uses local information to provide an input signal that minimises a user supplied objective function of the state, and a controller based on sliding mode control which is highly modular and also allows configuration by supplying an objective function.

The aforementioned QSS based estimator and control algorithms are evaluated in a closed-loop control setting with a high-fidelity simulation model simulating a Deep Space Probe conducting a Jovian fly-by. The results favours the sliding mode controller in terms of both performance and robustness.

A simulation architecture for Hybrid System models are developed, allowing translation from XML specifications of hybrid models into run-time representations. It is demonstrated how the tools developed for hybrid model simulation can be combined with the aforementioned algorithms for continuous control to implement hybrid supervisory control systems. Finally, it is proposed how the different algorithms and tools can be combined into a declarative control system.

Synopsis - Danish Abstract

Nærliggende afhandling omhandler metoder til udvikling af software til kontrol at autonome systemer med en høj grad af modularitet. Udviklingen sigter imod et deklarativt kontrol paradigme, hvor kontrol systemet bliver i stand til løse estimations- og kontrolopgaver baseret på modelbeskrivelser og målbeskrivelser. Denne målsætning forfølges på basis af Kvantiserede State Systemer (KSS), som er en ny formalisme for behandling af ordinære differentialligninger i et computersystem baseret på diskrete interaktioner imellem komponenter.

Indledningsvis beskrives Kvantiserede State Systemer sammen med en specifikation af Discrete EVent Systems (DEVS), som er en formalisme der implementerer KSS systemer. Et simuleringstudie sammenligner KSS simulering med traditionelle tidsdiskrete metoder for et eksempel omhandlende en autonom undervandsbåd.

Derefter udvikles en ny variation af den kendte Extended Kalman Filter (EKF) algoritme baseret på KSS systemer. Denne tilgangsvinkel tillader estimering uden kendskab til Jacobian matricen for systemet og understøtter event baserede målinger. Den nye algoritme sammenlignes med den traditionelle EKF algoritme på et eksempel omhandlende en interplanetarisk probe.

To forskellige reguleringsalgoritmer er herefter udviklet med base i KSS tilgangen; en optimerende regulator, som benytter lokal information til at udlede et styresignal der minimerer en brugerdefineret målfunktion, og en modulær opbygget controller baseret på teorien om glidende manifolder, som også styres af en brugerdefineret målfunktion.

De førnævnte KSS baserede værktøjer til estimering og kontrol er evalueret på et reguleringsproblem omhandlende en interplaneterisk probe der flyver forbi Jupiter.

En simuleringsarkitektur for modeller af hybride systemer er udviklet, som tillader oversættelse fra XML specifikationer til software-objekter. Det er demonstreret hvordan disse værktøjer for hybrid simulering kan kombineres med førnævnte algoritmer og implementere hybrid control systemer. Endeligt, er det forelsået hvordan resultaterne kan videreudvikles henimod et deklarativt kontrol system.

Acknowledgements

I would like to thank my two supervisors associate professor Jan Dimon Bendtsen and professor Jakob Stoustrup for continuous support and input during the past three-and-a-half years. I have been privileged to pursue my own ideas to a great extend during this work and appreciate the patience and trust granted me in so doing. I hope we will continue to work as partners on joint projects in the future.

A special thanks to Jan Dimon Bendtsen and Karl Kaas Laursen for the fruitful discussions of our ad-hoc discussion group about hybrid systems and Java technology during 2005.

I would also like to thank professor Kristin Ytterstad Petersen for hosting me at the Norwegian University of Science and Technology during the fall of 2006. It was a pleasant period and opened my eyes to the fascinating world of underwater robotics.

Thanks are also due to all my colleagues at the section of Automation and Control at Aalborg university with whom I have shared many exciting experiences over the years - satellite launches not least. A special thanks to Karen Drescher for help tackling the bureaucracy seemingly inherent to life at the university.

A very special thanks is also due to Morten Bisgaard with whom I have had an excellent cooperation over the last past 8 years. In the same paragraph I wish to thank Tor Viscor for friendship over the years.

Finally, the greatest thanks goes to Birgit Krogh for love and support, and for clearing the clouds when it all looked almost impossible. A special thanks for the patience you showed during the last months.

Contents

1 Introduction **1**

1.1 Background and Motivation 1
1.2 Related Work 7
1.3 Contributions 11
1.4 Thesis Outline 12
1.5 Chapter Summary 15

I Preliminaries **17**

2 Discrete Event Systems **19**

2.1 Background and Motivation 19
2.2 DEVS - Discrete Event Specification 20
2.3 A Software Framework for DEVS 24
2.4 Discussion 27
2.5 Chapter Summary 28

3 Quantised State Systems **29**

3.1 Motivation for Quantised State Systems 29
3.2 The QSS2 Method - First Order Quantisation 31
3.3 Properties and Benefits of QSS2 38
3.4 Simulation of an Autonomous Underwater Vehicle 42
3.5 Chapter Summary 48

II Estimation and Control using Quantised State Systems **49**

4 Kalman Filter Estimation in QSS **51**

4.1 Introduction 51
4.2 Review of Extended Kalman Filtering 51
4.3 Extended Kalman Filtering in Quantised Systems 54
4.4 Simulation Case Study: Attitude Determination 59
4.5 Chapter Summary 68

5 Optimising Control of QSS Systems **69**

5.1 Event Based Control and Quantised State Systems 69
5.2 A Simple Optimising QSS2 Controller 70
5.3 Stability Analysis 74
5.4 Extension to Multiple Objective Control 79
5.5 Control Algorithm Summary and Implementation 79
5.6 Control of an Autonomous Underwater Vehicle 81
5.7 Chapter Summary 87

6 Sliding Mode Control in QSS Systems **89**

6.1 Introduction 89
6.2 Sliding Mode Stabilisation of MIMO Systems 90
6.3 A QSS2 Implementation of Sliding Mode Control 96
6.4 Simulation Results for a Deep Space Probe 101
6.5 Chapter Summary 107

7 Evaluation of Estimation Based Control **109**

7.1 Introduction and Infrastructure for Evaluation 109
7.2 Case Study Details 110
7.3 Optimising Control Results 114
7.4 Sliding Mode Control Results 119
7.5 Chapter Summary 124

III Hybrid Systems, Simulation and Control **125**

8 Hybrid Systems and QSS Based Simulation **127**

8.1 Introduction and Motivation 127
8.2 Hybrid System Models 128
8.3 Hybrid System Execution in DEVS/QSS 133
8.4 Declaring Models for Simulation/Execution 139
8.5 Simulation of Raibert's Hopper 140
8.6 Matlab Comparison 147
8.7 Chapter Summary 149

9 Towards Declarative Hybrid Supervisory Control **151**

9.1 Introduction 151
9.2 Hybrid Supervisory Control Example 152
9.3 Towards a Methodological Approach 154
9.4 Chapter Summary 159

IV Closure **161**

10 Concluding Remarks **163**

10.1 Summary of the Results 163
10.2 Conclusions on Research Objectives and Contributions 167
10.3 Recommendations for Future Work 169

V Appendices **171**

A Hybrid Systems and XML Specifications **173**

A.1 Definition of a Hybrid System 173
A.2 Specialised Specifications 174
A.3 Composition of Hybrid Systems 176

A.4 Overview of XML Tags for Defining a Hybrid System 179
A.5 Document Type Definition for a Hybrid System 180
A.6 Example of Subsystem Specification 182

B Software Overview 187
B.1 Obtaining the Software 187
B.2 Software Structure 187
B.3 Getting Started 189

C Code Examples 191
C.1 Code Example for Deep Space Probe Case 191

Bibliography 195

List of Figures

1.1 Advances in control engineering enabled the Apollo missions . . . 2
1.2 The concept of a declarative control system 4
1.3 Elements in an implementation of a declarative control system . . 5
1.4 An artist's conception of the Deep Space One spacecraft 9
1.5 Closed loop sampled system 11
1.6 Mapping from chapters to the DCS structure 14

2.1 An example of a DEVS model hierarchy 20
2.2 Coupled DEVS model 22
2.3 Class diagram in UML for the DevsCore package 25
2.4 Sequence diagram showing how classes interact during a simulation 26

3.1 Comparison between time quantisation and state quantisation . . . 30
3.2 Illustration of the calculation of the time until the next event. . . . 31
3.3 Structure of a QSS2 simulation 32
3.4 A Sample QSS2 Trajectory 34
3.5 Class diagram for the QSS2 package 38
3.6 Lightly damped oscillator 41
3.7 System with different response time-scales 42
3.8 The Naval Postgraduate School Autonomous Underwater Vehicle 43
3.9 A slow turn of the NPSAUV 45
3.10 Non-nominal initial conditions 47

4.1 EKF temporal flow 54
4.2 QSS/EKF data flow 56
4.3 Block diagram for a QSS based EKF implementation 57

4.4 A deep space mission scenario 60
4.5 Attitude sensors for the DSP 62
4.6 DSP simulation results 64
4.7 EKF performance 65
4.8 QSS/EKF filter performance 66
4.9 QSS/EKF Performance with increased quantum 67

5.1 Feedback control structure 69
5.2 QSS Control structure 72
5.3 The MSP control strategy 78
5.4 MOC results for AUV states 83
5.5 MOC results for AUV control objective functions 84
5.6 SOC results for AUV states 86
5.7 SOC objective function vs. MOC summed objective functions . . 87

6.1 Illustration of the chattering phenomena. 93
6.2 Sliding mode controller structure 97
6.3 Adding dynamic uncertainty bounds to the controller structure. . . 101
6.4 Simple SMC control results, a large degree of chattering is evident 103
6.5 SMC control results with approximated switching function 105
6.6 SMC control results with performance tuned reaching law 106
6.7 SMC control in steady state 107

7.1 The infrastructure for estimation based evaluation 110
7.2 Inspiration for the case study; A Jovian fly-by. 111
7.3 Class diagram for DEVS model with IO to/from *Simulink*. 113
7.4 The steps in the rendezvous protocol. 113
7.5 Jupiter gravity assist results with the optimising controller 116
7.6 Results with filtered inputs and exaggerated disturbance 118
7.7 Sliding mode results 121
7.8 Results with only Equivalent Control 122

7.9 Results with IMU 123

8.1 A graphical representation of a hybrid system model. 131
8.2 Detecting events consistently in sampled systems 134
8.3 Architecture for simulating and executing hybrid system models . 135
8.4 The Hybrid Location Control class 136
8.5 Schematic drawing of Raibert's Hopper [Back et al., 1993]. 141
8.6 The different location in the Raibert's Hopper model 142
8.7 Graphical presentation of a hybrid model for Raibert's Hopper . . 143
8.8 Raibert's Hopper under stable hopping motion 145
8.9 Raibert's Hopper - Getting stuck in decompression 146
8.10 Transition detection comparison to Matlab 148

9.1 Layers of capabilities as developed during the dissertation 151
9.2 Hybrid model for the extended DSP Case 153
9.3 Jupiter gravity assist with side-scanning motion. 155
9.4 Methodology 156

A.1 Composition of two hybrid systems 177

List of Tables

3.1 Quanta selection for QSS2 simulation 46
3.2 Number of outputs for each state during QSS2 simulation 46

5.1 Actuator signal updates to the model. 85

8.1 Tags in XML descriptions of hybrid system models 132

List of Algorithms

2.1 Behaviour of coupled models 23
2.2 A simple runner algorithm with external I/O 24
3.1 DEVS implementation of a QSS2 integrator 37
3.2 DEVS implementation of QSS2 function map 37
5.1 The Controller Block 80

1 Introduction

This thesis concerns development of methods and tools to enable declarative solutions to control problems, i.e. the development of generic algorithms that can easily adapt to problem specific model descriptions. This chapter provides motivation for the work and an overview of the thesis and its contributions.

1.1 Background and Motivation

Control problems relating to human artifacts date back to ancient times when Egyptian inventor Ctesibius invented the float valve as a feed-back device in water clocks to help keep a constant pressure and hence increase accuracy. As civilisation entered the industrialised age more industrial processes needed to be controlled and in 1788 James Watt developed his famous fly-ball governor to control the shaft speed in steam engine powered equipment.

Until this time all control inventions were based on physical insight and ingenuity rather than mathematical analysis, and the control inventions were implemented by mechanical modifications to the system to be controlled. However, the 19th century saw the development of the first mathematical analysis tools for control problems, pioneered by names such as James C. Maxwell, E. J. Routh, and A. M. Lyapunov.

At the start of the 20th century the invention of flight and the rapid deployment of combustion engines inspired increased research in control problems. Also in these years the development of electronics allowed controllers to be implemented in electric circuits as an alternative to mechanical devices. Frequency design methods, well suited for implementation using operation amplifier technology, were developed during the time with H. W. Bode and H. Nyquist as leading researchers.

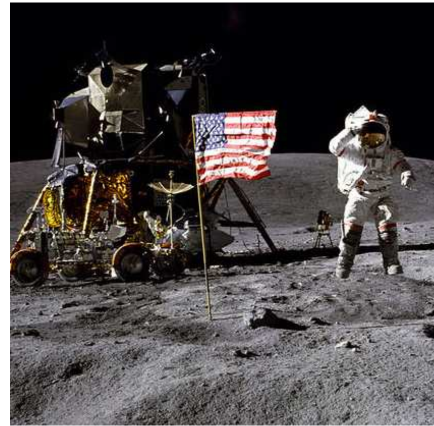


Figure 1.1: Advances in control engineering and digital computers enabled the Apollo missions to the Moon during the sixties and seventies.

Following the second world war advances in control engineering became necessary for the super powers in support of the raging space race. This time saw the development of state-space methods, Kalman filtering, and optimal control theory among others.

Since then there has been a continued research effort to provide increasingly complex algorithms to deal with challenging control problems such as e.g. non-linearity or embedded discrete behaviour. One trend that can be identified is that the role of the digital computer continues to be more dominant throughout all phases from analysis and modelling to implementation of control laws. Recent methods such as e.g. non-linear model predictive control or particle filtering emphasises this trend.

1.1.1 The Gap Between Theory and Application

As argued above there exists today a large body of advanced theory to address challenging control problems. However, it is reasonable to ask if this theory is being applied in applications today? It is a well known saying in the control community that "In industry they always implement PID* controllers anyway".

A good motivation for why this might be true to a large extend is that PID control is available out-of-the-box from e.g. Programmable Logic Control (PLC)

*Proportional Integral Derivative control

units, which can be installed by an electrician, who can also tune the control parameters by himself. On the other hand application of more recent and advanced control theory require domain experts and control experts to work together for extended time during which they will gradually progress from problem definition to implementation of a control system.

In other words the investment required to apply modern control techniques at typical industrial fabrication facilities can be prohibitive. Hence, in order to make advanced control methods more cost effective, we must look at ways to make them easier to apply to real-life problems without extensive expert assistance. The price that is paid for not implementing the advanced control methods is increased resource consumption, reduced process quality and increased wear in machinery.

As an analogy, consider the advancement in computer graphics over the last decades; here also a very large body of theory has been developed and the methods also rely on digital computers for implementation. However, all of this research is available to the mainstream consumer in software packages such as GIMP[†] or Photoshop[‡], where the consumer can apply the latest results in wavelet analysis and related methods, when enhancing family photos. The user knows nothing about what is going on but is simply presented with a dialogue box with some intuitive parameters to adjust (e.g. sharpness). With this example in mind the following ultimate goal is presented for the research pursued in this thesis:

Ultimate Goal 1: *"Application of advanced control theory to real-life control challenges is as easy as enhancing family photos in Photoshop."*

The next subsection will elaborate this goal into more concrete terms.

1.1.2 A Declarative Control System

In Computer Science; programming languages are often categorised as supporting one or multiple programming paradigms, for example C++ is said both to be a procedural and object-oriented programming language. At the highest level one can distinguish between *imperative programming* and *declarative programming*. The first concept is the most well-known and refers to languages where the program elements describe ways to manipulate data, e.g. the implementation of an

[†] http://www.gimp.org
[‡] http://www.adobe.com/products/photoshop

algorithm. This way of thinking is typical in languages such as C/C++, Pascal, and Java.

Declarative programming on the contrary focuses on declaring the problem rather than specifying solutions in terms of algorithms. For example the XSLT[§] language is used to describe transformations between eXtended Markup Language (XML) documents by describing relations-ships rather than procedures for transformation. Other examples of declarative languages are LISP [Seibel, 2005] and Prolog [Callear, 2003]. With this terminology from computer science we are inspired to develop a system that can solve control problems declaratively:

Definition 1.1 (Declarative Control System (DCS))
A Declarative Control System is a system capable of controlling a plant based on a description of the plant and a description of the performance that must be obtained.

Certainly if the advanced control theories that exists today can be packaged in a DCS such that the algorithms automatically at *run-time* can adapt to the supplied models then these methods are much easier to apply for the technician in the field.

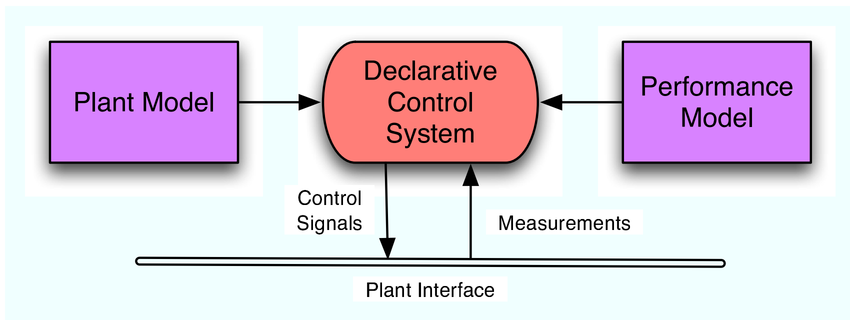


Figure 1.2: The concept of a declarative control system - accepting plant and performance models as basis for computation.

Figure 1.2 depicts this graphically; the DCS is fed with a model description (including all available sensors and actuators) and a performance specification. Based hereupon; the DCS is able to control the plant in real-time accepting measurements from the plant and supplying control signals to it. Figure 1.3 provides

[§]Extensible Style-sheet Language Transformations

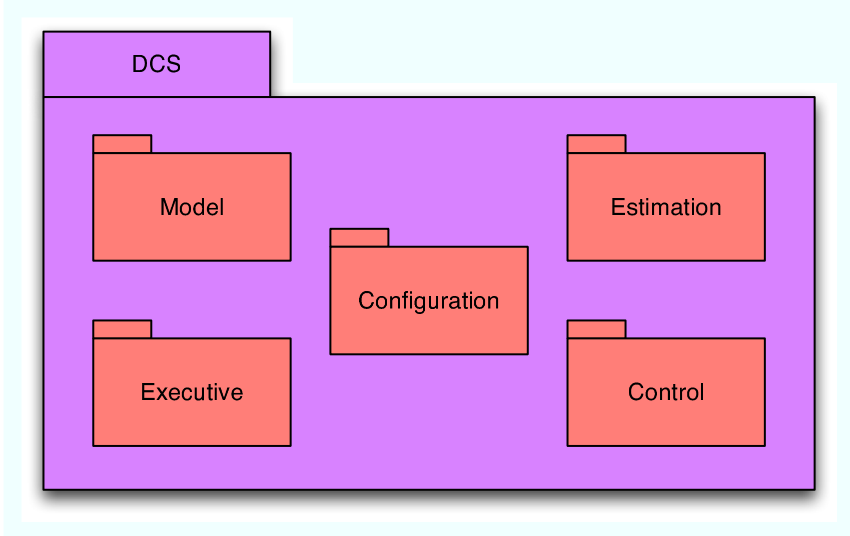


Figure 1.3: The different architectural elements in an implementation of a declarative control system.

a general overview of the software modules that must be developed in order to implement the DCS functionality. The following gives a brief discussion of each module:

`Model` this module must implement software which can represent the plant and performance model as described by the user. This entails software representations for integrators, function maps, and other model elements.

`Estimation` this module must implement various estimation algorithms that can be used to filter data from plant sensors.

`Control` here various control algorithms must be implemented generically such that they can be bound to the user supplied model without any modifications to the algorithms.

`Configuration` this module is responsible for analysing the user supplied models and set up relations (at run time) between model elements and algorithms, both estimation and control. This includes choosing which algorithms are best suited to the problem.

`Executive` based on the relations set up between components by the configuration module the executive module is responsible for executing the algorithms

and interacting with the plant.

Deploying the DCS architecture fully is beyond the scope of a single PhD project and the next subsection will limit the scope of the present study and present specific research objectives pursued in this thesis. Focus will be on methodologies that can support the idea of a DCS system.

1.1.3 Scope of this Study

Control and estimation algorithms such as e.g. sliding mode control and Kalman filtering replicate parts of the plant model in their structure; for example in Kalman filtering the plant dynamics is used to propagate states and covariance, and a sliding mode controller will use the plant control matrix to derive input signals. Typical implementations therefore combine model and algorithm elements in the same sequential code.

To enable the DCS architecture the algorithms must be implemented in a generic manner that allows them to be composed with any model that the user may supply. This is expressed in the following objective:

Research Objective 1: *"To provide and demonstrate a framework that allows control/estimation algorithms and plant models to be described independently and then be composed at run time"*

Today control system software is often developed and validated in a simulation environment, e.g. Matlab/Simulink and then at the end of the development cycle the solution is hand-implemented on the target system in code or the control software is auto-generated by the tool and then adapted to the target system. In both cases the programming paradigm used in the implementation code is that of structured programming.

For the framework to be developed this approach will be replaced with an Object Oriented (OO) implementation philosophy. There are two reasons for this. 1) for a framework that will grow over time the OO approach will be more maintainable and flexible and 2) the OO approach allows *encapsulation*, meaning that when parts of the system have been validated (perhaps even certified for safety-critical applications), these parts can remain fully encapsulated and further development on these parts of the system can be based on modifying behaviour in derived classes. These ideas have led to the formulation of the second research

objective:

Research Objective 2: *"To demonstrate the applicability of object oriented design to the domain of control systems software for on-line execution"*

The elements in the framework must be able to communicate in a structured manner during run-time where the elements together implement the control systems functionality. Most control system implementations rely on a sample driven approach. However, recent research [Kofman, 2002] has pointed to Quantised State Systems (QSS) as an alternative approach to deal with system dynamics in a computing environment. With this approach communication between model entities are based on events, which are dispatched when each component undergoes a significant internal state change. This approach promises to be more efficient in terms of required computing resources than the typical sample driven approach. Therefore it has been chosen to use this QSS approach for the implementation of the framework:

Research Objective 3: *"To demonstrate and evaluate a Quantised State Systems approach to control systems software in contrast to typical sample driven implementations"*

With the formulation of these research objectives most of the functionality of the DCS system described in the previous subsection is addressed except for the functionality in the `Configuration` module. However, the result of the work to be presented in this thesis will be a good starting point to address the issues of automatic model analysis and coupling to relevant algorithms.

1.2 Related Work

It has been noted by other researchers that most control systems are first developed on a mathematical foundation using tools such as Matlab/Simulink to analyse, design, and verify the algorithms of the control system, whereafter the design is implemented on the target platform with little regard for the inherent timing issues involved with consistent implementation of control software on embedded computing platforms.

Discussions in [Horowitz et al., 2003, Henzinger et al., 2003] points to the

same issues regarding implementation of control software and proposes a platform approach where a special purpose kernel, dubbed Giotto, implements the interface to the specific real time operating system of the control computer.

In [Koo et al., 2005] a framework called *ReachLab* is presented which allows hybrid dynamical models as well as analysis algorithms to be described in an abstract language: Hybrid System Analysis and Design Language (HADL). The framework allows such models to be translated to the language of a number of *computational kernels*, which are independent environments for analysis of hybrid models.

During the nineties NASA launched a number of independent deep space exploration probes and noticed that almost no control software was reused from mission to mission, and that the implementations lacked a common methodology to help avoid implementation mistakes [Dvorak et al., 2000]. This led to the formulation of the *mission data system (MDS)*, a vision for future control software, which emphasises a number of themes to be addressed, of which many are relevant to the present thesis; the most relevant themes are listed below by their short descriptions from [Dvorak et al., 2000]:

1. Construct subsystems from their architectural elements, not the other way around
2. Design interfaces to accommodate foreseeable advances in technology
3. System state and models form the foundation for information processing
4. Express domain knowledge explicitly in models rather than implicitly in programs
5. Operate missions via specifications of desired state rather than sequences of actions

Of these the first two are relevant in connection with the object oriented approach that will be taken in this thesis and the latter three relates to the concept of declarativity as described in the previous section. With offset in the MDS it is demonstrated [Dvorak et al., 2004] that object oriented software written in Java is suited for real-time execution for a Mars rover platform.

Many of the themes in the MDS were motivated from the Deep Space One (DS1) spacecraft launched by NASA on the 24th of October 1998 which featured the Remote Agent eXperiment (RAX) [Bernard et al., 1999]. It was an experimental flight software which featured intelligent on-line planning and execution.

Figure 1.4: An artist's conception of the Deep Space One spacecraft, which featured the Remote Agent eXperiment (RAX).

One of the software modules was the Mode Identification and Recovery (MIR) system [Williams and Nayak, 1999]; this system was a declarative system which was able to fulfil configuration goals, set by the planner component of RAX, by using models of the spacecraft. Further, it was able to identify faults in hardware and autonomously plan around them to the extent possible. However, the system only considered discrete models and discrete switches as actuators.

Despite the fact that an evaluation of the RAX experiment [Bernard et al., 2000] concluded that the technology could be applied to future missions this has not happened as of yet. However, parts of RAX have seen further theoretical development, e.g. described in [Williams et al., 2003].

1.2.1 Declarative Algorithms

Focusing on specific algorithmic approaches to automatic controller synthesis there have been significant results in recent times. One methodology is Model Predictive Control (MPC) [Maciejowski, 2002], where optimisation is used to derive optimal input sequences based on a specified performance index. MPC can be applied to a large class of systems, e.g. hybrid systems [Bemporad and Morari, 1999], and results in either an explicit controller derived off-line for simple systems or an implicit controller that solves the optimisation problem on-line, e.g. for non-linear systems. However, MPC in its implicit version can often be too computationally intensive to support on-line implementation [Oort et al., 2006]. MPC methods are not researched in this thesis, but MPC algorithms are clearly

candidates for inclusion in the proposed DCS framework.

Other automatic synthesis methods focus on complete off-line synthesis of controllers, where the results is a bank of Piece-Wise-Affine (PWA) control laws that each apply to a constrained polytopic subset of the state-space. For instance, in [Rodrigues and How, 2003] general non-linear systems are automatically transformed into a number of local PWA models that approximate the original system. From these local models local PWA control laws are generated, which is shown to satisfy a global piece-wise-cubic Lyapunov function. In [Habets et al., 2006] a PWA approach is taken to provide automatic controller synthesis for hybrid automata where the models in each discrete location are PWA models.

1.2.2 Event Based Control

Another theme in this thesis is the use of Quantised State Systems (QSS) to represent dynamics, which is in contrast to typical discrete time representations. Previous work concerning QSS will be reviewed throughout the thesis, while this subsection will discuss other event-based approaches than QSS.

In [Sandee et al., 2005, Årzén, 1999] it is shown through simulations that for a regulation problem the processor load can be reduced significantly by enabling event driven sampling in the vicinity of the set-point, i.e. control updates are only performed when an error threshold is exceeded. This approach was validated experimentally in [Sandee et al., 2006]. Theoretical work based on this approach [Sandee and Heemels, 2006], has established ultimate boundedness criteria for the solution trajectories, which are similar to results for quantised state systems, which are used in this thesis.

Other work [Lunze, 1994, Philips et al., 2003, Heemels et al., 2006] focuses on the effects of quantisation of sensors and actuators (effects of analogue to digital conversion and digital to analogue conversions, see Figure 1.5). In [Heemels et al., 2006] a method is presented that minimises the effect of the quantisation on closed loop performance. [Philips et al., 2003] and [Sandee et al., 2006] both approach this problem by abstracting the continuous dynamics into a discrete event automata where each state represents a small hypercube in the state-space of the continuous system. These abstractions are then treated off-line using discrete event methods for assessing reachability of the automata.

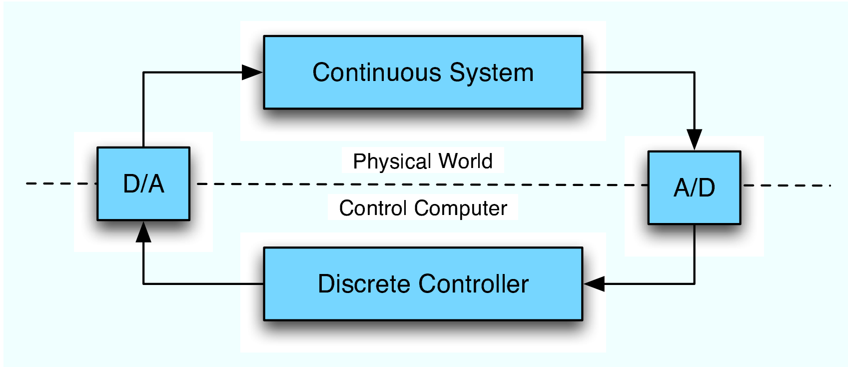


Figure 1.5: Closed loop system with digital to analogue and analogue to digital converters to interface between a continuous plant and discrete controller.

1.3 Contributions

The contributions of this thesis can be summarised in the following main points. Citations are provided for contributions which have been published.

- A comparison of performance and robustness between Quantised State Systems (QSS) and usual time discrete simulation for a model of an autonomous underwater vehicle, which is the first reported study on using QSS based simulation for simulation of non-linear systems with a high-dimensional state-space (to the author's knowledge).

- A novel generalisation of the well-known extended Kalman filter based on QSS models, which eliminates the need to analytically derive Jacobian matrices and allows the algorithm to be used declaratively. The filter accepts incoming measurements as events rather than equidistant samples. This work is published in [Alminde et al., 2007a].

- A novel declarative control strategy which utilises an on-line QSS model of the system under control to effectively derive small-signal models that is used for local minimisation of a convex control objective function. This work is published in [Alminde et al., 2007b].

- A generalisation of sliding mode control based on QSS models of the system under control with a guidance control based on minimisation of a user supplied

performance function. The algorithm implementation is highly modular and can be composed with models in a declarative manner

- The estimators and controllers developed for QSS have been verified in a comprehensive study of a deep space probe example complete with evaluation of the effect of uncertainties and disturbances.
- A specification of hybrid systems defined to reflect typical control systems and problems in contrast to many verification oriented specifications. This specification in various states of development is published in [Alminde et al., 2006a, Alminde et al., 2006b].
- A QSS approach for simulating hybrid systems as defined by the above specification and a mechanism for directly translating a specification, written in eXtended Markup Language, into an executable software object. Early version published in [Alminde et al., 2006a].
- Throughout the thesis methods are developed and demonstrated to support the idea of a declarative control system. These results are tied together in a software framework based on discretely interacting components.

An international journal publication is under preparation which summarises the results concerning the use of quantized state systems for control and estimation purposes.

1.4 Thesis Outline

The following gives a chapter by chapter overview of the contents of this thesis.

Chapter 1: Introduction
Provides motivation for the thesis and an overview of its structure and results.

Chapter 2: Discrete Event Simulation
This chapter introduces the Discrete EVent Specification (DEVS) which is utilised throughout the thesis as a framework for simulating/executing discrete event models.

Chapter 3: Quantised State Systems
The concept of Quantised State Systems (QSS) is central to this thesis and is de-

scribed in this chapter in the context of simulation of continuous system models. A comprehensive simulation study is conducted using a model of an autonomous underwater vehicle.

Chapter 4: Kalman Filter Estimation in QSS
Here it is shown how quantised state systems can be used to implement the extended Kalman filtering algorithm and how this approach makes analytical evaluation of Jacobians for covariance propagation unnecessary. The conventional and QSS filters are demonstrated and compared on a model of a deep space probe, which must determine its attitude from vector observations.

Chapter 5: Optimising Control of QSS Systems
This chapter develops a control strategy based on local models of a non-linear multiple-input-multiple-output plant generated by the QSS approach. At each control calculation a change in control signals is found from an optimisation problem based on the local model. The control strategy is evaluated on a model of an autonomous underwater vehicle.

Chapter 6: Sliding Mode Control in QSS Systems
Here a sliding mode controller for state stabilisation is developed within the QSS framework and demonstrated on a deep space probe example. Compared to the control strategy of the previous chapter the sliding mode approach provides better stability and robustness properties.

Chapter 7: Evaluation of Estimation Based Control
This chapter wraps up the work from the previous three chapters by providing a performance evaluation of the two proposed control strategies with the QSS based estimator in the loop. The evaluation uses the deep space probe as a case study.

Chapter 8: QSS Simulation of Hybrid Systems
Hybrid systems features both continuous state evolution as well as discrete events affecting the state evolution. This chapter describes how such systems can be simulated using quantised state systems and provides software that can translate a hybrid system model described in a dedicated language into a software entity that can be used for simulation or control.

Chapter 9: Towards Declarative Hybrid Supervisory Control
In this chapter the work on controllers for continuous systems and the work on hybrid models are combined to demonstrate how one can implement a hybrid supervisory control system with the tools developed. Further, it is discussed what future work is required to utilise the results for a methodological approach to declaratively specifying control systems.

Chapter 10: Conclusions and Perspectives
The final chapter wraps up the results and points to areas of further work.

Figure 1.6 provides a graphical overview of how the individual chapters contribute to the proposed DCS architecture from Subsection 1.1.2. As can be seen all of the architectural elements will be addressed throughout the thesis although only limited emphasis will be put on the `Configuration` module.

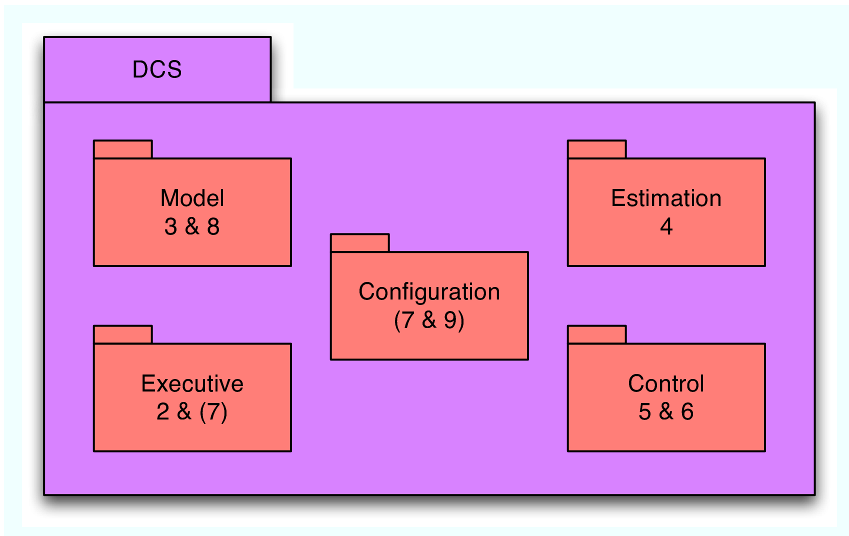


Figure 1.6: Mapping from chapters to the DCS structure. Chapter numbers in parenthesis signify minor contributions.

1.5 Chapter Summary

This introductory chapter proposed the development of a declarative control system to enable advanced control and estimation algorithms to be used as solutions to real-life challenges with reduced development effort. This idea was transformed into three research objectives that are pursued throughout this thesis. The three objectives are restated here:

Research Objective 1: *"To provide and demonstrate a framework that allows control/estimation algorithms and plant models to be described independently and then be composed at run time"*

Research Objective 2: *"To demonstrate the applicability of object oriented design to the domain of control systems software for on-line execution"*

Research Objective 3: *"To demonstrate and evaluate a Quantised State Systems approach to control systems software in contrast to typical sample driven implementations"*

Further, an overview of related work was presented and the structure of the thesis and its scientific contributions were summarised.

Part I

Preliminaries

Discrete Event Systems 2

This chapter introduces the Discrete EVent Specification (DEVS), which is a framework for simulating and executing systems characterised by their discrete interactions. This specification and associated software provide a tool that will be used to implement the algorithms developed in the remainder of this dissertation.

2.1 Background and Motivation

This dissertation concerns methods for declarative autonomy; these methods are based on the notion of Quantised State Systems (QSS), which is introduced in the next chapter. In short; the QSS approach transforms continuous dynamics into a discrete event system. In order to implement, on a computer, solutions based on QSS it is necessary to have a software framework that can execute these discrete event systems consistently.

Such a framework was introduced in [Zeigler, 1976] under the name "Discrete EVent Specification" (DEVS) and has seen a number of revisions and extensions since then, see e.g. [Zeigler et al., 2000]. The specification defines an abstract view of model components and their discrete interactions through message passing, and in addition algorithms for consistent model execution are specified.

Section 2.2 presents the DEVS specification and associated algorithms, while section 2.3 provides an overview of the software implementation of DEVS, which was developed as an execution platform for the algorithms to be developed throughout this dissertation. Further, details on the software framework, than given in this chapter, are available in Appendix B on page 187. The DEVS framework development is published as part of [Alminde et al., 2006a].

2.2 DEVS - Discrete Event Specification

The Discrete EVent Specification (DEVS) was formalised by Bernard P. Zeigler [Zeigler, 1976] as a language formalism for discrete event systems. In contrast to more widely adopted discrete event system descriptions [Cassandras and Lafortune, 1999], which enumerates all possible system configurations into a number of discrete states with associated transitions between, DEVS takes an alternative view and considers a number of units, called DEVS *atomic models*. These can implement complex processing, but interacts with other components through discrete interactions. DEVS has been applied for a wide range of modelling and simulation applications spanning from protocol verification to neural nets, see [Sarjoughian and Cellier, 2001] for an overview. A large number of DEVS variations have been presented throughout the years; This dissertation makes use of the DEVS specification in [Zeigler et al., 2000] dubbed "Classical DEVS with Ports".

The core entities in a DEVS model are model components called *atomic models*. Each atomic model is a self-contained system with an internal state and the ability to receive and send messages to/from other components through a number of enumerated input- and output ports respectively. Each port can be used to communicate one object, e.g. a real, a vector, or matrix. The behaviour of each atomic model is specified by a function describing behaviour when events are received and another function that describes internal autonomous state transitions.

Atomic models can be connected to form *coupled models*; it is the responsibility of the coupled model to handle message passing between ports of the atomic models contained in the coupled model and the in- and output ports of the coupled model to/from the contained atomic models. The coupled model has the same external interface as an atomic model meaning that a model hierarchy can be established, see Figure 2.1.

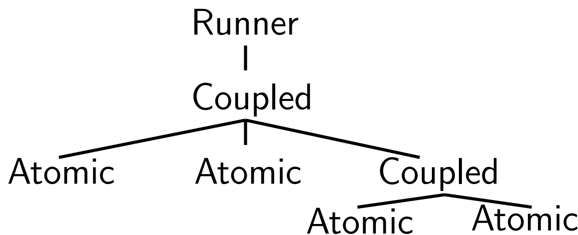


Figure 2.1: An example of a DEVS model hierarchy composed of atomic and coupled sub-models.

A stand alone atomic model or a coupled model is executed by a *runner* object which is in control of advancing time and mapping inputs and outputs from the outside (i.e. outside of the DEVS model) to the top level model component. The following subsections describe the three model components: atomic, coupled and runner objects in more detail.

2.2.1 Atomic DEVS Models

At first we define a message, $\mathcal{M}$, used for communicating events between different model components:

$$\mathcal{M} = \{m = (i, v) \mid i \in \mathbb{Z}^{+},\ v \in \mathbb{R}^{a \times b}\}$$

that is a set $\mathcal{M}$ consisting of pairs, m, described by a port identifier, i and a value, v. The value can represent a real, vector or matrix (depending on the dimensions: a, b). In the following e will be used to denote the time since the last event in the atomic model. A general atomic DEVS model is specified as an 8-tuple:

$$\mathcal{D} = (\mathcal{S}, \mathcal{X}, \mathcal{Y}, \delta_{int}, \delta_{ext}, \lambda, ta)$$

where:
$\mathcal{S}$: are internal states, representation is up to the user
$\mathcal{X} = \{x \in \mathbb{Z}^{+} | 1 \leq x \leq p_x\}$: is the set of input ports, p_x the number of inputs
$\mathcal{Y} = \{y \in \mathbb{Z}^{+} | 1 \leq x \leq p_y\}$: is the set of output ports, p_y the number of outputs

$\delta_{int}(e, \mathcal{S}) : \mathbb{R}^{+} \times \mathcal{S} \rightarrow \mathcal{S}'$: is the state transition function
$\delta_{ext}(e, \mathcal{M}, \mathcal{S}) : \mathbb{R}^{+} \times \mathcal{M} \times \mathcal{S} \rightarrow \mathcal{S}'$: is the external event function ($i \in \mathcal{X}$)
$\lambda(e, \mathcal{S}) : \mathbb{R}^{+} \times \mathcal{S} \rightarrow \mathcal{M}$: is the output mapping ($i \in \mathcal{Y}$)
$ta(e, \mathcal{S}) : \mathbb{R}^{+} \times \mathcal{S} \rightarrow \mathbb{R}_{+}$: is the time advance function i.e. time to next internal event.

The behaviour of a given model is implemented in the two transitions functions $\delta_{int}(\cdot)$ and $\delta_{ext}(\cdot)$, while the output, $\lambda(\cdot)$, and time advance, $ta(\cdot)$, functions are used by the coupled model or runner driving the atomic model for extracting output and scheduling information, respectively.

2.2.2 Coupled DEVS Models

Atomic models can be coupled as specified in a coupling specification. Consider Figure 2.2, which shows two atomic models that are coupled. There are three model entities in the figure; a_1 and a_2 are internal atomic models and c is the coupled model. There are connections between the two contained models and also connections between the contained models and the input and output ports of the coupled model.

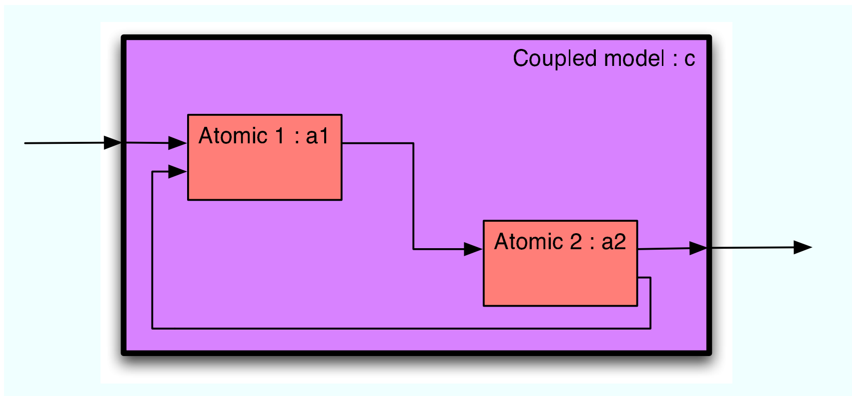


Figure 2.2: Coupled DEVS model consisting of two atomic sub-models, their interconnections and connections to the outside.

In the following c will mean the coupled model and $a_1, \ldots, a_l$ will denote atomic models that make up a coupled model. The coupling specification consists of three distinct sets with elements of the form $((m_1, p_1), (m_2, p_2))$, where m_j specifies a model component and p_j a port associated with that component. The three sets defining a coupled model are [Zeigler et al., 2000]:

EIC external input coupling, with m_1 being the coupled model, c, with $p_1 \in \mathcal{X}_c$ and $m_2 \in \{a_1, \ldots, a_l\}$ with $p_2 \in \mathcal{X}_{m_2}$.

EOC external output coupling coupling, with $m_1 \in \{a_1, \ldots, a_l\}$ and $p_2 \in \mathcal{X}_{m_1}$, and $m_2 \in \{c\}$ with $p_2 \in \mathcal{Y}_c$

IC internal coupling, with $m_1 \in \{a_1, \ldots, a_l\}$ and $p_1 \in \mathcal{X}_{m_1}$, and $m_2 \in \{a_1, \ldots, a_n\}$ with $p_2 \in \mathcal{X}_{m_2}$.

Further, a coupled model must adhere to the definition of an atomic model as defined in the last section; this means that coupled models can be internal parts of

other coupled models.

```
δ_int():
t_n  ← least m.ta(·) where m = {a_1, ..., a_n}
I ← set of all {m : m.ta(·) = t_l}
forall m ∈ I call m.δ_int(·)
forall m ∈ I call m.λ(·) and store messages in P
Generate new messages from P according to EOC and IC:
·  if (p ∈ P) ∈ EOC store as output of c
·  if (p ∈ P) ∈ IOC add to target to distribution set D
forall d ∈ D call δ_ext(·) on receiving model

δ_ext(P):  P is incoming messages
Generate new messages from P according to EOC and IC:
·  if (p ∈ P) ∈ EOC store as output of c
·  if (p ∈ P) ∈ IOC add to target to distribution set D
forall d ∈ D call δ_ext(·) on receiving model

λ():
Return stored output messages

ta(e):  e is time since last event
t_n  ← least m.ta(·) where m = {a_1, ..., a_n}
return t_n − e
```

Algorithm 2.1: Behaviour of coupled models

The behaviour of a coupled model is described in Algorithm 2.1, which gives an overview of how a coupled model interacts with other models; both with regard to internal models and to the outside world through the same mechanisms as for an atomic model.

2.2.3 Root Coordinator and Execution

In order to execute a model all components, both atomic models and coupled models, must be contained in one coupled model, which is then driven by a single software entity called a *Runner*. The runner is responsible for global time keeping and input/output management to/from the DEVS model. Different runners can be implemented to e.g. perform stand-alone executions or executions with input and output to local hardware or e.g. a network. Algorithm 2.2 gives an example of how such a runner could work together with external input and output (referred to as "ext.").

```
t ← startTime
while (t < endTime)
·   t_n ← t + c.ta()
·   if (ext. input before t_n)
·       call c.δ_ext(·)
·       t ←(time of ext. input)
·       continue
·   t ← t_n
·   call c.δ_int(·)
·   call c.λ(·) and write to ext.
```

Algorithm 2.2: A simple runner algorithm with external I/O

2.3 A Software Framework for DEVS

DEVS has mainly been used in academical projects and there are therefore no widespread commercial software packages for DEVS. However, a number of DEVS tools have been developed at university centres around the world of varying levels of maturity. An overview of available tools can be seen on **http://www.sce.-carleton.ca/faculty/wainer/standard/tools.html**.

For this project an implementation of the DEVS core functionality as described above was implemented in order to support the research objectives with a maximum degree of flexibility. The software was implemented in Java*. The following will provide an overview of the developed DEVS implementation and more details can be found in Appendix B on page 187 which describe all the software developed for this dissertation.

The DEVS implementation is implemented in the `DevsCore` package and an overview can be seen on the class diagram of Figure 2.3. The `DevsAtomic` class is an implementation of the atomic model specification given in Subsection 2.2.1 and `DevsCoordinator` is an implementation of the coupled model specified in Subsection 2.2.2, likewise the `DevsMessage` class is an implementation of the message construct introduced in Subsection 2.2.1 and finally the `DevsRunner` is a simple stand-alone runner as described in Subsection 2.2.3.

The `DevsAtomic` class is abstract and must be inherited in order to implement specific behaviour by overriding the abstract functions: $\delta_{int}(\cdot)$ and $\delta_{ext}(\cdot)$. Default behaviour for $\lambda(\cdot)$ (called `output()`) and $ta(\cdot)$ (called `timeAdvance()`) is implemented in the class for convenience but can likewise be overrided to suit

*Java Standard edition version 1.5

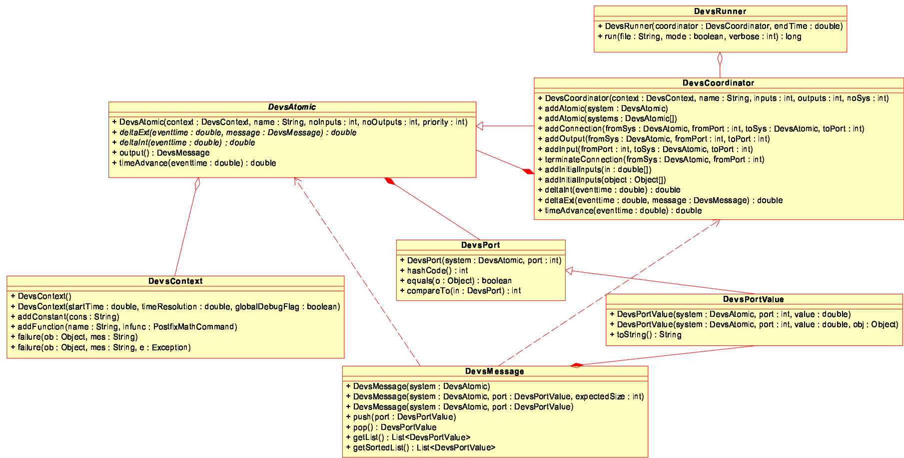


Figure 2.3: Class diagram in UML for the DevsCore package implementing DEVS core functionality.

individual needs.

The `DevsContext` class is not part of the DEVS specification, but it is included as a class to contain attributes and helper functions that can be shared between a number of `DevsAtomic` objects, for example numerical constants, definitions of mathematical functions, and functions for reporting errors and diagnostic information.

A DEVS model is constructed by defining a number of concrete `DevsAtomic` objects and adding them to a `DevsCoordinator` object using the `addAtomic()` call. Hereafter, connections are declared between the model components within the coordinator using the following functions calls: `addConnection()`, `addInput()`, `addOutput()`, and `terminateConnection()`. Finally, a runner object is constructed with the coordinator as argument and the model can then be executed by the `run()` method of the runner object.

Figure 2.4 provides an example sequence showing how a runner object performs one simulation/execution step for a configuration as indicated on Figure 2.2; As can be seen the coordinator is in charge of calling all atomic model components, while the runner only interacts directly with the coordinator as the top-level component in the model.

To enable real-time execution on an embedded platform the runner object should be implemented as a real-time thread constantly scheduling its next execu-

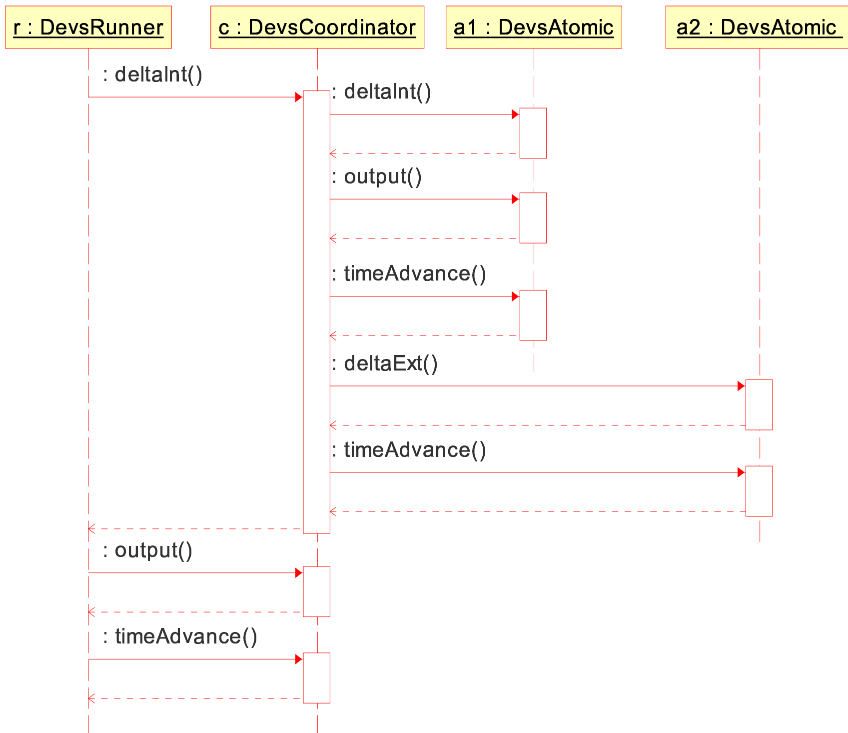


Figure 2.4: Sequence diagram showing how classes interact during a simulation. One simulation step.

tion at the next event time in the model. For such a scheme to work the underlying scheduler should use a dynamic scheduling principle, e.g. *earliest deadline first* scheduling.

2.3.1 Associated DEVS Tools

The software implementation described above constitutes the core functionality required to execute DEVS components. The remainder of this dissertation will develop various simulation, estimation and control algorithms, which are implemented on top of the DEVS implementation described above.

In addition a set of DEVS components have been developed which are not described in detail, but is used throughout the examples in this dissertation. These tools are standard tools to provide such functionality as for example:

- Reading and writing data to files during simulation/execution
- Plotting data at the end of simulation/execution runs
- Time discretisation of signals (sample-and-hold)

An overview of these tools can also be seen in Appendix B on page 187.

2.4 Discussion

A few points concerning the DEVS specification and implementation are found worthy of further discussion.

Push and Pull Communication in Devs
The DEVS specification, and its implementation as described above, is a very operational mechanism to implement discrete event based software entities that communicate on a subscription basis, or in Object Oriented jargon; components that are updated through the *observer* pattern [Gamma et al., 1994].

For model components that must always react to changes in its inputs immediately the subscription based communication is very adequate. However, as will be seen in the coming chapters, some model components may have inputs that are only used occasionally and therefore do not need to trigger recomputation in the component. This is easily adopted in the DEVS scheme by using subscription, and then just storing the input received from these ports. However, from a point of view of performance this is not ideal and the coming chapters will show how mechanisms can be implemented to support this kind of communication outside the DEVS specification.

As a consequence on can argue the need to extend the DEVS specification with descriptions of data bindings between components that is not based on subscription in order to keep intact a clear link between a functional specification in DEVS and the corresponding implementation.

Compositionality of DEVS Components
The behaviour of a component in DEVS, i.e. the ability to schedule internal dynamics in the component and react to external events is very similar to the way control software is typically implemented on an embedded computer: as a periodic thread and associated interrupt handler that handle external inputs (e.g.

samples from the analogue-to-digital-converter or push buttons), which can reschedule the periodic process if e.g. a change of controller parameters has been ordered.

Therefore, in our view, it is a strength of DEVS that it captures this behaviour in a model that is not bound to an implementation. The `Runner` mechanism in the implementation of DEVS is here very important, because the model can be composed with different runners.

For example if the model is of a specific controller for a given problem the DEVS implementation can be composed with a runner object that interfaces it to a simulation environment for a first evaluation and then later the exact same controller implementation can be composed with another runner object that interfaces to the system calls of the Real Time Operating System on the target control computer.

This philosophy has much in common with how software is developed for business applications; here software components are encapsulated using technologies such as for example *Enterprise Java Beans* [Sun_Microsystems, 2007a], whereafter it can be deployed in different contexts without changing the component itself. In this work we pursue this approach of encapsulation and composition for control systems software, as described in the research objectives defined in Subsection 1.1.3 on page 6.

2.5 Chapter Summary

This chapter introduced the DEVS formalism, which will be used throughout the dissertation to implement the methods and algorithms to be developed. In summary; a DEVS model is made up of atomic and coupled models contained in a top-level coupled model that is executed by a runner object.

A software framework was developed in Java which implements the specified DEVS capabilities and an overview of this work was given. Further, issues concerning the communication model used in DEVS were discussed.

The merit of the DEVS approach as a platform for implementing control systems software is the encapsulation and compositionality that it can provide for software components, such that they can be reused in various contexts with no changes.

3 Quantised State Systems

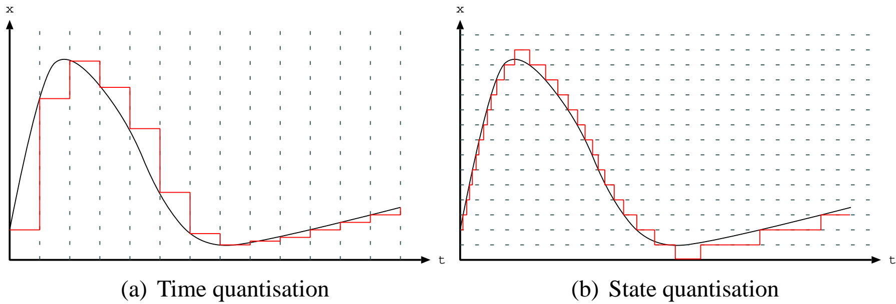
This chapter describes Quantised State Systems (QSS) and specifically the QSS2 algorithm for propagating ordinary differential equations. Mathematical properties are discussed and illustrative examples are given. Finally, a comprehensive simulation study of an autonomous underwater vehicle is given to demonstrate the benefits over traditional methods.

3.1 Motivation for Quantised State Systems

Complex autonomous systems often require that models are propagated on-line for use in state estimation (e.g. extended Kalman filtering [Grewal and Andrews, 1993]), feed-forward control (e.g. Model Predictive Control [Alamir, 2006]), as well as other usages. Typically, numerical integration techniques used for on-line model propagation are based on quantisation of time in evenly distributed intervals. Examples include forward-Euler integration or Runge-Kutta integration, which are often applied for such applications.

The sample-based approach is challenged in complex non-linear systems where it can be difficult to select a reasonable sample-rate which is fast enough for worst case conditions and at the same time computationally efficient under nominal circumstances. In networked systems information may be received over a network arriving with non-uniform intervals making it difficult to incorporate using a sample based approach.

An alternative approach is to consider quantisation of the value axis rather than the time axis, consider Figure 3.1. The (a) graph depicts a typical ideal time quantised solution (red line) that approximates a continuous trajectory ($\dot{x} = f(x),\ x \in \mathbb{R}$). The (b) graph shows an ideal state quantised solution where the value axis is quantised. It is noted that in the value quantised trajectory, discrete



(a) Time quantisation (b) State quantisation

Figure 3.1: Time and state quantisation. The black line is the continuous state evolution and the red line is the discretised state evolution.

value changes appear at a rate that is proportional to the level of change of the continuous trajectory, whereas the rate is constant for the time quantised approximation.

In other words; a state quantised approach allows the computational effort required to propagate the model to be adjusted to the current rate of change in the system. This overcomes the problem of selecting an appropriate sample rate for non-linear systems as discussed above.

A practical implementation of the scheme described above can be based on the forward-Euler method but with the modification that instead of calculating value increase over a sample period, the interval until the next quantised level is reached is calculated based on the chosen quantisation distance, ΔQ, and the current derivative, $\dot{x}$ (see also Figure 3.2):

$$\Delta t = \frac{\Delta Q}{|\dot{x}(t_k)|} \tag{3.1}$$

Then when the time interval elapses, the state is updated according to:

$$x(t_k + \Delta t) = x(t_k) + \mathrm{sgn}(\dot{x}(t_k))\Delta Q$$

whereupon the derivative is evaluated and the next interval is calculated using Equation (3.1). This scheme is easily formulated in the DEVS discrete event formulation as described in the previous chapter. Also, using the external event function $\delta_{ext}(\cdot)$, new information can be injected into the model at arbitrary times - e.g. information in arriving network packages.

This way of thinking and associated DEVS formulations for operational algorithms can be found in early work by Bernard P. Zeigler [Zeigler, 1976]. However,

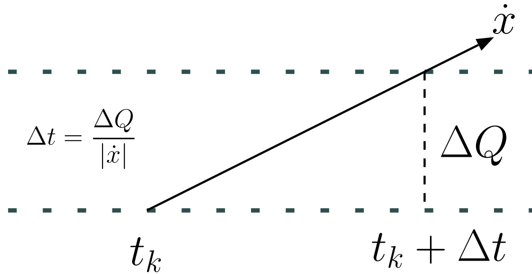


Figure 3.2: Illustration of the calculation of the time until the next event.

with these methods it was known that for some pathological examples the propagation of the system would require an infinite number of discrete transitions in a finite time interval - rendering the problem non-computable.

Recently Ernesto Kofman extended the algorithm to include a hysteresis level that ensures a minimum time between transitions [Kofman et al., 2001] and also developed a second order algorithm [Kofman et al., 2001] where the quantisation is related to the second derivative of the each state trajectory. The first and second order algorithms are called QSS1 and QSS2 respectively*. Related work on the development of multi-point integration schemes for quantised systems have been published [Nutaro, 2005], but will play no role in this dissertation.

The QSS2 algorithm forms the basis for most of the work in this dissertation and is described in Section 3.2. In Section 3.3 mathematical properties are discussed and some simple illustrative examples are given. Finally, in section 3.4 a more comprehensive simulation study is performed using a non-linear model of an autonomous underwater vehicle. To the author's knowledge this study is the first comprehensive study of the QSS2 algorithm applied to a high-dimensional non-linear system.

3.2 The QSS2 Method - First Order Quantisation

This section describes the QSS2 method for propagating ordinary differential equations and provides details on how this method is mapped to the DEVS specification introduced in the previous chapter. Consider a system specified by a gen-

*In this dissertation the abbreviation 'QSS' will be used to denoted quantised state methods in general

eral time-invariant non-linear ordinary differential equation with a time-invariant non-linear output map:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \tag{3.2}$$

$$\mathbf{y} = \mathbf{h}(\mathbf{x}) \tag{3.3}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the state vector of dimension n, $\mathbf{u} \in \mathbb{R}^m$ is the input vector of dimension m and $\mathbf{y} \in \mathbb{R}^p$ is the output vector of dimension p. $\mathbf{f}(\cdot)$ is a differentiable mapping $\mathbf{f} : \mathbf{x} \times \mathbf{u} \rightarrow \dot{\mathbf{x}}$ and $\mathbf{h}(\cdot)$ is likewise a mapping $\mathbf{h} : \mathbf{x} \rightarrow \mathbf{y}$.

Recall that any time-varying equation $\mathbf{f}(\mathbf{x}, \mathbf{u}, t)$ can be put on the form of Equation (3.2) by augmenting the system with an extra state, x_a, with derivative $\dot{x}_a = 1$, hence no loss of generality is inferred by the time invariance limitation of Equation (3.2).

The QSS2 algorithm integrates the state and produces the output $\mathbf{h}(\mathbf{x})$ by decoupling the system into event-communicating DEVS components representing functions and integrators, respectively - see Figure 3.3.

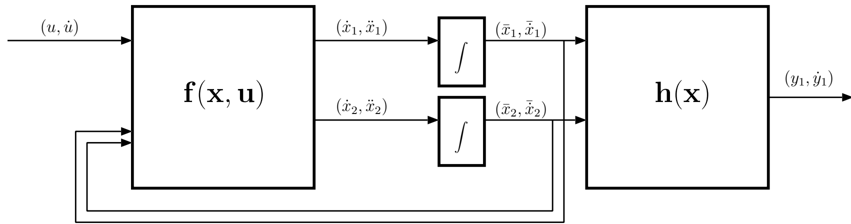


Figure 3.3: Structure of a QSS2 simulation. This figure is an example for a system with two states, one input and one output.

The objects that are communicated between the objects are ordered pairs $(a \in \mathbb{R},\ b \in \mathbb{R})$ representing a specific block output variable and its derivative. The following two subsections describe how the integrator and function blocks process information, respectively.

3.2.1 QSS2 Integrators

Each integrator block represents a single state x_i of the state vector $\mathbf{x}$ of the system as described by Equation (3.2). The integrator maintains a first order and second order internal model of this state respectively:

$$\bar{x}_i(t) = \bar{x}_i(t_i) + \bar{\dot{x}}_i(t - t_i) \tag{3.4}$$

$$x_i(t) \quad = \quad x_i(t_e) + \dot{x}_i(t - t_e) + \frac{1}{2}\ddot{x}_i(t - t_e)^2 \tag{3.5}$$

where t_i is the time of the last DEVS internal event of the integrator block[†] and t_e is the time of the last DEVS external event, i.e. new derivatives received from the function block.

When a new external event is received then $x_i(t_e)$ is reset to $x_i(t)$ - meaning that the second order model is a piecewise parabolic trajectory. The condition for updating the first order model, i.e. Equation (3.4), is given by a quantum separation principle between the two models:

$$|\bar{x}_i(t) - x_i(t)| > \Delta Q \tag{3.6}$$

where ΔQ is the chosen quantisation. When this expression becomes true, the first order model is reset to the value of the second order model, i.e.:

$$\bar{x}_i(t) = x_i(t) \;\; \wedge \;\; \bar{\dot{x}}_i(t) = \dot{x}_i(t) \tag{3.7}$$

Equation (3.6) can be seen to be related to the curvature of the second order model, which equals the second state derivative. The above scheme is sketched in Figure 3.4, which provides an example trajectory. Here, it can be seen how the models for $\bar{x}_i(t)$ and $x_i(t)$ are allowed to evolve independently whereafter upon reaching the difference ΔQ are reset to the same condition.

With this formulation one can think of $(\bar{\mathbf{x}}, \bar{\dot{\mathbf{x}}})$ as an operating point, or rather an *operating trajectory*, with the guarantee that it is correct to within the chosen quantum within the time interval until the next event.

At any time the time until the next internal event of the integrator can be found by solving Equation (3.6), which is the result returned by the time advance function ($ta(\cdot)$) of the integrator. It is worth to note that near linear trajectories result in very few events, while the rate of events increase in proportion to the magnitude of the second derivative of the state trajectory.

3.2.2 QSS2 Function Maps

In the following we will discuss $\mathbf{f}(\mathbf{x}, \mathbf{u})$, but the discussion is the same for $\mathbf{h}(\mathbf{x})$ as well. The block representing $\mathbf{f}(\mathbf{x}, \mathbf{u})$ receives working state and input trajectories

[†]corresponding to the last time the block produced an output

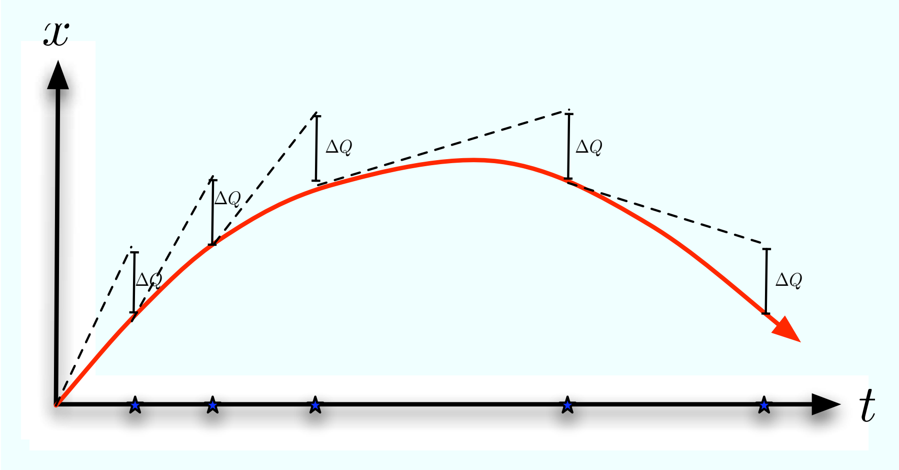


Figure 3.4: Sample QSS2 trajectory; showing when internal events occur (blue stars) and linear trajectories of the quantised states between events (dashed lines).

and is tasked with producing first and second state derivatives. For the function block there is no difference if the block inputs are from external inputs or from states, therefore with no loss of generality we consider the input vector:

$$\mathbf{z} = [\mathbf{x}^T \ \mathbf{u}^T]^T = [z_1 \ \dots \ z_{n+m}]^T$$

and input derivative vector:

$$\dot{\mathbf{z}} = [\dot{\mathbf{x}}^T \ \dot{\mathbf{u}}^T]^T = [\dot{z}_1 \ \dots \ \dot{z}_{n+m}]^T$$

Further, we consider the vector valued function $\mathbf{f}$ as an ordered set of scalar functions, i.e.

$$\mathbf{f} = (f_1, \dots, f_n)$$

We let a matrix $\mathbf{D} \in \mathbb{R}^{n \times (n+m)}$ describe connections from inputs $(z_j : j < n + m)$ to outputs of $\mathbf{f}$ such that z_j selects a column in $\mathbf{D}$ which has entries, $d_{i,j}$, which are one if the corresponding output, i, depends on the input z_j and zero if there is no dependence. Finally, we define the Jacobian matrix of $\mathbf{f}$:

$$\mathbf{J_f} = \begin{bmatrix} \frac{\partial f_1(\mathbf{z})}{\partial z_1} & \dots & \frac{\partial f_1(\mathbf{z})}{\partial z_{n+m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n(\mathbf{z})}{\partial z_1} & \dots & \frac{\partial f_n(\mathbf{z})}{\partial z_{n+m}} \end{bmatrix} \tag{3.6}$$

The total internal state of the function block is made up of these quantities, e.g. $\mathbf{z}$, $\dot{\mathbf{z}}$, $\mathbf{D}$ and $\mathbf{J_f}$. The following will describe how input events are processed and how outputs are generated, respectively.

Input Processing

When an external event occurs the function block first updates the internal state according to:

$$\mathbf{z}(t) = \mathbf{z}(t_k) + \dot{\mathbf{z}} \cdot (t - t_k) \tag{3.6}$$

where t is the current time and t_k is the time of the last event. Hereafter the input message set $\mathcal{M}$ (see Section 2.2.1 on page 21) containing new inputs $\{(z_j^*,\ \dot{z}_j^*) : j \in \mathcal{X}\}$ are processed in the following manner one by one; first the derivative information is copied to the internal state:

$$\dot{z}_j = \dot{z}_j^*$$

Hereafter the j'th column in $\mathbf{D}$ is selected and for each non-zero element, $d_{i,j}$, corresponding terms in $\mathbf{J_f}$ are calculated using the numerical difference:

$$\frac{\partial f_i}{\partial z_j} = \frac{f_i(\mathbf{z}) - f_i(\mathbf{z}^*)}{z_j - z_j^*} \quad \forall \{i : d_{i,j} \neq 0\} \tag{3.5}$$

where $\mathbf{z}^*$ is the original $\mathbf{z}$ vector but with the j'th element replaced by the new information, i.e. $z_j = z_j^*$. It is important to realize that the above difference calculation, due to the integrator quantisation, is always performed with the denominator satisfying:

$$|z_j - z_j^*| = \Delta Q_j$$

meaning that the accuracy of the calculation of the Jacobian term, Equation (3.2.2), is controlled by the choice of the quantum.

Output Calculation

The function block is memoryless in the sense that as soon as new information is received then outputs must be produced. This means that following an external event we have $ta(\cdot) = 0$, meaning that an output is produced immediately. Consider the first order Taylor expansion of $\mathbf{f}(\mathbf{z})$:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{z}(t_k)) + \mathbf{J_f}(t_k)\dot{\mathbf{z}}(t - t_k)$$

where t_k is the time of the just processed external event. From this expansion we identify that:

$$\dot{\mathbf{x}} \;=\; \mathbf{f}(\mathbf{z}(t_k))$$

$$\ddot{\mathbf{x}} \quad = \quad \mathbf{J_f}(t_k)\dot{\mathbf{z}}(t_k)$$

hence output pairs $(\dot{x},\ \ddot{x})$ can be evaluated for any output index, i, as:

$$\left(f_i(\mathbf{z}(t_k)),\ \sum_{j=1}^{(n+m)} \frac{\partial f_i(\mathbf{z}(t_k))}{\partial z_j} \dot{\mathbf{z}}_j(t_k) \right) \tag{3.1}$$

in other words; the second derivative is the directional derivative of f_i. Outputs are only produced for outputs that depended on the inputs received in the last external event, i.e. indexes $j \in \mathcal{M}$. This entails the index-set:

$$i \in \bigcup_{j \in \mathcal{M}} \{s : d_{s,j} \neq 0 \vee 1 \leq s \leq n\} \tag{3.1}$$

The fact that QSS2 produces new outputs based on coupling information of the system means that the method is able to utilise sparsity in the system and it therefore reduces the number of required function evaluations.

3.2.3 Software Implementation

This subsection provides an overview of how the QSS2 algorithm has been implemented in the software framework described in Section 2.3 on page 24. Algorithm 3.1 shows how the QSS2 integrator functionality described in Subsection 3.2.1 maps the DEVS specification of functions. Likewise for Subsection 3.2.2 describing the functionality of QSS2 map blocks; Algorithm 3.2 describes how it is implemented in the DEVS specification.

Figure 3.5 is a class diagram for the QSS2 package implemented to be used with the DEVS package described in Section 2.3 on page 24. Central to the implementation is the `Qss2Integrator` and `Qss2Map` which implement the QSS2 integrator and function respectively, they are both concrete implementations of the abstract `DevsAtomic`.

Communication between the integrators and functions are delegated to the `Qss2Port` class which is a representation of the pairs consisting of a value and its derivative. The `Qss2Map` class delegates responsibility for representing a set of equations through the `EquationSet` interface.

The `RpeEquationSet` is a concrete implementation of `EquationSet` which use the external library "Java Equation Parser 2.4 (JEP)[‡]", which manages

[‡] Available from http:///www.singularsys.com/jep/

```
δ_int():
integrate Equation (3.5) to current time
x̄ ← x, x̄̇ ← ẋ

δ_ext(P):  P is a set of messages
integrate Equation (3.5) to current time
update ẋ and ẍ with new values from P

λ():
return (x̄, x̄̇)

ta():
return solution to Equation (3.6)
```

Algorithm 3.1: DEVS implementation of a QSS2 integrator

```
δ_int():
return

δ_ext(P):  P is a set of messages
update internal state, Equation (3.2.2)
select row in D and calculate Jacobian terms, Equation (3.2.2)

λ():
foreach index, i, in the set given by Equation (3.2.2):
·  calculate outputs, Equation (3.2.2)

ta():
return ∞
```

Algorithm 3.2: DEVS implementation of QSS2 function map

parsing and evaluation of equations based on a representation as text strings. JEP also allows custom functions and constants to be utilised. An algorithm has also been developed which analyses the equations, represented by string input from the user, and derives the coupling matrix $\mathbf{D}$.

The delegation of equation representation through the interface allows a great deal of flexibility; for example one might use the `RpeEquationSet` implementation throughout the whole analysis and design phase of a project and then for implementation on an embedded processor the equations can be hard-coded as (Java) functions in order to minimise processing time requirements.

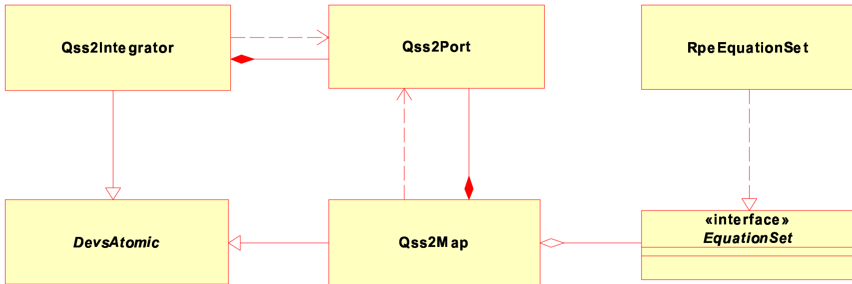


Figure 3.5: Classes that implement the QSS2 approach and their inter-dependencies.

3.3 Properties and Benefits of QSS2

This section provides an overview of the mathematical properties of the QSS2 algorithm and discusses how these properties match up with the properties of other integration methods. Hereafter some benefits specifically related to the work in this dissertation are summarised. Finally, a few illustrative examples are presented to demonstrate the points.

3.3.1 Mathematical Properties

Important properties of the QSS2 algorithm are derived in [Kofman, 2003] based on a perturbation analysis approach. The following gives a brief overview of the main idea in the analysis and the results. It is noted that by construction of the QSS2 algorithm the difference between the quantised and unquantised state:

$$\mathbf{\Delta x} = \mathbf{x} - \bar{\mathbf{x}}$$

is bound by the choice of quantisation:

$$|\mathbf{\Delta x}| \preceq \mathbf{\Delta Q}$$

therefore properties relating to the QSS2-simulated system can be analysed by analysing the perturbed dynamical system:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x} + \mathbf{\Delta x}, \mathbf{u})$$

Based on this approach, the following properties have been derived in [Kofman, 2003]:

1. QSS based solutions approach analytical solutions exactly as the quantisation approach zero (Theorem 4.1 in [Kofman, 2003]).
2. Stable equilibrium points in the system $\mathbf{f}(\mathbf{x}, \mathbf{u})$ are preserved in the quantised system (Lemma 4.1 in [Kofman, 2003]).
3. It is always possible to find a sufficiently small quantisation ΔQ, such that states in the quantised systems converge to a small region centred at the corresponding stable equilibrium point of the continuous system (Theorem 4.2 [Kofman, 2003]).

To summarise; QSS solutions are qualitatively consistent with analytical solutions in terms of equilibrium points, but solutions converge not to equilibrium points exactly, but instead to regions near those points. It is always possible to find a sufficiently small quantisation that the quantisation effects are negliable for any practical problem.

From a functional level the use of QSS methods is analogous to ordinary time-discrete methods such as forward-Euler integration or integration using the Runge-Kutta algorithm; if we choose a reasonably small quantisation of time then the algorithm provides a simulated trajectory that is sufficiently consistent with the analytical solution.

As is often the case with analysis of numerical algorithms [Ascher and Petzold, 1998] then when considering Linear Time Invariant (LTI) systems the extra structure can be used to derive useful properties that to some extend also can describe how the algorithm copes with non-linear systems. By analysing LTI systems it has been found [Kofman, 2003] that the QSS2 algorithm propagates LTI systems with a global error bound that can be derived from the eigenvalues.

This is a stronger guarantee than what is provided by forward-Euler integration and Runge-Kutta integration; in general this kind of guarantee is only provided by implicit methods for numerical integration. Implicit methods are not suited for on-line implementation due to their non-causality, however. Also QSS2 copes consistently with stiff systems, which are typically challenging for explicit time discretised methods like forward-Euler or Runge-Kutta [Ascher and Petzold, 1998].

In summary; QSS2 is a method for numerical integration that has some advantages over explicit methods, which are usually only found in non-causal implicit methods, but introduces a residual perturbation in the solution trajectory.

3.3.2 Discussion of the QSS2 Algorithm

In this dissertation the QSS2 algorithm will be utilised heavily, and we shall briefly list some of the benefits of QSS2 that will be exploited in the work during the remainder of the dissertation:

- The use of the coupling matrix $\mathbf{D}$ allows for exploitation of inherent sparsity in systems with loosely coupled subsystems.
- The Jacobian $\mathbf{J_f}$ is maintained throughout the propagation and can be exploited for control and estimation purposes.
- The event formalism inherently supports changes at arbitrary times. This will be utilised in connection with hybrid systems, which is the subject for Part III of this dissertation.

The main drawback is that budgetting of computer time is difficult with QSS2, whereas required computer time is usually deterministic when using fixed step algorithms. Secondly, it should be mentioned that the complete state of a QSS2 simulation consists not only of the state values, but also their derivatives and event times, whereas the state only consist of the state values in a time discrete method. This means that QSS2 state representation requires more storage and are more computationally expensive to make copies off.

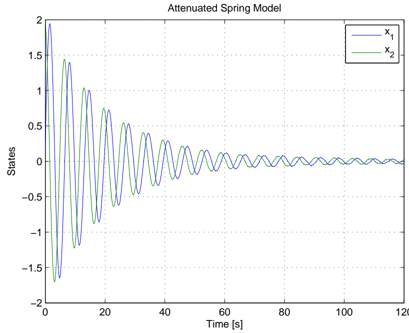
3.3.3 Illustrative Examples

This subsection provides a few examples of simple systems propagated using QSS2 in order to discuss some of the phenomena described above. The next section will provide a comprehensive case study of a complex system. Consider the following system, a lightly damped oscillator:

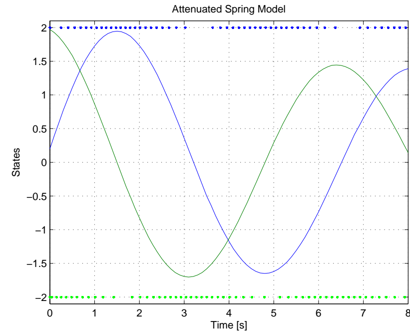
$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -0.95x_1 - 0.1x_2 \end{aligned}$$

This system has been simulated with the QSS2 method with quanta of $\Delta Q = 0.01$ for both states. Results are shown on Figure 3.6 for a simulation with initial conditions, $x_1 = 2$ and $x_2 = 0$.

The (a) graph shows initially an exponentially decaying envelope for the system, as expected. However, at the end of the simulation it is clear that the residual



(a) Trajectories over time



(b) First seconds with marked integrator events

Figure 3.6: Lightly damped oscillator showing decaying oscillations.

oscillation is not decaying any further, as predicted in the last section. The residual oscillations can, however, be made arbitrarily small by reducing the quantum.

The (b) graph is a zoom of the initial 8 seconds where the blue/green diamonds indicate integrator outputs of the corresponding integrator. i.e. when new output trajectories are generated, cf. Equation (3.7). It can be seen that events occur frequently where each state trajectory is most curved and that when it is near to a straight line then there are only a small number of events.

As a second example; consider the following non-linear system, which is characterised by a change of time-scale from the initial response to the prevailing response:

$$\begin{aligned} \dot{x}_1 &= -100(x_1 + \sin\ x_2) \\ \dot{x}_2 &= 1 \end{aligned}$$

This system has been simulated with initial conditions: $\mathbf{x} = [1\ 0]$ using QSS2 with quanta $\Delta Q_1 = 0.001$ and $\Delta Q_2 = 0.1$. The result for x_1 can be seen on Figure 3.7.

The figure shows the simulated trajectory of x_1 as the blue line and the integrator output event times as green diamonds. It is clear from the figure how the QSS2 algorithms adapts to the changing dynamics by scheduling more integrator events when the trajectory has a high curvature, while the number of integrations decrease as the trajectory resembles a straight line.

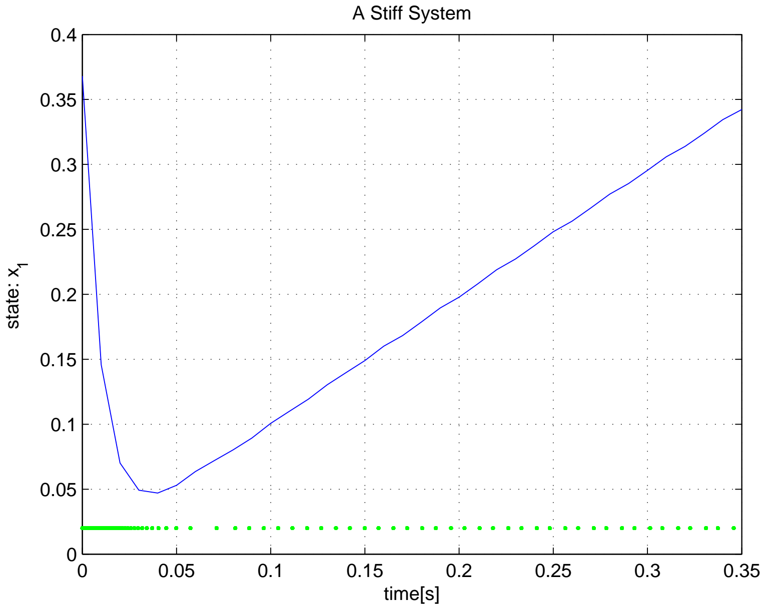


Figure 3.7: Example of a system with different response time-scales and the resulting internal integrator events.

3.4 Simulation of an Autonomous Underwater Vehicle

This section describes the application of the QSS2 algorithm to simulation of an Autonomous Underwater Vehicle (AUV). The purpose of this case study is threefold:

- To demonstrate the QSS2 algorithm on a complex non-linear dynamical system
- To evaluate the consistency and accuracy of QSS2 simulation compared to an explicit method
- To evaluate the execution speed of QSS2 relative to an explicit integration method at the same level of accuracy

To facilitate the comparison between QSS2 and an explicit method on equal terms the forward-Euler integration method has been implemented in the DEVS framework. The QSS2 simulation will be performed using the software described in Subsection 3.2.3. The forward-Euler integration approach was chosen due to

its simplicity which facilitated quick implementation in DEVS/Java.

The AUV model which will be used for the study is known as the Naval Postgraduate School Autonomous Underwater Vehicle (NPSAUV) and is thoroughly described in [Healey and Lienard, 1993]. The following will at first provide an overview of the model of the NPSAUV, whereafter evaluation results for three simulation cases are given.

3.4.1 Model Description

Figure 3.8 depicts the NPSAUV. The craft is $5.3m$ long and has a mass of $5.4T$. The model is described on basis of [Healey and Lienard, 1993], but is formulated in the standard marine notation of [Fossen, 2002a].

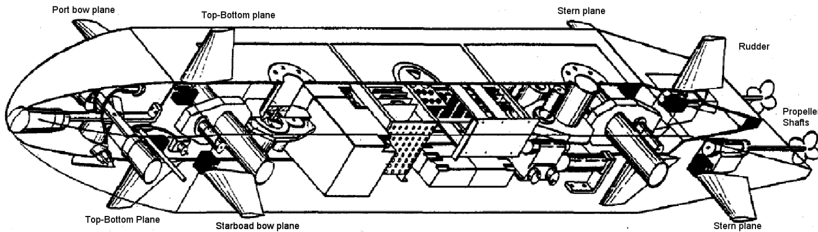


Figure 3.8: A sketch of the Naval Postgraduate School Autonomous Underwater Vehicle [Healey and Lienard, 1993].

The vector of controllable inputs to the system is given by:

$$\mathbf{u} = [\delta_r \ \ \delta_{bs} \ \ \delta_{bp} \ \ \delta_s \ \ \delta_{tb} \ \ n]^T$$

where the different actuators are: δ_r - Rudder, δ_{bs} - Starboard bow plane, δ_{bp} - Port bow plane, δ_s - Stern plane, δ_{tb} - Top-Bottom plane, and n - propeller speed. The rudder and control planes all saturate at $\pm 20^o$ and the propeller can run at between 0 and 1500 RPM.

The model contains 12 states, six of which are described in a body-fixed coordinate system and six described in an assumed inertial North-East-Down frame (NED). The body fixed states are:

$$\mathcal{V} = [u \ \ v \ \ w \ \ p \ \ q \ \ r]^T$$

which respectively represent: surge speed, sway speed, heave speed, roll rate, pitch rate, and yaw rate. The state variables in the NED frame are:

$$\eta = [x \ \ y \ \ z \ \ \phi \ \ \theta \ \ \psi]^T$$

which respectively represent: x-, y-, and z-position, roll-, pitch-, and yaw-angle. The dynamical/kinematical model is of the form:

$$\begin{aligned} \mathbf{M}(\mathcal{V})\dot{\mathcal{V}} + \mathbf{C}(\mathcal{V})\mathcal{V} + \mathbf{D}(\mathcal{V})\mathcal{V} + \mathbf{g}(\eta) &= \tau(\mathbf{u}, \mathcal{V}, \eta) \\ \dot{\eta} &= \mathbf{J}(\eta)\mathcal{V} \end{aligned}$$

where the terms describe:

$\mathbf{M}(\mathcal{V})\dot{\mathcal{V}}$: Rigid body mass and added mass due to hydrodynamics

$\mathbf{C}(\mathcal{V})\mathcal{V}$: Coriolis and centripetal forces and torques including added mass effects

$\mathbf{D}(\mathcal{V})\mathcal{V}$: Hydrodynamic dampening forces and torques

$\tau(\mathbf{u}, \mathcal{V})$: Propulsion forces and torques

$\mathbf{g}(\eta)$: Gravitational and buoyancy forces and torques

$\mathbf{J}(\eta)\mathcal{V}$: Transformation between body and NED frame

A Matlab implementation of the model can be found as part of the "Marine Control Toolbox (MCC)" [Fossen, 2002b]. This implementation has been used as reference for the implementation for the QSS2 algorithm. The complete model contains around 120 constants and consists of 30 non-linear equations and 4 integrals (cross-flow drag coefficients) to be solved numerically for each simulation step. The model has singular points in $\theta = \pm\pi$ due to Euler-angle formulation of kinematics and the thrust model is singular in $u = 0$.

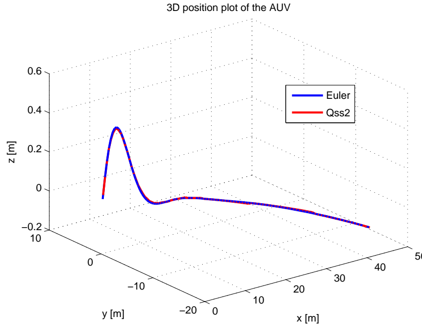
3.4.2 Evaluation Results

This subsection presents simulation results for three distinct cases. Case 1 will compare performance between QSS2 and forward-Euler integration. Case 2 and 3 will investigate the capabilities of QSS2 to adapt to varying initial conditions.

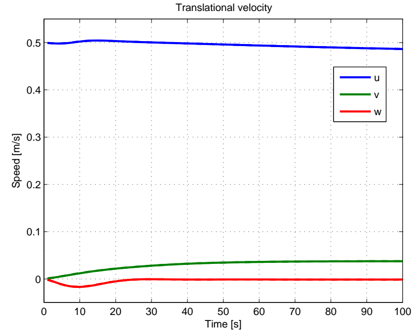
Case 1 - Slow Turn

In this case we compare the simulated trajectory generated by forward-Euler and QSS2 integration respectively. During the simulated 100 s the AUV will maintain a near constant surge speed and perform a slow turn. The initial conditions are a

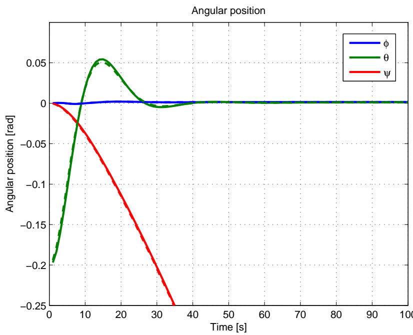
surge speed of $u = 0.5$ m/s and an initial pitch angle of $\theta = -0.2$ rad, all other states are initially zero. The rudder is set to a constant deflection of $\delta_r = 0.1$ rad and the propeller shaft is set to $n = 400$ RPM. Results can be seen in Figure 3.9.



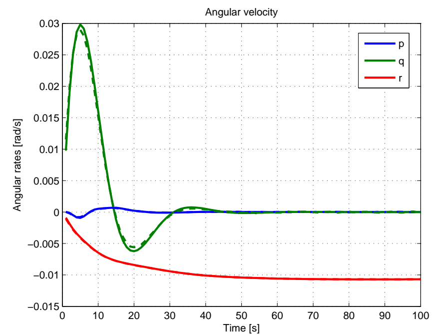
(a) Translational position



(b) Translational velocity



(c) Angular position



(d) Angular velocity

Figure 3.9: A slow turn. Except for (a) dashed lines indicate QSS2 trajectories and full lines are forward-Euler trajectories. It can be seen that the two simulated trajectories are equivalent.

The forward Euler integration is performed with a time-step of 0.2 s, which was found by decreasing the step-size until further decrease gave no visible improvement in the simulated trajectory. The quanta for the QSS2 simulation were chosen such that the solution trajectory is as near to the forward-Euler solution as possible. The quanta can be seen on Table 3.1.

The results on Figure 3.9 show that the two trajectories are almost identical. The endpoint difference of the translational position is 14.3 cm. The main differ-

State	u, v, w	p, q, r	x, y, z	ϕ	θ, ψ
Quantum (ΔQ)	10^{-5} m/s	10^{-6} rad/s	$5 \cdot 10^{-4}$ m	10^{-6} rad	10^{-5} rad

Table 3.1: Quanta selection for QSS2 simulation

ence, as is evident from the graph of the angular velocity (d) and the graph of the angular position (c) is that the two methods handle oscillatory behaviour differently; the forward-Euler method in general generates a trajectory that reaches a larger amplitude than the QSS2 method does.

Execution time[§] for forward-Euler is 0.23 s and for QSS2 it is 0.17 s. QSS2 therefore performs approximately 35% better than forward-Euler integration in this case. During the simulation the forward-Euler calculates 500 sample points. The QSS2 methods generates a number of output trajectories for each state which can be seen on Table 3.2.

State	u	v	w	p	q	r	x	y	z	ϕ	θ	ψ
# Outputs	23	24	37	94	70	49	43	59	27	88	117	17

Table 3.2: Number of outputs for each state during QSS2 simulation

In total this amounts to 648 output events for QSS2, which is higher than the 500 samples for the forward Euler method, but each QSS2 event only requires recomputation of a subset of rows in $\mathbf{f}(\cdot)$, whereas forward-Euler integration requires all rows to be calculated at each sample-time, hence the reduced execution time.

Case 2 - QSS2 Adaptability

As described previously, in Subsection 3.2.1, QSS2 schedules integrator updates according to the level of deviation from linearity of the individual state trajectories. If another simulation is performed where the rudder input and engine RPM input is set to zero and the decay from initial conditions is simulated then QSS2 executes in 0.11 s, while forward-Euler still takes 0.23 s.

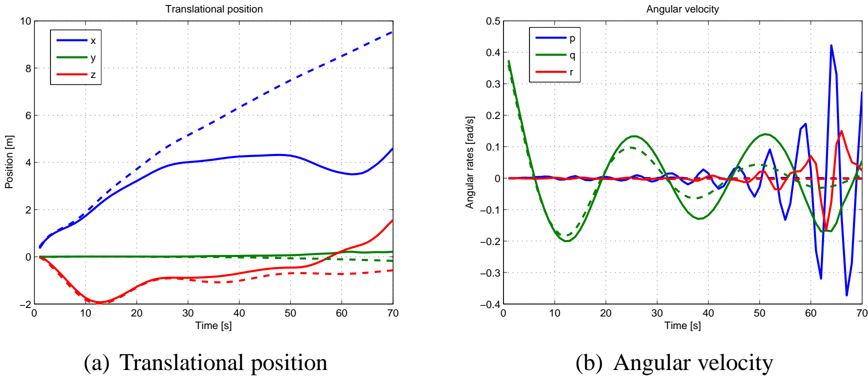
Case 3 - QSS2 Robustness

When one selects a time-step for a numerical integration method applied on a non-linear system one must make some assumptions on the possible initial states. If these assumptions do not hold, the integration may experience numerical prob-

[§]on a contemporary lap-top computer, 1.6GHz

lems.

Consider Case 1 presented above; we change the engine speed to $n = 0$ RPM (i.e. no energy is added to the system) and set the initial pitch rate to $q = 0.45$ rad/s. The outcome of this simulation is presented in Figure 3.10.



(a) Translational position (b) Angular velocity

Figure 3.10: Non-nominal initial conditions; forward-Euler (full lines) integration is inconsistent, while QSS2 remains (dashed lines) consistent

It is clear from the (a) graph that the QSS2 and forward-Euler trajectories diverge. Looking at the (b) graph it can be seen that after 10 s the forward-Euler solution begins to develop oscillations with increasing amplitude for the roll and yaw state - eventually leading to numerical instability for the forward-Euler solution. Meanwhile the QSS2 method performs consistently although the number of integrator outputs have increased to 2063 to cope with the increased dynamics of the system, which also causes an increase in execution time to 0.4 s.

3.4.3 Discussion of Simulation Study Results

The results presented above demonstrates that the QSS2 method competes well against forward-Euler integration on the AUV problem. The forward-Euler integration approach was chosen due to its simplicity which facilitated quick implementation in DEVS. Algorithms such as e.g. Runge-Kutta can also be implemented in DEVS, but would require more effort.

Comparison between Matlab based Runge-Kutta integration based on the model in the "Marine Control Toolbox (MCC)" [Fossen, 2002b] shows similar performance between the two, but it is hard to tell if the performance is due to the

method or the time it takes for function evaluation in the two execution environments (Matlab vs. DEVS/Java). In these simulations the Runge-Kutta algorithm requires a step-size of 0.5 s.

An important point is the robustness properties demonstrated in Case 3; all traditional explicit time-discrete integration methods have a limited stability region [Ascher and Petzold, 1998] meaning that one can imagine pole-locations where not all poles can be inside the stability region. This is the case for the forward-Euler algorithm in Case 3, where it cannot cope with the fast roll and yaw dynamics at the same time as the relatively slow pitch dynamics. Since the quantised solution is able to handle each state individually, it can adapt to the situation. This property is desirable when implementing robust control systems.

Applying a code profiler in Case 1; it was measured that 16% of the simulation time is overhead in the sense that it is spent by the DEVS framework, i.e. the software classes described in Section 2.3 on page 24, 2% of the time is spent in the `Qss2Integrator` class and finally 82% of the time is spent evaluating $\mathbf{f}(\cdot)$ in the `Qss2Map` class.

3.5 Chapter Summary

Initially it was argued that numerical simulation algorithms are an important part of many advanced control and estimation approaches, and an alternative to well known discrete-time methods was presented which relies on quantisation of the states. The merit of this approach is automatic adjustment of the required number of integration steps to the level of change experienced by the solution trajectory - and further a decoupling of states in the calculations that can exploit sparsity.

A specific QSS based algorithm, the QSS2 algorithm, was presented in detail. The algorithm relies on a first order quantisation of the state and maintains internally Jacobian matrices for the system being propagated.

The properties of the QSS2 algorithm were discussed and demonstrated through both simple illustrative examples and a more intensive simulation study of an autonomous underwater vehicle. It was demonstrated that the QSS2 algorithm has performance and robustness features that makes it interesting for use in control applications. To the author's best knowledge the simulation study currently is the most comprehensive study of a higher dimensional non-linear system being simulated with the QSS2 algorithm.

Part II

Estimation and Control using Quantised State Systems

4 Kalman Filter Estimation in QSS

This chapter introduces the well-known Extended Kalman Filter algorithm for quantised state systems within the DEVS framework. Hereby, the algorithm can provide Jacobian free estimation using the partial derivative matrices generated by the QSS2 algorithm. The new and original algorithm are compared on an attitude determination example.

4.1 Introduction

Kalman filtering and Extended Kalman Filtering (EKF) are arguably the most widely applied methods for state-observation in linear and non-linear systems respectively. This chapter contributes with a formulation and implementation of the algorithm for quantised state-system, which provides additional benefits over traditional EKF implementations by being asynchronous in nature and by providing Jacobian-free estimation using Jacobian estimates generated by the QSS2 algorithm when propagating the state.

Section 4.2 reviews the classic EKF algorithm for non-linear systems, whereas Section 4.3 describes how the algorithm has been implemented for quantised state system models. Finally, Section 4.4 provides a comparative case-study of attitude determination for a deep space probe. The QSS/EKF filter development and a similar simulation study as presented here is published in [Alminde et al., 2007a].

4.2 Review of Extended Kalman Filtering

The EKF algorithm estimates the mean value of the system states and associated covariance matrix. The state estimate and covariance matrix is propagated in intervals where no measurements are available and when a measurement is available

the algorithm calculates a state update and updated covariance matrix. Contrary to the linear Kalman filter the EKF uses small-signal models, around the current estimation for covariance propagation and Kalman gain calculation.

Under the assumption that process and measurement noises are independent multi-variate Gaussian distributions with zero mean, the EKF is optimal in the sense that it minimises the covariance of the prediction error. The following reviews the important equations for a typical time discrete implementation, based on [Grewal and Andrews, 1993]. The starting point is a continuous non-linear model:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) + \mathbf{w}(t) \quad \mathbf{w} \sim N(0, \mathbf{Q}(t)) \tag{4.1}$$

$$\mathbf{y}(t) = \mathbf{h}(\mathbf{x}, t) + \mathbf{v}(t) \quad \mathbf{v} \sim N(0, \mathbf{R}(t)) \tag{4.2}$$

where $\mathbf{w}(t) \in \mathbb{R}^n$ and $\mathbf{v}(t) \in \mathbb{R}^m$ represent process noise and measurement noise respectively, both of which are assumed to follow multivariate Gaussian distributions with zero mean and covariance parameters $\mathbf{Q}(t) \in \mathbb{R}^{n \times n}$ and $\mathbf{R}(t) \in \mathbb{R}^{n \times m}$, respectively. A covariance matrix $\mathbf{P}_{\mathbf{xx}}(t) > 0$ is associated with the process and describes the expected estimation error:

$$\mathbf{P}_{\mathbf{xx}}(t) = \mathbf{E}\left([\mathbf{x}(t) - \mathbf{E}(\mathbf{x}(t))] [\mathbf{x}(t) - \mathbf{E}(\mathbf{x}(t))]^T \right) \tag{4.3}$$

Between points in time where measurements are available the state estimate $\hat{\mathbf{x}}(t) = \mathbf{E}(\mathbf{x}(t))$ is propagated using Equation (4.1) (using some numerical integration algorithm) and the covariance matrix is propagated in discrete time (time step τ) using the relation:

$$\mathbf{P}_{\mathbf{xx}}(t_{k+1}) = \mathbf{\Phi}(t_k, \tau)\mathbf{P}_{\mathbf{xx}}(t_k)\mathbf{\Phi}^T(t_k, \tau) + \mathbf{Q}(t_k) \tag{4.4}$$

where $\mathbf{\Phi}(t_k, \tau) \in \mathbb{R}^{n \times n}$ is the state transition matrix of Equation (4.1) and is found from the Jacobian matrix of the system:

$$\mathbf{A_f}(\hat{\mathbf{x}}, \mathbf{u}, t) = \frac{\partial \mathbf{f}(\hat{\mathbf{x}}, \mathbf{u}, t)}{\partial \mathbf{x}^{\mathbf{T}}} \tag{4.5}$$

from which the state transition matrix for the interval of length τ can be found as:

$$\mathbf{\Phi}(t_k, \tau) = \mathbf{I} + \sum_{k=1}^{\infty} \frac{\mathbf{A_f}^k(\hat{\mathbf{x}}, \mathbf{u}, t)\tau^k}{k!} \tag{4.6}$$

which is usually evaluated to only the first few powers in k. The discrete time process noise, required for evaluating Equation (4.4), over the interval between two samples t_k and t_{k+1} is calculated from:

$$\mathbf{Q}(t_{k+1}) = \int_{t_k}^{t_k+1} \mathbf{\Phi}(t_k, \tau)\mathbf{Q}(t_k)\mathbf{\Phi}^T(t_k, \tau)\, d\tau \tag{4.7}$$

When a new measurement $\mathbf{y}(t_k)$ is available at some discrete time, the state estimate is updated according to:

$$\hat{\mathbf{x}}^+(t_k) = \hat{\mathbf{x}}(t_k) + \mathbf{K}(t_k)\left(\mathbf{y}(t_k) - \mathbf{g}(\hat{\mathbf{x}}(t_k))\right) \tag{4.8}$$

where superscript "+" indicates the value after the update is applied and $\mathbf{K}$ is the Kalman gain, which is calculated according to:

$$\mathbf{K}(t_k) = \mathbf{P_{xx}}(t_k)\mathbf{C_h}(\hat{\mathbf{x}}(t_k))\left(\mathbf{C_h}(\hat{\mathbf{x}}(t_k))\mathbf{P_{xx}}(t_k)\mathbf{C_h}^T(\hat{\mathbf{x}}(t_k)) + \mathbf{R}(t_k)\right)^{-1} \tag{4.9}$$

where the Jacobian of the output equation, $\mathbf{h}(\mathbf{x}, t)$, is:

$$\mathbf{C_h}(\hat{\mathbf{x}}, t) = \frac{\partial \mathbf{h}(\hat{\mathbf{x}}, t)}{\partial \mathbf{x^T}}$$

after the state correction, i.e. Equation (4.8), the covariance matrix is updated to represent the increased knowledge of the state inferred from the measurement:

$$\mathbf{P}^+_{\mathbf{xx}}(t_k) = \left[\mathbf{I} - \mathbf{K}(t_k)\mathbf{C_h}(\hat{\mathbf{x}}(t_k))\right]\mathbf{P_{xx}}(t_k) \tag{4.10}$$

4.2.1 EKF Temporal Flow

Typical implementations of the EKF algorithm assume a constant sample time with measurements arriving precisely at these sample times. The propagation of the state and the covariance in time is sketched on Figure 4.1.

As time progresses the covariance and state equations are propagated using Equations (4.1) and (4.4) respectively. At sample times new measurements are processed according to Equations (4.8), (4.9), and (4.10), i.e. a state correction is calculated and applied and the covariance is updated to reflect the increase in information about the state values.

In practical applications of the EKF algorithm the process noise, $\mathbf{Q}(t)$ is often used as a tuning parameter in order to ensure proper filter operation considering inaccuracies in state-propagation, inexact process knowledge, and inaccuracies due to the linearised models utilised in the filter [Zarchan and Musoff, 2000].

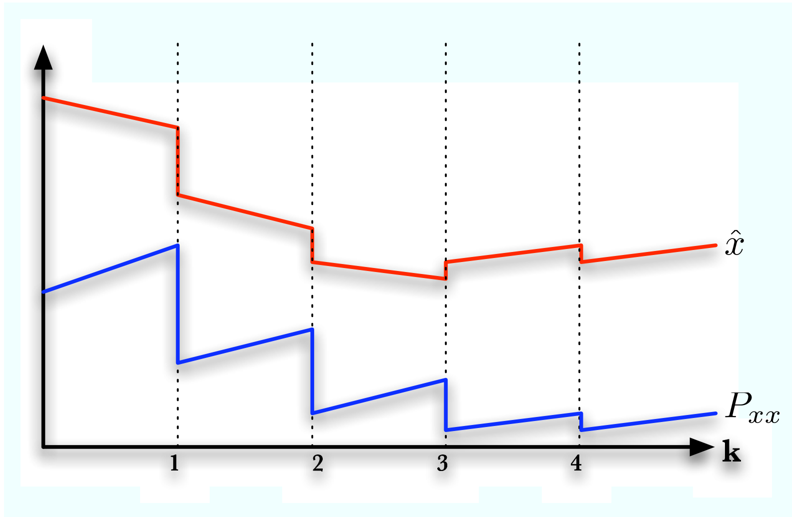


Figure 4.1: Propagation of state (red) and covariance (blue) in the EKF algorithm. New measurement arrive at sample times.

4.3 Extended Kalman Filtering in Quantised Systems

This section describes an implementation of the EKF algorithm for the quantised state systems framework introduced in the previous chapter, which utilises the Jacobian matrices as estimated as part of the QSS2 algorithm (see equation 3.2.2 on page 35). The benefits of this approach are:

- a Jacobian-free declarative estimator
- support for measurements arriving at arbitrary times

The first point is due to the QSS2 algorithm and the second point is due to the formulation as a discrete event system (i.e. using DEVS). There exist other Kalman filter formulations, for non-linear systems that do not require the availability of an expression for the Jacobian.

In [Schei, 1997] the Jacobian is estimated on-line using a central difference calculation for each element in the Jacobian matrix, where the points at which the function is evaluated are determined from the covariance. The method is shown to be marginally more accurate than the standard EKF algorithm.

The Unscented Kalman Filter (UKF) [Wan and Merwe, 2000] maintains an ensemble set of $2n + 1$ state vectors that are propagated through the non-linear process and measurement models in order to estimate the covariance - the UKF reaches a higher degree of accuracy than the EKF, because it effectively estimates the distribution up to the fourth moment. A similar approach is taken in [Quine, 2006] where the presented algorithm only propagates $n + 1$ state vectors and provides the same accuracy as the EKF.

The drawback of these methods is that they require a large number of function evaluations for each update step which for complex models can be computationally expensive. It will be shown in the following that using the QSS2 algorithm for state propagation and Jacobian estimation provides an efficient means to implement extended Kalman filtering, when it is intractable or inconvenient to derive an analytic expression for the Jacobian.

4.3.1 Data Flow and Block Diagram of the QSS/EKF Filter

As mentioned before, typical implementations of the EKF algorithm are sample-based, i.e. all discrete times, t_k, are spaced equally. However, this is not required; with proper time-keeping non-equal time-steps can be implemented using the equations presented in Subsection 4.2. The algorithm which will be described in the following implements the EKF equations with proper time-keeping in the DEVS specification on top of an existing QSS2 model. The result is a generic EKF block that can be used declaratively with any system described by its QSS2 model.

Consider Figure 4.2; here covariance propagation and measurement processing are decoupled. The covariance is propagated at a guaranteed maximum sample time (black dashed vertical lines) and when measurements arrive asynchronously (green dashed vertical lines). The state trajectory and associated output trajectory is propagated by the QSS2 algorithm.

The Quantised State Systems Extended Kalman Filter (QSS/EKF) is a DEVS block that can be added to an existing QSS2 simulation model. A block diagram with all elements is presented in Figure 4.3.

The structure of a normal QSS2 simulation (cf. Figure 3.3 on page 32 for comparison) is augmented with the QSS/EKF block which receive measurements, Jacobian estimates from QSS2 function blocks, state and output trajectories. The QSS/EKF block outputs state corrections and optionally the covariance matrix in

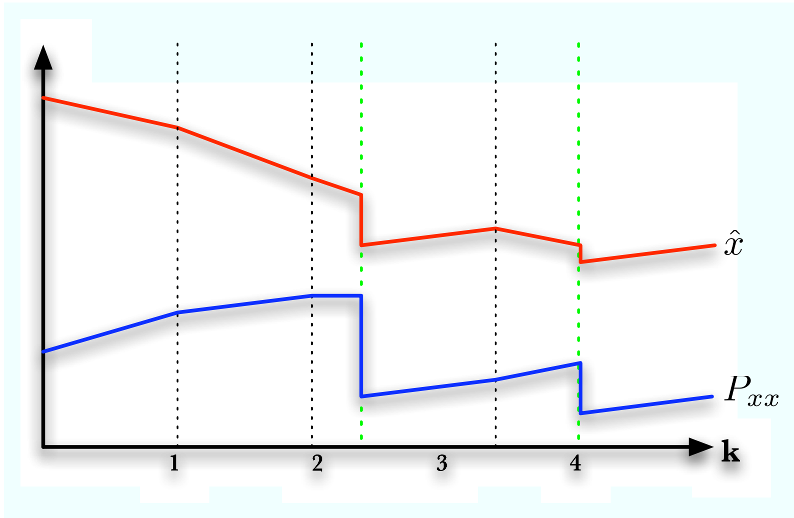


Figure 4.2: Propagation of state (red) and covariance (blue) in the asynchronous QSS/EKF algorithm. Green vertical lines are measurement events.

order to track filter performance.

Arrows originating in a small circles indicate data-flow that is not controlled by the DEVS specification, but through *call-backs* from the destination to the source of the arrow; this is due to the fact that the QSS/EKF block only needs information on the state and outputs (including the Jacobian of $\mathbf{h}(\mathbf{x})$) when new measurements are received, hence passing DEVS messages with this information on all state updates is redundant - see discussion in Section 2.4 on page 27.

4.3.2 State and Covariance Propagation

The state is propagated by the QSS2 based algorithm as presented in the last chapter. This is decoupled from the QSS/EKF block, except that it receives a new Jacobian estimate, $\mathbf{A_f}(\bar{\mathbf{x}}, \bar{\mathbf{u}})$, whenever the QSS2 algorithm updates one or more entries in this matrix. In other words; the QSS/EKF evaluates Equation (4.5) by setting it equal to the last received QSS2 estimate of the Jacobian:

$$\mathbf{A_f}(\hat{\mathbf{x}}, \mathbf{u}, t) = \frac{\partial \mathbf{f}(\hat{\mathbf{x}}, \mathbf{u}, t)}{\partial \mathbf{x}^\mathbf{T}} \simeq \mathbf{A_f}(\bar{\mathbf{x}}, \bar{\mathbf{u}})$$

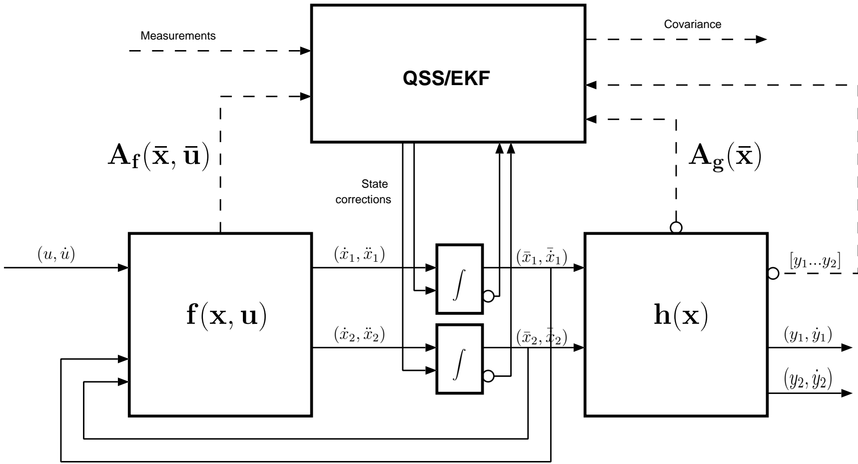


Figure 4.3: Block diagram for a QSS based EKF implementation. Dashed lines are vector/matrix signals. This example has two states, two outputs, and one input.

When the QSS/EKF block receives a new Jacobian matrix $\mathbf{A_f}(\bar{\mathbf{x}}, \bar{\mathbf{u}})$ it first calculates the state transition matrix, Equation (4.6) using the old Jacobian for the time interval from the last event until the current time and then solves Equation (4.7) and Equation (4.4) in order to propagate the covariance to the current time. In case Jacobian updates are rare, e.g. if the system is linear (only one initial update), a user specified maximum time between covariance propagation ensures proper numerical integration. After updating the covariance the new Jacobian is stored for use in the next propagation step.

4.3.3 Measurement Processing

When a new measurement is available a measurement update is performed, which entails:

1. The covariance is propagated to the current time, as in Subsection 4.3.2
2. The Kalman gain is calculated using Equation (4.9)
3. State correction is performed using Equation (4.8)
4. The covariance is updated due to the measurement, i.e. Equation (4.10)

Kalman gain calculation and state correction require information about the current state trajectory, current output trajectory, and the measurement Jacobian $\mathbf{A}_{\mathbf{h}}(\bar{\mathbf{x}})$ to be available. The QSS/EKF acquires this information using the call-back functions during processing of the external event function ($\delta_{ext}(\cdot)$) triggered by the measurement. State correction requires an additional input in each integrator block, which resets the states to the value estimated by the EKF.

Referring to Subsection 3.2.1 on page 32 which describes the QSS2 integrator block; when a reset is received $x(t_e)$ in Equation (3.5 on page 33), the state is reset to the value from the EKF, x^*. Hereafter the quantum criteria is applied (Equation (3.6 on page 33), restated here):

$$|\bar{x}(t) - x^*(t)| > \Delta Q$$

If this inequality holds the integrator behaves as described in Subsection 3.2.1 on page 32 meaning that the time until the next internal event is found. On the other hand, if it is violated, the integrator sets its time-advance function to zero ($ta(\cdot) = 0$), which causes the integrator to produce a new output immediately and hence update all first and second derivative outputs of the connected function block (representing $\mathbf{f}(\mathbf{x}, \mathbf{u})$), which causes all integrators to be updated with current information following the measurement update.

4.3.4 QSS/EKF Implementation Details

The implementation of the quantised state filter requires a QSS2 simulation to be setup, see Figure 3.3, with the process model as the driving function, $\mathbf{f}(\mathbf{x})$, and the sensor model as the output function, $\mathbf{h}(\mathbf{x})$, with such a model in place (see e.g. Chapter 3 on page 29) the call for constructing the filter is:

```
EKF(double cT,int nM, Matrix P, Matrix Q, Qss2Map mMap,
    Qss2ResetIntegrator[] ints);
```

where `cT` is a guaranteed maximum time between covariance propagation, `nM` is the number of associated measurements, `P` is the initial covariance matrix, `Q` is a matrix of continuous time process noise variances, `mMap` is a reference to the function, $\mathbf{h}(\mathbf{x})$, and `ints` are references to the reset-able integrators.

The `Qss2ResetIntegrator` is a class that extends the `Qss2Integrator` class defined in Section 2.3 on page 24) with the reset functionality described in Subsection 4.3.3. Measurements are registered using the following call for each measurement:

```
addMeasurement(int nM, int[] rows, Matrix R);
```

where `nM` is the measurement number, `rows` are row-indexes for $\mathbf{h}(\mathbf{x})$ for the corresponding measurement and `R` is an associated matrix of measurement noise variances. In this way measurements from multiple sensors can be supported.

In addition to these calls then DEVS connections between the blocks must be set-up using the standard calls for connecting blocks in a coupled model (see Section 2.3 on page 24).

The discussion of the function calls above serves to demonstrate how simple it is to add estimation to a QSS2 model; This allows a user to concentrate on the modelling part of the task rather than implementation of standard algorithms.

The `Matrix` type that is used to represent matrices in the software is the Jama public available matrix library for Java developed by *MathWorks* and the *National Institute of Standards and Technologies**.

During operation of the filter `addMeasurement()` can be called again to update measurement descriptions. If e.g. in an industrial plant a sensor is replaced with a more accurate one.

4.4 Simulation Case Study: Attitude Determination

Inspired by deep space missions such as Cassini [Johnson and Brown, 1998] and New Horizons [Stern and Spencer, 2003] the following presents a Deep Space Probe (DSP) attitude control case study that will be used for evaluating the developed filter (and which will be revisited in subsequent chapters). The next subsection will provide an overview of the case and the following subsection will provide simulation results for the QSS/EKF algorithm applied on the case and compare performance against a standard EKF implementation.

4.4.1 Model of a Deep Space Probe

The DSP is a single body spacecraft for deep space exploration; a generic mission profile with operational phases following spacecraft launch is indicated in Figure 4.4.

*Available from http://math.nist.gov/javanumerics/jama/

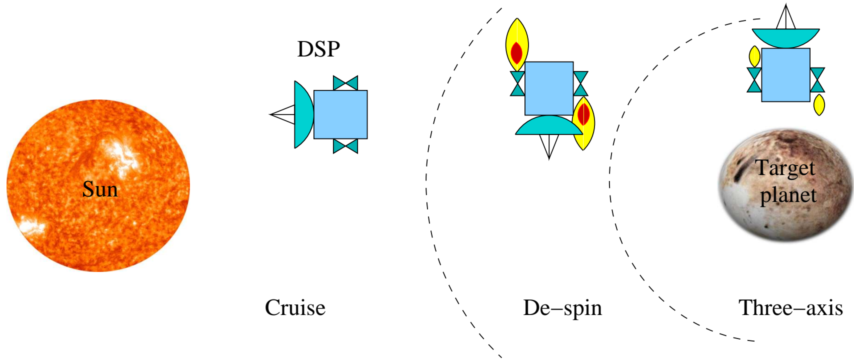


Figure 4.4: A deep space mission scenario with three autonomous attitude control modes depending on distance to target.

In the *Cruise phase* the probe is spinning around its major axis of inertia at 5 RPM with the main dish pointing towards the sun. Except for occasional spin-axis adjust manoeuvres the craft is not actively controlled.

As the DSP reaches a certain distance from the target planet, it performs a *de-spin* manoeuvre using pulse-controlled thrusters, whereafter the craft enters a *three-axis* controlled mode utilising precision thrusters and where the DSP must maintain inertial pointing at the target attitude with minimal disturbances rates.

This chapter will consider only the cruise mode where the task is to estimate the attitude of the craft using the QSS/EKF algorithm driven by simulated sensor data.

Kinematical Model

The kinematical description of the DSP is based on Euler angles (3-2-1 rotation order) and the kinematical equation is described by [Wertz, 1978]:

$$\mathbf{f}_1([\theta^T\ \omega^T]^T) = \dot{\theta} = \frac{1}{\cos\theta_2}\begin{bmatrix} \cos\theta_2 & \sin\theta_1\sin\theta_2 & \cos\theta_1\sin\theta_2 \\ 0 & \cos\theta_1\cos\theta_2 & -\sin\theta_1\cos\theta_2 \\ 0 & \sin\theta_1 & \cos\theta_1 \end{bmatrix}\omega \tag{4.11}$$

Dynamical Model

The time invariant dynamical model of the DSP described in a body centred coordinate system has the following form:

$$\mathbf{f}_2([\theta^T\ \omega^T]^T) = \dot{\omega} = \mathbf{J}^{-1}\left(-[\omega\times]\mathbf{J}\omega + \mathbf{n}_{con} + \mathbf{n}_{dist}\right) \tag{4.12}$$

where $\omega = [\omega_1\ \omega_2\ \omega_3]^T$ are the body rates, $\mathbf{n}_{con}$ is the vector of control torques, $\mathbf{n}_{dist}$ is the vector of disturbance inputs, and the parameter $\mathbf{J}$ is the inertia matrix with nominal values:

$$\mathbf{J} = \begin{bmatrix} 30.0 & -0.5 & -1.0 \\ -0.5 & 30.0 & -1.5 \\ -1.0 & -1.5 & 50.0 \end{bmatrix} \tag{4.13}$$

and where $[\omega\times]$ is a skew symmetric matrix representing the gyro-scopic coupling [Wertz, 1978]:

$$[\omega\times] = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix}$$

During cruise mode, the DSP is spinning about its major axis at a nominal rate of 5 RPM and there is no actuation, i.e. $\mathbf{n}_{con} = 0$. The DSP is affected by unmodelled disturbance torques originating from e.g. :

- Solar radiation pressure
- Material out-gassing
- Propellant sloshing in propellant tanks
- Magnetic induced torques

It will be assumed that these disturbance effects can be adequately regarded as zero mean torques with Gaussian distributions with standard deviations of $\sigma_{dist} = 10^{-4}$ for each axis.

Sensor Model

The DSP utilises a simple sun-sensor and simple star-sensor in cruise mode which are mounted on the Earth facing side of the spacecraft, see Figure 4.5.

A sensor model for both sensors can be described by the relation:

$$\mathbf{y} = \mathbf{h}(\theta) = \mathbf{C}_{321}(\theta) \cdot \left(\frac{\mathbf{X}_{DSP} - \mathbf{X}_{target}}{|\mathbf{X}_{DSP} - \mathbf{X}_{target}|} \right) + \mathbf{v}$$

where $\mathbf{X}_{DSP}$ is the position of the probe, $\mathbf{X}_{target}$ is the position of the sun or guide star respectively, and $\mathbf{C}_{321}(\theta)$ is the direction cosine matrix corresponding to the rotation specified by the Euler angles. Noise, $\mathbf{v}$, for both sensors is Gaussian distributed with zero mean and with the following standard deviation on all axes and the stated update frequency.

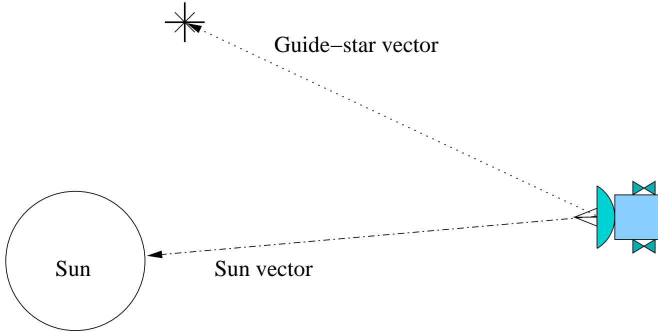


Figure 4.5: The DSP gets its attitude information in cruise mode by measuring the angle to the sun and a bright guide star respectively.

- Sun-sensor: $\sigma_{sun} = 1^o$ @ 2 Hz
- Star-sensor: $\sigma_{star} = 0.1^o$ @ 0.2 Hz

It should be noted that each sensor by itself does not provide full state observability since the rotation around the boresight of a single vector sensor cannot be inferred.

4.4.2 Implementation of the Standard EKF Algorithm

In order to compare results of QSS/EKF filtering with results from the standard EKF algorithm, an EKF filter was implemented in *Matlab*, based on Section 4.2 and the model in Subsection 4.4.1. The challenge in implementing the filter is the derivation of expressions for the required Jacobians. A Jacobian for the dynamical mode, i.e. Equation (4.12), can be derived by direct differentiation [Bhanderi, 2005]:

$$\mathbf{J}_{dyn}([\theta^T \omega^T]^T) = \frac{\partial \mathbf{f}_2}{\partial[\theta^T \omega^T]} = \mathbf{J}^{-1}\left([\mathbf{J}\omega\times] - [\omega\times]\mathbf{J}\right)$$

The Jacobian for the kinematical equation, i.e. e.q. (4.11), was derived using the *Maple* software package for computer assisted analytical mathematics ($s(\cdot)$ and $c(\cdot)$ are short for $\sin(\cdot)$ and $\cos(\cdot)$ respectively):

$$\mathbf{J}_{kin}([\theta^T \omega^T]^T) = \frac{\partial \mathbf{f}_1}{\partial[\theta^T \omega^T]} =$$

$$\begin{bmatrix} s(\theta_2)(c(\theta_1)\omega_2 - s(\theta_1)\omega_3)/c(\theta_2) & (s(\theta_1)\omega_2 + c(\theta_1)\omega_3)/c(\theta_2)^2 & 0 & 1 & s(\theta_1)s(\theta_2)/c(\theta_2) & c(\theta_1)s(\theta_2)/c(\theta_2) \\ -s(\theta_1)\omega_2 - c(\theta_1)\omega_3 & 0 & 0 & 0 & c(\theta_1) & -s(\theta_1) \\ (c(\theta_1)\omega_2 - s(\theta_1)\omega_3)/c(\theta_2) & s(\theta_2)(s(\theta_1)\omega_2 + c(\theta_1)\omega_3)/c(\theta_2)^2 & 0 & 0 & s(\theta_1)/c(\theta_2) & c(\theta_1)/c(\theta_2) \end{bmatrix}$$

Finally, a Jacobian is required for each measurement; Again *Maple* has been utilised to analytically evaluate the Jacobian of the measurement model, $\mathbf{h}(\cdot)$:

$$\mathbf{J}_g([\theta^{\mathbf{T}}\omega^{\mathbf{T}}]^T) = \frac{\partial \mathbf{h}}{\partial[\theta^{\mathbf{T}}\omega^{\mathbf{T}}]} =$$

$$\begin{bmatrix} 0 & -s\theta_2 c\theta_3 u_1 - \dots & s\theta_2 s\theta_3 u_2 - c\theta_2 u_3 & -c\theta_2 s\theta_3 u_1 + c\theta_2 c\theta_3 u_2 & 0 & 0 & 0 \\ (c\theta_1 s\theta_2 c\theta_3 + s\theta_1 s\theta_3)u_1 + (c\theta_1 s\theta_2 s\theta_3 - s\theta_1 c\theta_3)u_2 + c\theta_1 c\theta_2 u_3 & s\theta_1 c\theta_2 c\theta_3 u_1 + \dots & s\theta_1 c\theta_2 s\theta_3 u_2 - s\theta_1 s\theta_2 u_3 & (-s\theta_1 s\theta_2 s\theta_3 - c\theta_1 c\theta_3)u_1 + (s\theta_1 s\theta_2 c\theta_3 - c\theta_1 s\theta_3)u_2 & 0 & 0 & 0 \\ (-s\theta_1 s\theta_2 c\theta_3 + c\theta_1 s\theta_3)u_1 + (-s\theta_1 s\theta_2 s\theta_3 - c\theta_1 c\theta_3)u_2 - s\theta_1 c\theta_2 u_3 & c\theta_1 c\theta_2 c\theta_3 u_1 + \dots & c\theta_1 c\theta_2 s\theta_3 u_2 - c\theta_1 s\theta_2)u_3 & (-c\theta_1 s\theta_2 s\theta_3 + s\theta_1 c\theta_3)u_1 + (c\theta_1 s\theta_2 c\theta_3 + s\theta_1 s\theta_3)u_2 & 0 & 0 & 0 \end{bmatrix}$$

where $\mathbf{u} = [u_1 \;\; u_2 \;\; u_3]$ is a unit vector to the target, i.e. the sun or a chosen guide-star, and s and c is short for $\sin(\cdot)$ and $\cos(\cdot)$ respectively. The derivation of these Jacobians clearly demonstrate that even for a relatively simple system the results can be quite complex.

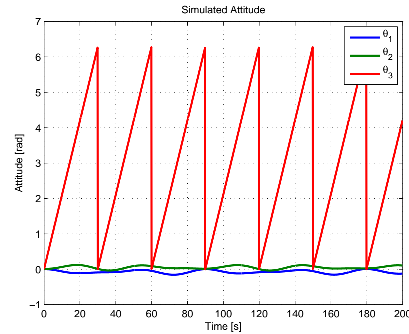
The EKF algorithm was implemented with a sample-time of 0.05 s, which was found by observing that no improvement in the estimation accuracy resulted from decreasing the sample-time further. Measurement updates are performed for each of the sensors at appropriate multiples of the sample number consistent with the sample interval for the sun-sensor and star-sensor respectively.

4.4.3 Simulation Results

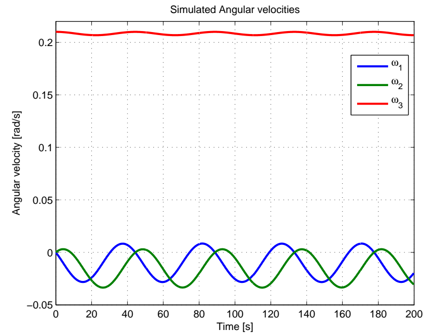
In order to evaluate performance, a model based on Subsection 4.4.1, was implemented in *Simulink* and it is used to generate measurement data sequences and associated state values. The QSS/EKF filter implemented in DEVS and the normal EKF implemented in *Matlab* have been tested with the generated data sequences and the results are presented in the following.

For the case to be presented here, the initial conditions for the state were $\omega = [0\;0\;0.21]^T$ rad/s and $\theta = [0\;0\;0]^T$ rad. Each filter was initialised with initial

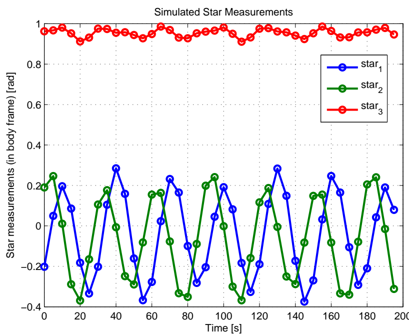
state $\hat{\omega} = [0 \ -0.03 \ 0.15]^T$ rad/s and $\hat{\theta} = [0.1 \ -1 \ 0]^T$ rad. The covariance matrix was initialised with standard deviations of $\sigma = 0.05$ rad for attitude states and $\sigma = 0.001$ rad/s for angular velocity states.



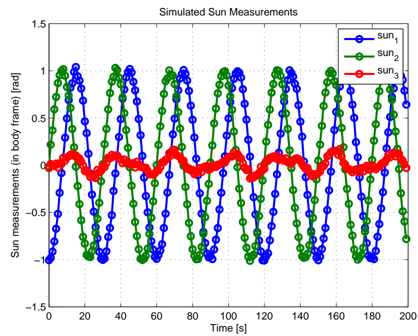
(a) Attitude



(b) Angular velocities



(c) Star sensor measurements



(d) Sun sensor measurements

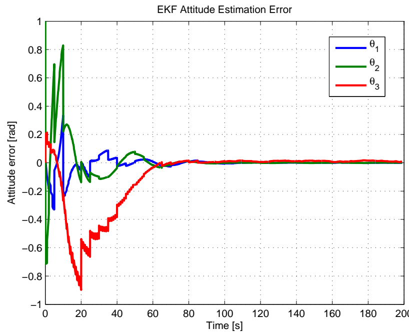
Figure 4.6: "Truth model" state and measurement values as simulated by *Simulink*.

Figure 4.6 shows the resulting state trajectories and measurements produced by the simulation in *Simulink* from the stated initial values. It can be seen how the behaviour is dominated by the slow rotation around the major axis of inertia with a small amount of precession due to the off-diagonal terms in the inertia matrix.

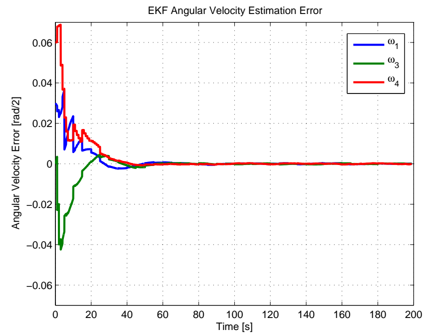
EKF Results

On Figure 4.7 the result using the normal EKF algorithm can be seen in terms of error signals between the estimated states by the EKF and the simulation results

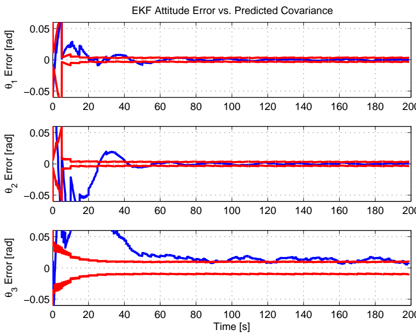
presented in Figure 4.6.



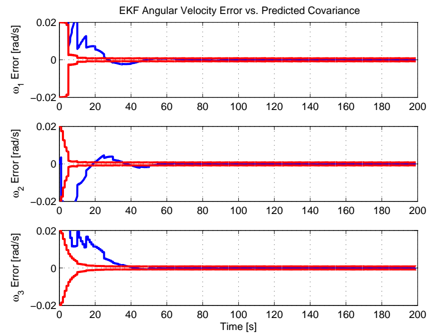
(a) Attitude error



(b) Angular velocity error



(c) Attitude errors with 2σ bounds



(d) Angular velocities with 2σ bounds

Figure 4.7: EKF filter performance. Note the small bias on the θ_3 state.

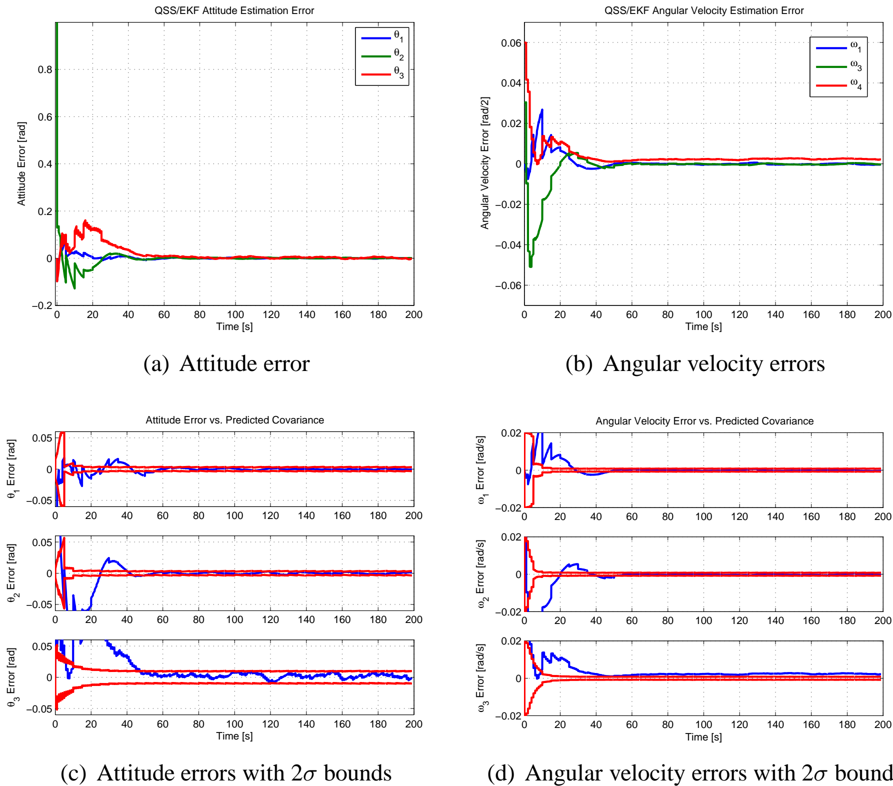
The (a) and (b) graphs show the error plots for attitude and angular velocity states respectively, while the (c) and (d) plots show how the error fits with the predicted covariance for each state. The filter converges in approximately 50 seconds and it can be seen that after convergence there is good correspondence between estimation errors and their associated covariances. However, it can be seen from the (c) plot that the θ_3 state suffers from a small bias in the estimate, however, it is known [Grewal and Andrews, 1993, Zarchan and Musoff, 2000] that the EKF is not an unconditionally unbiased estimator - performance can to some extent be recovered by introducing extra process noise [Zarchan and Musoff, 2000].

The behaviour of the EKF filter as illustrated in these plots is consistent, qualitatively, with other runs of the filter with different initial state estimates, $\hat{\omega}$.

QSS/EKF Results

The QSS/EKF filter was initialised with the same initial conditions as described above for the EKF. The QSS2 implementation of the DSP model utilises quanta of $\Delta Q = 10^{-5}$ rad for the attitude states and $\Delta Q = 10^{-6}$ rad/s for angular velocity states. A maximum covariance propagation delay of 0.05 s was chosen to make the QSS/EKF filter perform as similar as the EKF as possible.

On Figure 4.8 the result using the normal QSS/EKF algorithm can be seen in terms of error plots between the estimated states by the QSS/EKF and the simulation results presented in Figure 4.6.



(a) Attitude error

(b) Angular velocity errors

(c) Attitude errors with 2σ bounds

(d) Angular velocity errors with 2σ bounds

Figure 4.8: QSS/EKF filter performance. Results are qualitatively equal to the conventional EKF filter.

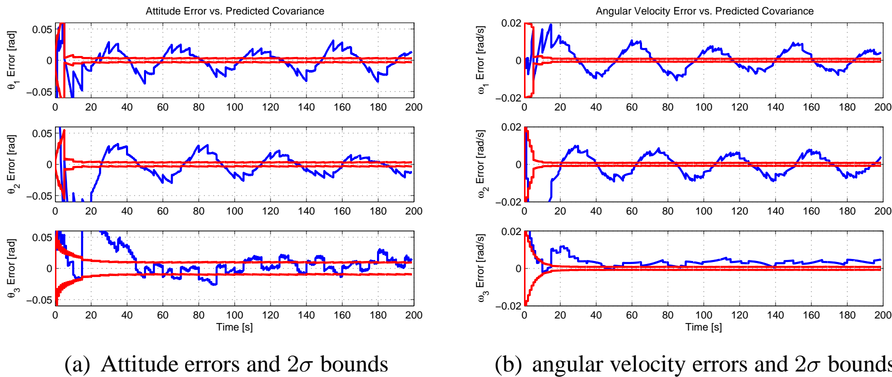
Again it can be seen that the filter converges in approximately 50 seconds, as

for the EKF, and if one compares the error trajectories to Figure 4.7 it is clear that the two filters provide almost identical filtering. However, in the QSS/EKF filter the small bias on state θ_3 has disappeared, but instead a small bias has now become apparent on the ω_3 state (which again can be reduced by introducing more process noise). The results are consistent with other initial conditions.

The execution time for the *Matlab* based EKF is 2.3 s and 1.1 s for the DEVS based QSS/EKF filter. However, these numbers cannot be compared directly as it would be as much a comparison of execution platforms as algorithm performance. From the plots presented it is concluded that the QSS/EKF provides an quantised state systems alternative to the sample-based extended Kalman filter.

Effect of Increased QSS2 Quanta

It is interesting to investigate the influence of the quantum size selection on the filter performance; Figure 4.9 shows results for the QSS/EKF filter on the same problem as before, but with increased quanta, now all equal to $\Delta Q = 10^{-3}$.



(a) Attitude errors and 2σ bounds

(b) angular velocity errors and 2σ bounds

Figure 4.9: QSS/EKF Performance with increased quantum. Periodic oscillation is evident.

It can be seen that the filter does converge to the correct state trajectories, but that the estimation error is no longer consistent with the predicted covariances. Generally speaking the oscillations that can be seen in the error trajectories are on the same magnitude as the chosen quantum. The reduced quanta here also decreased execution time from 1.1 s to 0.7 s as less integration steps are carried out in the QSS2 model.

Based on this example and others, it is stated that as a rule of thumb one

should chose quanta for each state that are at least a magnitude lower than the expected covariance in steady-state operation in order for the QSS/EKF filter to operate consistently.

4.5 Chapter Summary

This chapter introduced the QSS/EKF filter which is an extended Kalman filter implemented for use with quantised state systems. A case study of attitude determination for a deep-space probe demonstrated that the QSS/EKF filter performs almost identically to the conventional sample-based EKF implementation.

Contrary to the original EKF algorithm, the QSS/EKF alternative does not require analytical expressions for the state and measurements Jacobians respectively, but instead Jacobians are provided at no additional computational cost by the QSS2 algorithm used for state propagation. For systems where it is impractical or impossible to analytically derive expressions for the Jacobian, or where such expressions becomes very computationally expensive the QSS/EKF algorithm provides an interesting alternative to the conventional EKF algorithm.

Secondly, the QSS/EKF filter is a reusable implementation that effectively encapsulates the algorithm and only requires the user to specify the model of the system and associate measurements as a QSS2 model. The QSS/EKF algorithm and the model can then be composed at run-time.

Optimising Control of QSS Systems

5

This chapter contributes with an optimisation based control algorithm that can be composed with quantised state system models to provide a controller for a given plant. The algorithm is presented, stability properties are discussed, and finally the algorithm is evaluated on an example involving an autonomous underwater vehicle.

5.1 Event Based Control and Quantised State Systems

In this chapter, and the next, controllers are sought which from a description of the system and a description of the control objective can generate input signals which drives the system's states towards fulfillment of the control objective. The controllers will utilise QSS2 models of both plant and objectives in the process.

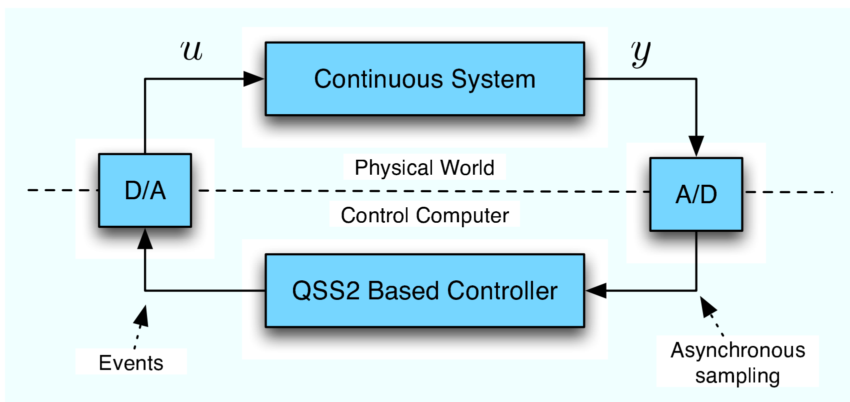


Figure 5.1: Feedback control structure for a system being controlled by a control computer executing a QSS2 implemented control law.

In [Kofman, 2003] it is shown that the feedback-coupling of a continuous system and a quantised state system (See Figure 5.1) results in a closed loop which fulfils similar ultimate boundedness criteria as for when simulating a QSS system, see Section 3.3 on page 38. This means that any feedback controller designed for a continuous system, which can be described as a set of ordinary differential equations, can be implemented as a QSS2 model on the control computer. The drawback is, as in the simulation chapter, small oscillations in the solution trajectories. However, it is always possible to find a sufficient small quantisation which makes these oscillations insignificant compared to system noise that is inherent to any control system.

Further, in [Kofman, 2003], it is shown that if controllers for Linear Time Invariant (LTI) systems are designed and then implemented on the control computer as QSS systems and asynchronous sampling* is utilised, significant savings in computation time is obtained, when compared to implementations of the same controller using a sample-based approach.

In this chapter QSS2 models will be used to provide local linear abstractions of the system which is to be controlled. The control algorithms presented in this dissertation utilises these local linear models and finds inputs which minimise a control objective function chosen by the system designer.

This chapter contributes with one such algorithm where focus is on simplicity of the required calculations, while the next section describes a more evolved approach. Section 5.2 presents the development of the controller, Section 5.3 analyses stability of the method, and Section 5.6 demonstrates, through simulation, the applicability of the controller to the autonomous underwater vehicle presented earlier.

The algorithm and simulation study involving the autonomous underwater vehicle was published in [Alminde et al., 2007b]. Estimation based control based on the controller developed here is presented in Chapter 7 on page 109.

5.2 A Simple Optimising QSS2 Controller

This section presents a control algorithm developed with the aim to provide an algorithmicly simple control algorithm using an on-line QSS2 model of the plant to be controlled and a control objective function. The idea is to present an on-line

*using a second or first order quantum separation principle to determine event times

algorithm that works well for a large class of systems and which only requires the user to specify models defining the system and objective.

5.2.1 A Control Algorithm for Single Objective Control

For control purposes we will augment the system description with a control objective in the form of a scalar convex control objective function which maps the state to a scalar value. The control problem is defined by the functions:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \tag{5.1}$$

$$v_c = v_c(\mathbf{x}) \tag{5.2}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the state vector of dimension n, $\mathbf{u} \in \mathbb{R}^m$ is the input vector of dimension m, $\mathbf{f}(\cdot)$ is a differentiable map $\mathbf{f} : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^n$ representing the system dynamics, and $v_c(\cdot) : \mathbb{R}^n \rightarrow \mathbb{R}^1$ is the control objective function defined by the user. The control problem is to find input signals that minimise Equation (5.2).

At this point we will require the control objective function to be a convex function with a single minimum point at the point where the user wish to stabilize the system. The next section will derive more stringent requirements on v_c that will guarantee stability of the system.

The QSS2 algorithm provides the Hessian $\frac{\partial \mathbf{f}}{\partial \mathbf{z}}(\mathbf{z}(t_k))$ with $\mathbf{z} = [\bar{\mathbf{x}}^\mathbf{T}\ \bar{\mathbf{u}}^\mathbf{T}]^\mathbf{T}$ (see Subsection 3.2.2 on page 33), which can be divided into two matrices $\mathbf{A}(\bar{\mathbf{x}}, \bar{\mathbf{u}}) \simeq \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ and $\mathbf{B}(\bar{\mathbf{x}}, \bar{\mathbf{u}}) \simeq \frac{\partial \mathbf{f}}{\partial \mathbf{u}}$ representing state sensitivity and input sensitivity, respectively.

Similarly for $v_c(\bar{\mathbf{x}})$ the QSS2 algorithm provides a state sensitivity matrix $\mathbf{E}(\bar{\mathbf{x}}) \simeq \frac{\partial v_c}{\partial \mathbf{x}}$, which is the Jacobian of Equation (5.2)[†]. These matrices are communicated to the controller, and the controller output $\dot{\mathbf{u}}$ is fed to the plant through a set of integrators, see Figure 5.2.

The choice of letting the controller control the input slopes $\dot{\mathbf{u}}$ rather than $\mathbf{u}$ directly, is due to the fact that the matrices used in the calculation are based on a fixed operating point $(\bar{\mathbf{x}}, \bar{\mathbf{u}})$, so any immediate change in the control signal would cause these matrices to be updated and hence cause the controller to perform another control calculation. To avoid this algebraic loop the control slopes are taken

[†]In fact a row vector since v_c is scalar, but we leave the matrix symbol for future extension to multiple objective control

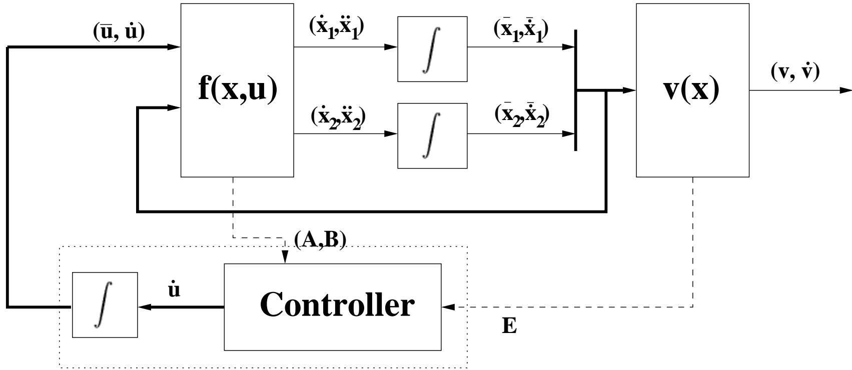


Figure 5.2: QSS Control structure. Thick lines are vector signals. Dashed lines are matrix signals. This figure has two states and one input.

as outputs from the controller which thereby leaves the QSS2 mechanism for automatically switching operating points intact.

The control strategy chosen is to provide an input signal that maintains $\dot{v}_c$ negative. In order to do this with only information about the matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{E}$ we neglect the autonomous response of the system and only consider the forced response, implications for stability will be addressed later. To this end we need to see how $\dot{v}_c$ is affected by the control vector $\dot{\mathbf{u}}$ over a time horizon $\tau = t_{k+1} - t_k$, which corresponds to the time until the next scheduled QSS2 event. This change is given by the expression:

$$\Delta_{\dot{\mathbf{u}}}\dot{v}_c(\bar{\mathbf{x}}, \dot{\mathbf{u}}, \tau) = \mathbf{E}(\bar{\mathbf{x}})\mathbf{\Gamma}(\bar{\mathbf{x}}, \bar{\mathbf{u}}, \tau)\dot{\mathbf{u}} \tag{5.3}$$

where $\Delta_{\dot{\mathbf{u}}}\dot{v}_c(\bar{\mathbf{x}}, \dot{\mathbf{u}}, \tau)$ is to be read as "the change of $\dot{v}_c(\bar{\mathbf{x}}, \dot{\mathbf{u}}, \tau)$ due to $\dot{\mathbf{u}}$ over the time-horizon τ", and where $\mathbf{\Gamma}(\bar{\mathbf{x}}, \bar{\mathbf{u}}, \tau)$ is the input transition matrix [Gene F. Franklin and Emami-Naeini, 1994]:

$$\mathbf{\Gamma}(\bar{\mathbf{x}}, \bar{\mathbf{u}}, \tau) = \sum_{k=0}^{\infty} \frac{\mathbf{A}^k(\bar{\mathbf{x}}, \bar{\mathbf{u}})\tau^{k+1}}{(k+1)!}\mathbf{B}(\bar{\mathbf{x}}, \bar{\mathbf{u}}) \tag{5.4}$$

Based on the limited information herein we will construct an algorithm that seeks to minimise $v_c(\mathbf{x})$ by choosing appropriate control action, $\dot{\mathbf{u}}$. As Figure 5.2 introduces integrators for control signal $\mathbf{u}$, we also need to penalise this state in the control objective function in order to ensure that control signals approach $\mathbf{u_f}$

as the control objective function is minimised. To this end we use a quadratic cost term for the control state:

$$v_u(\mathbf{u}) = \frac{1}{2}(\mathbf{u} - \mathbf{u_f})^T \mathbf{P}(\mathbf{u} - \mathbf{u_f})$$

with parameters given by the matrix $\mathbf{P} = \mathbf{P^T} > 0$, and possible preset input levels (feed-forward), $\mathbf{u_f}$. We now introduce the joint performance function:

$$v(\mathbf{x}, \mathbf{u}) = v_c(\mathbf{x}) + v_u(\mathbf{u}) \tag{5.5}$$

and introduce a term for the input cost in Equation (5.3):

$$\Delta_{\dot{\mathbf{u}}}\dot{v}(\bar{\mathbf{x}}, \bar{\mathbf{u}}, \dot{\mathbf{u}}, \tau) = \Big(\mathbf{E}(\bar{\mathbf{x}})\mathbf{\Gamma}(\bar{\mathbf{x}}, \bar{\mathbf{u}}, \tau) + \tau(\mathbf{u} - \mathbf{u_f})^T \mathbf{P}\Big)\, \dot{\mathbf{u}} \tag{5.6}$$

it is noted that the expression between the outer parenthesis evaluates to a row vector which we will denote $\mathbf{c}$, i.e. :

$$\Delta_{\dot{\mathbf{u}}}\dot{v}(\bar{\mathbf{x}}, \bar{\mathbf{u}}, \dot{\mathbf{u}}, \tau) = \mathbf{c}\dot{\mathbf{u}}$$

The minimisation will be performed over a control horizon that corresponds to the next scheduled integrator output event in the QSS2 simulation. Furthermore, since it is not possible to evaluate the infinite sum of Equation (5.4), it is evaluated using the first n terms, where n is the number of states in the system. This choice ensures that minimisation includes full information about state controllability. The minimisation problem can now be stated as:

$$\begin{aligned} &\text{minimize } \Delta_{\dot{\mathbf{u}}}\dot{v}(\bar{\mathbf{x}}, \bar{\mathbf{u}}, \dot{\mathbf{u}}, \tau) = \mathbf{c}\dot{\mathbf{u}} \\ &\text{subject to:} \\ &\quad \dot{\mathbf{u}} \preceq \dot{\mathbf{u}}_{\text{max}} \\ &\quad \dot{\mathbf{u}} \succeq \dot{\mathbf{u}}_{\text{min}} \\ &\quad \mathbf{u} \preceq \mathbf{u}_{\text{max}} \\ &\quad \mathbf{u} \succeq \mathbf{u}_{\text{min}} \end{aligned} \tag{5.7}$$

where the constraints $\dot{\mathbf{u}}_{\text{max}}$ and $\dot{\mathbf{u}}_{\text{min}}$ are due to rate limiting and $\mathbf{u}_{\text{max}}$ and $\mathbf{u}_{\text{min}}$ are due to actuator saturation. Denoting the optimal input $\dot{\mathbf{u}}^*$ it can be seen that due to the simple dot-product form of the problem, each component, index i, of $\dot{\mathbf{u}}^*$ can easily be found:

$$\dot{u}_i^* = \begin{cases} \dot{u}_{\text{min},i} & \text{if } c_i > 0 \ \wedge\ u_i < u_{\text{max},i} \\ \dot{u}_{\text{max},i} & \text{if } c_i < 0 \ \wedge\ u_i > u_{\text{min},i} \\ 0 & \text{otherwise} \end{cases} \tag{5.8}$$

Described in words the control algorithm finds a control slope for each control signal which based on local information of the model and control objective simultaneously seeks to minimise the functions v_c and v_u. The control slopes are recalculated whenever the QSS2 model informs the controller that either the model matrices $\mathbf{A}, \mathbf{B}$, or the objective matrix $\mathbf{E}$ has changed.

In practise the control signal slope limits $\dot{\mathbf{u}}_{\text{max}}$ and $\dot{\mathbf{u}}_{\text{max}}$ can be chosen from physical insight representing the physical limits of the actuators.

5.3 Stability Analysis

We are investigating the stability properties of the following continuous system, where $\mathbf{k}(\mathbf{x}, \mathbf{u})$ represents the control law (cf. Equation (5.7)):

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \tag{5.9}$$

$$\dot{\mathbf{u}} = \mathbf{k}(\mathbf{x}, \mathbf{u}) \tag{5.10}$$

$$v(\mathbf{x}, \mathbf{u}) = v_c(\mathbf{x}) + \frac{1}{2}(\mathbf{u} - \mathbf{u_f})^{\mathbf{T}}\mathbf{P}(\mathbf{u} - \mathbf{u_f}) \tag{5.11}$$

Due to the formulation with $\mathbf{f}(\cdot)$ and $v_c(\cdot)$ being general non-linear functions and $\mathbf{k}(\cdot)$ a controller exhibiting switching behavior, stability for the general case is difficult to prove. The following analysis will show that the proper framework for analysing stability of the system is within stability theory for hybrid systems, and sufficient conditions for (exponential) stability will be formulated.

5.3.1 Equivalent Switched System Model

The system described above can alternatively be described using the extended state-space $\xi = [\mathbf{x}^T \ (\mathbf{u} - \mathbf{u_f})^T]^T$. Furthermore, we introduce the notation: $\mathbf{g}(\xi) = [\mathbf{f}(\mathbf{x}, \mathbf{u})^T \ \mathbf{k}(\mathbf{x}, \mathbf{u})^T]^T$ and $\mathcal{V}(\xi) = v_c(\mathbf{x}) + v_u(\mathbf{u})$:

$$\dot{\xi} = \mathbf{g}(\xi) \tag{5.12}$$

$$v = \mathcal{V}(\xi) \tag{5.13}$$

It is noted that the control law, i.e. Equation 5.7, results in a control vector of discrete valued components each corresponding to a maximum, minimum or zero

value of a given input signal ($\dot{u}_i$). Therefore we can realise $\mathbf{g}$ as a switched system of vector fields $\mathbf{g}_i \in \mho = \{\mathbf{g}_1, \ldots, \mathbf{g}_{3^m}\}$ where:

$$\mathbf{g}_i(\xi) = \begin{bmatrix} \mathbf{f}(\mathbf{x}, \mathbf{u}) \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \mathbf{v}_i \end{bmatrix} \tag{5.14}$$

here $\mathbf{v}_i$ is one of the 3^m possible input vectors that $\mathbf{k}(\mathbf{x}, \mathbf{u})$ can switch between. By viewing the control problem in this light we see Equation (5.12) as a variable structure system where the controller switches between different realisations with the goal of stabilising the system at $\xi = 0$.

To introduce the quantised state approach we note that for each control calculation the controller is presented with a local affine model in place of the general non-linear model, i.e.:

$$\dot{\xi} = \begin{bmatrix} \mathbf{f}(\bar{\mathbf{x}}, \bar{\mathbf{u}}) \\ 0 \end{bmatrix} + \begin{bmatrix} \mathbf{A}(\bar{\mathbf{x}}, \bar{\mathbf{u}}) & \mathbf{B}(\bar{\mathbf{x}}, \bar{\mathbf{u}}) \\ 0 & \mathbf{I}_{m \times m} \end{bmatrix} \left(\xi - \bar{\xi}\right) + \begin{bmatrix} 0 \\ \mathbf{v}_i \end{bmatrix} \tag{5.15}$$

In principle we can think of the right hand side as a set of possible realisations of the system's dynamics and control input. Each time the QSS2 algorithm switches to a new realisation of the system model the controller switches to an appropriate realisation of the input signal.

For a simple linear system there would be 3^m possible systems to switch between and for a general non-linear system there could be infinitely many. The choice of quantisation vector $\mathbf{\Delta Q}$ is a parameter adjusting how often the system model is switched and hence how fine-masked the QSS2 algorithm approximates the underlying non-linear system.

In the following we will discuss stability of the continuous system described by Equation (5.12-5.13). We thereby assume that the user has chosen a quantization $\mathbf{\Delta Q}$ that approximates the continuous equations closely.

5.3.2 Hybrid Stability

To discuss stability of the switched system described above we need to define what is meant by stability for such a system. Consider an autonomous switched/hybrid system on the form:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, q(t)) = \mathbf{f}_{q(t)}(\mathbf{x}) \tag{5.16}$$

where $q : \mathbb{R} \rightarrow \{1, \ldots, M\}$ assigns a specific realisation of the vector field to the system at a given time. $q(t)$ is piecewise continuous from the right implying

that there can only be a finite number of switches per unit of time. For each $\mathbf{f}_i$ we associate a Lyapunov like function (definition below) v_i and region in the state-space Ω_i where the vector field is switched on.

Definition 5.1 (Lyapunov Like Function [Branicky, 1994])
A Lyapunov-like function v_i associated with a region Ω_i satisfies the following two conditions with $\mathbf{x} \in \Omega_i$:

- $v_i(0) = 0$ *and* $v_i(\mathbf{x}) > 0$ *for* $\mathbf{x} \neq 0$ *(positive definiteness)*
- $\dot{v}_i(\mathbf{x}) = \frac{\partial v_i(\mathbf{x})}{\partial \mathbf{x}} \mathbf{f}_i(\mathbf{x}) \leq 0$ *(negative definite derivative)*

With this notation and definition in mind we can state a theorem giving *sufficient conditions* for stability of a switched/hybrid system (proof in [Branicky, 1994]):

Theorem 5.2 (Stability of Hybrid Systems [Branicky, 1994])
Given the M-switched non-linear system of Equation (5.16), suppose each vector field $\mathbf{f}_i$ has an associated Lyapunov-like function v_i in the region Ω_i, each with equilibrium point $\bar{\mathbf{x}} = 0$, and suppose $\cup_i \Omega_i = \mathbb{R}^n$. Let $q(t)$ be a given switching sequence such that $q(t)$ can take on the value i only if $\mathbf{x}(t) \in \Omega_i$, and in addition:

$$v_i(\mathbf{x}(t_{i,k})) \leq v_i(\mathbf{x}(t_{i,k-1})) \tag{5.17}$$

where $t_{i,k}$ denotes the kth time that vector field $\mathbf{f}_i$ is "switched in", i.e. $q(t^-_{i,k}) \neq q(t^+_{i,k}) = i$. Then Equation (5.16) is Lyapunov stable.

If in Definition (5.1) condition two and in Theorem (5.2) Equation (5.17) the less-than-or-equal sign is replaced with a strictly less-than then the system is asymptotically stable in the sense of Lyapunov.

In words; Each time a new vector field is selected the associated Lyapunov like function must be less than or equal to the same function evaluated last time the same vector field was exited. Furthermore, the Lyapunov like function must be non-increasing while the associated vector field is turned on.

Consider $\mathcal{V}(\xi)$ a candidate Lyapunov like function for the system given by Equation (5.12) with the associated region $\Omega_{\mathcal{V}(\xi)} = \mathbb{R}^{n+m}$ meaning that $\mathcal{V}(\xi)$ is associated with all realisations $\mathbf{g}_i \in \mho$. Let $q(t)$ be the sequence of realisations $\mathbf{g}_i$ that is chosen by the controller (Equation (5.7)) then the system is stable if the conditions of Theorem (5.2) are satisfied from any initial condition.

As the controller always chooses the realisation $\mathbf{g}_i$ that will contribute most negatively to $\dot{\mathcal{V}}(\xi)$ stability is now a question of if the user supplied function $v_c(\mathbf{x})$ together with the chosen matrix $\mathbf{P}$ that in fact provides a $\mathcal{V}(\xi)$ that is a global Lyapunov function for the switched system described by Equation (5.12).

As a general declarative controller this is insufficient as it places the responsibility on the user. Practical experience, however, show that for some systems it is relatively easy to select a good control objective function and input cost matrix that provides a satisfactory response. The type of systems where the control strategy works well are systems with stable or marginal stable dynamics and free kinematic variables.

It should be noted that Theorem (5.2) represents only sufficient conditions, hence it is possible to have a response where $\mathcal{V}(\xi)$ grows locally but still decays to zero in finite time. In fact there are some versions of stability theorems for hybrid systems that are more general than Theorem (5.2) and allows this type of behavior (in models with a finite number of realisations) (cf. [Carlo et al., 2000]).

5.3.3 Quadratic Performance and Min-Skew-Projection

If we restrict $v_c(\mathbf{x})$ to be a quadratic performance index:

$$v_c(\mathbf{x}) = \mathbf{x}^T \mathbf{Q} \mathbf{x} \tag{5.18}$$

with $\mathbf{Q} > 0$ a positive definite matrix parameter then $\mathcal{V}(\xi)$ also becomes a quadratic performance index:

$$\mathcal{V}(\xi) = \xi^T \begin{bmatrix} \mathbf{Q} & 0 \\ 0 & \mathbf{P} \end{bmatrix} \xi = \xi^T \tilde{\mathbf{P}} \xi \tag{5.19}$$

The controller can now be described as the optimisation problem of choosing the vector field that gives the most rapid decrease of the quadratic performance function:

$$\mathbf{g}_i = \underset{\mathbf{g}_i \in \mho}{\operatorname{argmin}} \ \xi^T \tilde{\mathbf{P}} \mathbf{g}_i(\xi) \tag{5.20}$$

With this formulation the control problem is equivalent to the *min-skew-projection* strategy (MSP) proposed in [Pettersson and Lennartson, 1997]. Sufficient stability conditions are given by the following theorem with proof in [Pettersson and Lennartson, 1997].

Theorem 5.3 (Stability of MSP strategy [Pettersson and Lennartson, 1997]) *If for all states $\xi \in R^{n+m}$:*

$$\exists\, \mathbf{g}_i(\xi) \in \mho \;\; \textit{such that} \;\; \xi^T \tilde{\mathbf{P}} \mathbf{g}(\xi) \leq 0 \tag{5.21}$$

then the closed-loop system is stable using the min-skew-projection *control strategy. Specifically, if for all states $\xi \in R^{n+m}$ there exists a $\gamma > 0$ (independent of ξ) and:*

$$\exists\, \mathbf{g}_i(\xi) \in \mho \;\; \textit{such that} \;\; \xi^T \tilde{\mathbf{P}} \mathbf{g}(\xi) \leq -\frac{1}{2}\gamma ||\xi||^s \tag{5.22}$$

then the closed-loop system is exponentially stable using the min-skew-projection strategy.

With the MSP approach it is understood that the controller selects the vector field $\mathbf{g}_i$ which is the largest projection on the vector $-\tilde{\mathbf{P}}\xi$. This is depicted in Figure (5.3) below.

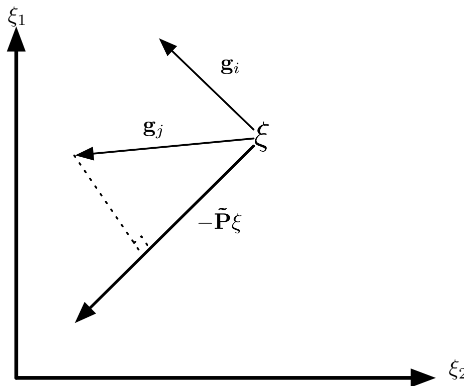


Figure 5.3: The MSP strategy chooses the vector field that is the largest projection on $-\tilde{\mathbf{P}}\xi$, in this case $\mathbf{g}_j$ over $\mathbf{g}_i$.

While the MSP formulation provides more insight into how the controller operates and what the conditions for stability are, it is still up to the user to select parameters $\mathbf{P}$ and $\mathbf{Q}$ such that the conditions of the stability theorem holds.

In other works where a similar control strategy is used, e.g. in [Pettersson and Lennartson, 1997, Rodrigues and How, 2003], it is proposed to search offline for proper parameters for the Lyapunov like functions using a linear-matrix-inequality (LMI) approach to determine either a global Lyapunov like function

(cf. [Pettersson and Lennartson, 1997]) or multiple Lyapunov like functions each valid in some domain (cf. [Rodrigues and How, 2003]).

5.4 Extension to Multiple Objective Control

For many Multiple-Input-Multiple-Output (MIMO) control problems it is often possible to separate the system into a number of loosely coupled subsystems and then design controllers for each subsystem while neglecting the cross couplings, as done in e.g. [Healey and Lienard, 1993] for the AUV model of Chapter 3 on page 29. The following describes how the just described algorithm can be formulated to support this approach.

An extension to multiple objective control is possible in cases where the actuators can be divided in complementary sets assigned to an objective function. For multiple objectives the control objective function will no longer be scalar, but:

$$\mathbf{v}_c(\mathbf{x}) = \begin{cases} v_{c,1}(\mathbf{x}) \\ \vdots \\ v_{c,l}(\mathbf{x}) \end{cases}$$

where each scalar function $v_{c,k}(\mathbf{x})$ is assigned an actuator set a_k such that:

$$\bigcap_{k=1}^{l} a_k = \emptyset$$

For multiple objective control Equation (5.7) and Equation (5.8) are solved independently for each control objective function with appropriate sub-matrices extracted from $\Gamma(\bar{\mathbf{x}}, \bar{\mathbf{u}}, \tau)$, $\mathbf{E}(\bar{\mathbf{x}})$ and $\mathbf{P}$. This way single objective control can be seen as a special case of multiple objective control with just one control objective function to which all actuators are assigned.

5.5 Control Algorithm Summary and Implementation

The controller is implemented as shown on Figure 5.2 by a control class that calculates control input slopes and a number of integrators that keep track of the integrated control signal. Algorithm 5.1 summarises the functionality of the controller class described by its DEVS interface.

$\underline{\delta_{int}()}$:
$\tau \leftarrow$ the least $ta(\cdot)$ in the set of model integrators
evaluate Equation (5.4)
foreach control objective, index k:
· evaluate Equation (5.6)
· **foreach** actuator $i \in a_k$
· $\dot{u}_i \leftarrow$ Equation (5.8)

$\underline{\delta_{ext}(P)}$: P is a set of messages
Receive one or more new matrices from P: $\mathbf{A}, \mathbf{B}, \mathbf{E}$
$newFlag \leftarrow$ **true**

$\underline{\lambda()}$:
return $\{\dot{u}_1, \ldots, \dot{u}_p\}$

$\underline{ta()}$:
if $newFlag$
· $newFlag \leftarrow$ **false**
· **return 0**
return ∞

Algorithm 5.1: The Controller Block

The methods necessary to set-up a controller is briefly presented in the following. At first a controller must be constructed:

```
MinimizingController(DevsContext context, String name,int
                     noControlObjectives, int controlDepth,
                     double[] uDotMax, double[] uDotMin,
                     double controlCost[]);
```

where `context` and `name` provides a context object associated with the DEVS environment and a name for the controller (cf. 2.3 on page 24). `noControl-Objectives` declares the number of control objectives the controller should handle, `controlDepth` declares how many terms in Equation (5.4) should be evaluated, `uDotMax` and `uDotMin` specify limits on the control slopes, and finally `controlCost` specify diagonal values for the control cost matrix $\mathbf{P}$.

The following method is then called to provide the controller with references to integrators in the QSS2 model from which the control horizon, τ, should be derived.

```
void registerStateIntegrators(Qss2Integrator ints[]);
```

Similarly, to keep track of the control states references to the control signals, integrators are provided:

```
void registerInputIntegrators(QssInputIntegrator ints[]);
```

The `QssInputIntegrator` class provides integrators which take the first derivative as input, as opposed to the `Qss2Integrator` class, and updates its output whenever the state has changed by a specified quantum. (i.e. 0-order quantisation as opposed to the first order quantisation of the QSS2 algorithm).

The following method binds control signals, described by their `indexes` in the $\mathbf{u}$ vector, to specific control objectives, `controlObjective`.

```
void registerInputsToControlObjective(int controlObjective,
                                      int[] indexes);
```

This method can also be used during run-time in order to reallocate actuators, e.g. if new actuators are plugged into the system or if an actuator becomes unavailable because of a fault. Finally, if control signal feed-forward (i.e. $\mathbf{u_f}$) is desired the feed forward vector can be set with the following method:

```
void registerInputOperatingPoints(double[] operatingPoint);
```

The control objectives are declared in a `Qss2Map` object (cf. Subsection 3.2.3 on page 36) and is then connected to the controller object using the `addConnection()` method of the `DevsCoordinator` class (cf. Section 2.3 on page 24).

5.6 Control of an Autonomous Underwater Vehicle

This section will provide simulated results for controller evaluation of both the Single Objective Control (SOC) and Multiple Objective Control (MOC) approach applied on the Autonomous Underwater Vehicle (AUV) model introduced in Section 3.4 on page 42. The control objective pursued here is that of depth and heading tracking under constant forward surge speed.

First the MOC approach will be presented, followed by the SOC approach which will use the sum of MOC objective functions as its single objective function. Of the six available actuators the Top-Bottom plane will not be used (it has virtually no influence due to its position and its control cost will be set to infinity).

Actuator saturation limits are as specified previously (cf. Section 3.4 on page 42) and the rate constraints have been set to $\dot{\mathbf{u}}_{max} = -\dot{\mathbf{u}}_{min} = [0.2\ 0.2\ 0.2\ 0.4\ 0\ 1.2]$ with units of rad/s for the rudders and RPM/s for the propeller shaft. A control cost matrix has been found as: $\mathbf{P} = \text{diag}([0.2\ 0.1\ 0.1\ 0.1\ \infty\ 0.001])$. The

input integrators have been setup with quanta of 0.01 rad for the control surfaces and 0.1 rad/s for the propeller shaft.

The AUV system is subject to dynamic dampening due to friction between the AUV and the water. This means that all dynamical states will tend to zero if no control is active, in order to fulfill the control objective of constant forward surge speed and the conditions required for stability of the method, see Section 5.3, it is necessary to include a feed-forward term as introduced in Equation (5.5). In order to move the open-loop equilibrium point of the system to coincide with the control objective.

The simulation case shown in the following graphs is for a combined maneuver where the AUV should dive 20m, while turning 0.9 rad, and increase the surge speed from 1.1m/s to 1.5m/s.

Multiple Objective Controller

For the MOC case the control objectives and associated actuator sets are described in the following. Variables with subscript r are reference values for the corresponding states.

Speed Control objective function for surge speed: $v_1 = 10(u - u_r)^2$ and is assigned the actuator-set: $a_1 = \{n\}$

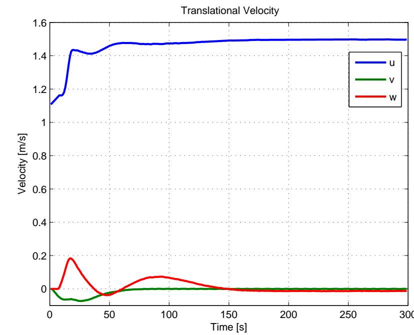
Heading Control objective function for heading: $v_2 = 2(\psi_r - \psi)^2$ and is assigned the actuator-set: $a_2 = \{\delta_r\}$

Depth A control objective function for depth and pitch stabilisation : $v_3 = (z - z_r)^2 + 3(q)^2$ and is assigned the actuator-set: $a_3 = \{\delta_s,\ \delta_{bs},\ \delta_{bp}\}$

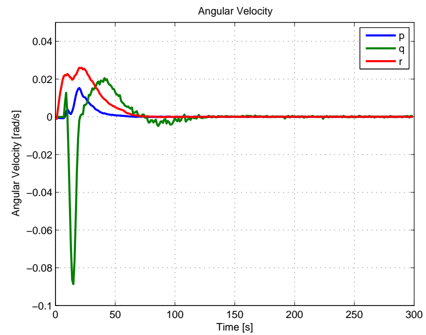
The third control objective is designed to control the depth and at the same time avoid oscillations of the lightly dampened pitch axis of the AUV. With the above choice of control objectives, way point tracking can be accomplished by a guidance controller that supplies new references: u_r, ψ_r, and z_r each time a way-point is reached.

The same decoupling into controllers for speed, heading and depth is seen in [Healey and Lienard, 1993], where single input controllers are designed for linearised system models. Here we retain the non-linear models and can assign multiple actuators to e.g. the depth control objective.

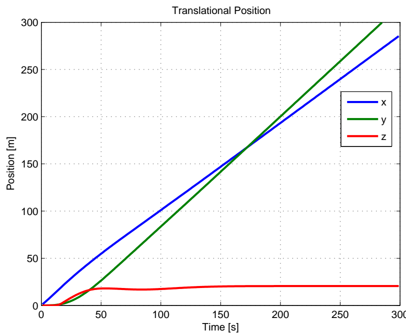
Figure 5.4 shows results for the states of the AUV for the first 300 seconds. It can be seen that the surge speed does stabilise at 1.5 m/s on graph (a), the heading stabilises in roughly 55 s on graph (d), and the depth reaches close to 20 m also in



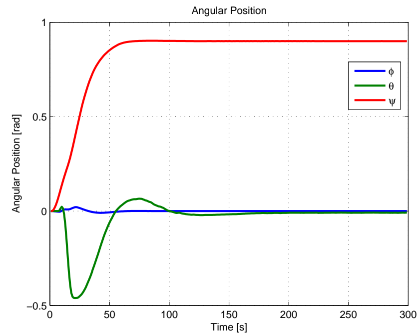
(a) Translational velocity



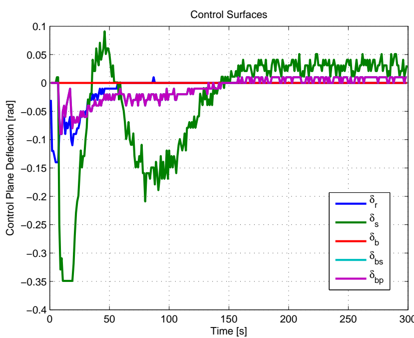
(b) Angular velocity



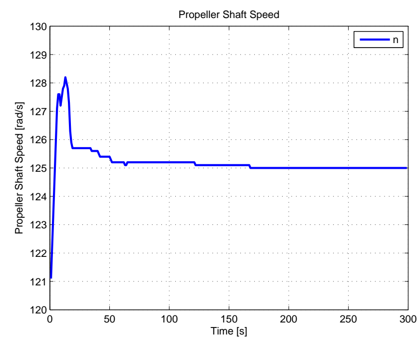
(c) Translational position



(d) Angular Position



(e) Control surface deflections



(f) Propeller shaft speed

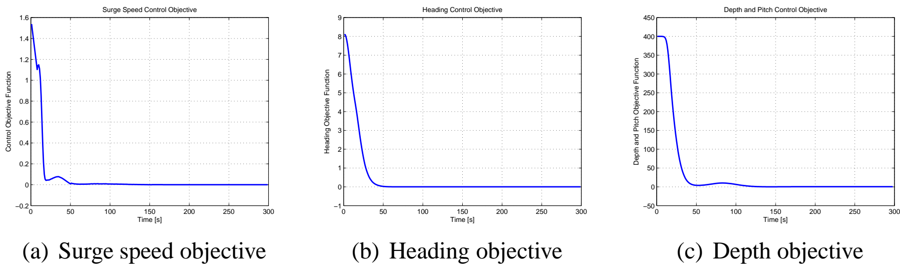
Figure 5.4: MOC results for AUV states and control signals - it can be seen that the control objectives are met.

55 s on graph (c), but it takes further 100 s before it finally settles. The remaining states responds to the maneuver as expected.

From the (e) graph it is seen that initially both the stern plane, δ_s, and the bow planes, δ_{bs} and δ_{bp}, are active in order to perform the dive. The two bow-planes perform the exact same actuation. It can also be seen that the rudder is active at first and then settles at zero as the correct heading is reached. Even after the control objectives have been met the switching nature of the controller can be seen in the control signals, especially for the stern plane.

The (f) graph shows that initially the surge speed controller increases the propeller shaft speed to gain surge speed and then as the heading and depth objectives are met the surge speed controller reduces the shaft speed to the feed-forward operating point of 125 rad/s.

Finally, Figure 5.5 shows the three control objective functions and how they are minimised throughout the simulated execution. It can be seen that both the surge speed controller and depth controller experiences short periods where it is not possible to prohibit growth of the objective function as it is influenced in a positive direction by the autonomous response of the system and the disturbances introduced by the other controllers.



(a) Surge speed objective (b) Heading objective (c) Depth objective

Figure 5.5: MOC results for AUV control objective functions.

Single Objective Controller

Now results for the SOC controller is presented the control objective function is the sum of objective functions for the MOC controller to facilitate comparison:

$$v_{soc}(\mathbf{x}) = v_1(\mathbf{x}) + v_2(\mathbf{x}) + v_3(\mathbf{x}) \tag{5.23}$$

And the set of actuators is:

$$a_{soc} = a_1 \cap a_2 \cap a_3 = \{\delta_r,\ \delta_s,\ \delta_{bs},\ \delta_{bp},\ n\}$$

Figure 5.6 shows results for the states of the AUV under SOC control. Again it can be seen that the states converges to the commanded set-points. However, compared to the MOC case, convergence is not as rapid.

It is clear from both (e) and (f) compared to (e) and (f) on Figure 5.4 that the SOC scheme result in more conservative use of the actuators. This is particularly clear for the stern plane and the rudder.

Finally Figure 5.7 shows the objective function for the SOC controller, i.e. Equation (5.23), and also, for comparison, the sum of the MOC performance functions (Figure 5.5) is plotted on the same graph. Again there is a short period where the objective function is increasing, but compared to the MOC case this period is shorter for SOC.

It can be seen that the MOC approach is somewhat faster than the SOC in achieving the commanded set-points for the controller.

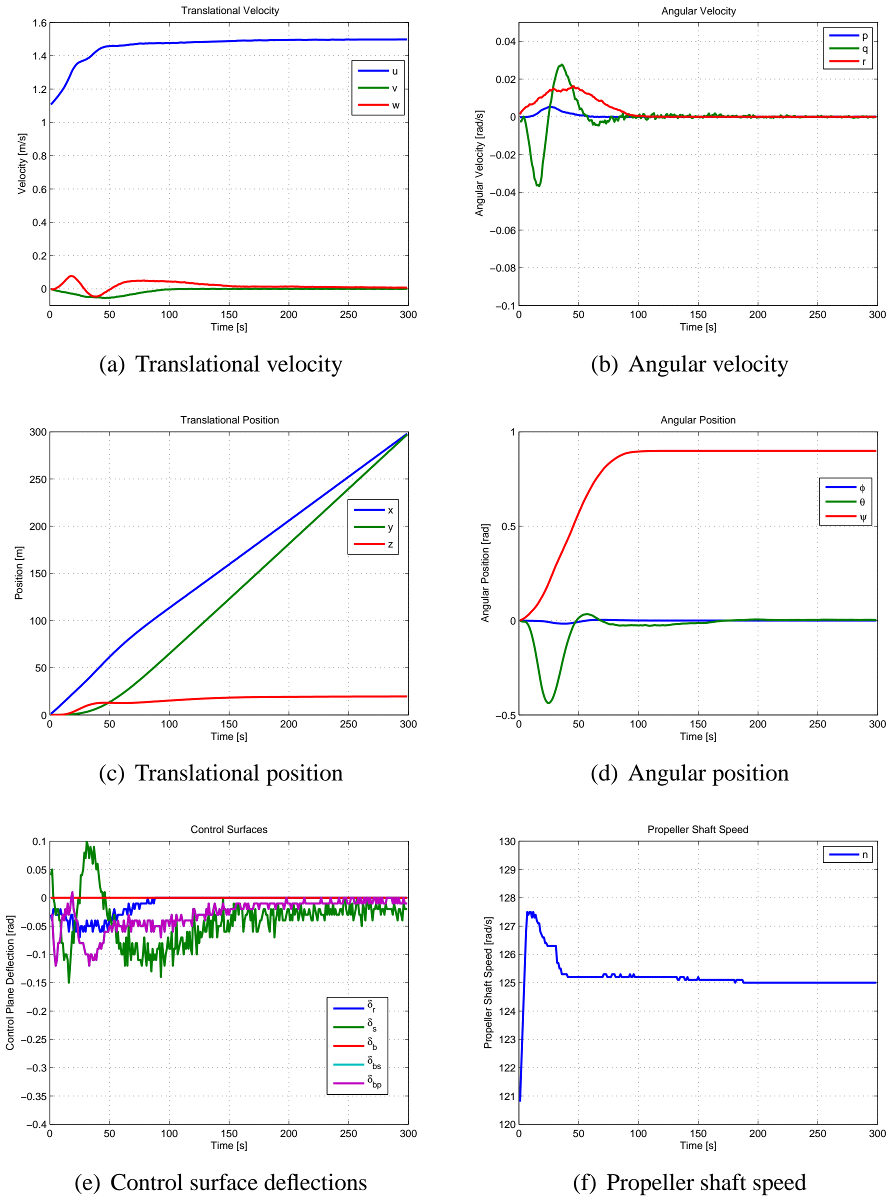
Discussion

Table 5.1 shows the number of control events for each actuator, i.e. how many times the corresponding input integrator has updated its output to the model. It can be seen that the stern-plane actuator is the most utilised for both the SOC and MOC case. In general the MOC case requires slightly fewer updates to perform the control. This is reflected in the execution time which for MOC is 8.47s and 8.63s for SOC.

Events	**SOC**	**MOC**
δ_r	523	385
δ_s	8842	8769
δ_{bp}	1428	1308
δ_{bs}	1428	1308
n	167	135

Table 5.1: Actuator signal updates to the model.

In conclusion it can be said that both algorithms lead to satisfactory performance on the AUV problem. The MOC approach provides slightly better performance and is also slightly faster to execute for the control computer for this problem.



(a) Translational velocity

(b) Angular velocity

(c) Translational position

(d) Angular position

(e) Control surface deflections

(f) Propeller shaft speed

Figure 5.6: SOC results for AUV states and control signals - it can be seen that control objectives are met.

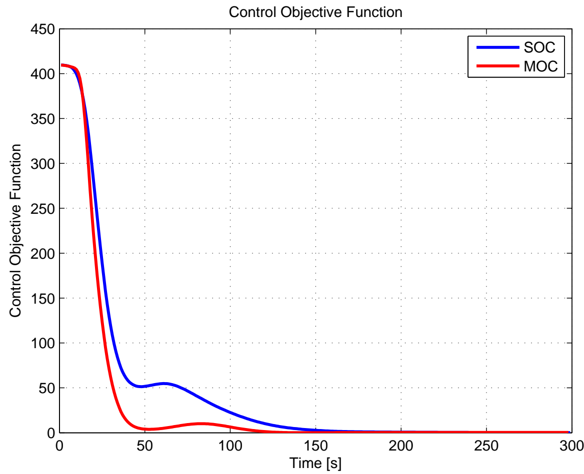


Figure 5.7: SOC objective function vs. MOC summed objective functions - It can be seen that the MOC approach is more effective.

5.7 Chapter Summary

This chapter introduced two algorithms for control of a class of non-linear multiple-input-multiple-output systems based on QSS2 models of the system and a QSS2 description of a control objective function which is minimised by the choice of control input slopes by the controller.

It was shown that stability of the proposed method must be viewed in the framework of switched/hybrid systems and depends on the users choice of control objective function and control cost matrix. Further, it was shown that if a quadratic control objective function is chosen the control strategy is equivalent to the *min-skew-projection* strategy described in [Pettersson and Lennartson, 1997].

In practice the method can be applied to a large number of systems, e.g. motion control system with inherent dynamical dampening, where the control objective is to guide the kinematical states to a given set-point.

The method was demonstrated using simulations of an autonomous underwater vehicle, based on a nominal model and full state knowledge, and it was demonstrated that the method was successful in controlling the system. Both the single objective and multiple objective control variant of the method were demonstrated successfully.

6 Sliding Mode Control in QSS Systems

This chapter contributes with a highly configurable control structure based on sliding mode control theory, which is developed for composition with quantised state models and offers adaptability by being implemented as a number of object oriented components that can be replaced or extended individually. Simulation results are presented for a case involving a deep-space probe.

6.1 Introduction

The concept of Sliding Mode Control (SMC) originates in the theory of variable structure systems (Cf. [Utkin, 1977] for an early survey paper). The idea is to have a discontinuous input which drives the system towards a manifold, called a sliding surface, which defines the desired system dynamics. One great advantage of the SMC approach is that on the sliding surface the state evolution is independent of the system model and can be made robust against bounded uncertainties. On the other hand SMC suffers from a phenomena called *chattering*, due to the discontinuous nature of the feedback.

Sliding mode control has been applied for many applications, e.g. autonomous underwater vehicles [Healey and Lienard, 1993] and space vehicle motion control [Wiesniewski, 1998]. This chapter introduces a SMC stabilisation scheme for Multiple-Input-Multiple-Output (MIMO) systems, which is based on the design procedure introduced in [Khalil, 2000], and develops an implementation of this approach using quantised state systems and DEVS based simulation/execution. The main contribution of the chapter is a generic SMC controller for quantised systems that easily can be re-used for various applications by specifying the relevant functions describing the plant to be controlled.

Section 6.2 describes the SMC approach for continuous systems and discusses

relevant properties, the subsequent section, Section 6.3, describes how this approach was adopted to the QSS/DEVS framework, and, finally, Section 6.4 provides simulation results of a deep space probe using a nominal model with full state knowledge. The next chapter provides results including the effects of uncertainty and running an estimator in the loop.

6.2 Sliding Mode Stabilisation of MIMO Systems

The following introduces and analyses the SMC stabilisation approach for non-linear MIMO systems introduced in [Khalil, 2000]. The starting point is a control affine system on regular form representing a nominal model with the origin ($\eta = 0$ and $\xi = 0$) being an open-loop equilibrium point.

$$\begin{aligned} \dot{\eta} &= \mathbf{f_a}(\eta, \xi) \\ \dot{\xi} &= \mathbf{f_b}(\eta, \xi) + \mathbf{G}(\eta, \xi)\mathbf{u} \end{aligned}$$

where $\xi \in \mathbb{R}^d$ represent the dynamical states, $\eta \in \mathbb{R}^d$ represent the kinematical states, $\mathbf{u} \in \mathbb{R}^d$ are control inputs, $\mathbf{f_b}(\eta, \xi)$ and $\mathbf{f_a}(\eta, \xi)$ function maps, and $\mathbf{G}(\eta, \xi) \in \mathbb{R}^{d \times d}$ a non-singular matrix, and d is equal to the number of degrees of freedom in the system.

We consider an extension of the previous system which includes dynamic disturbances and parameter uncertainties in $\mathbf{G}(\cdot)$:

$$\dot{\eta} = \mathbf{f_a}(\eta, \xi) \tag{6.1}$$

$$\dot{\xi} = \mathbf{f_b}(\eta, \xi) + \mathbf{G}(\eta, \xi)\mathbf{E}(\eta, \xi)\mathbf{u} + \delta(\eta, \xi, \mathbf{u}, t) \tag{6.2}$$

where $\mathbf{E}(\eta, \xi) \in \mathbb{R}^d$ is a diagonal matrix of strictly positive diagonal elements $e_i(\eta, \xi) > 0$ for $i = 1..d$ representing parameter uncertainties of $\mathbf{G}(\eta, \xi)$ and is equal to the identity matrix when there are no uncertainties. The vector $\delta(\eta, \xi, t)$ describes dynamic disturbances, it is important to note that these disturbances are matched, which means that $\delta(\eta, \xi, t)$ is in the column-space of $\mathbf{G}(\eta, \xi)$.

Associated with the system is a *sliding variable* expressed as:

$$\mathbf{s} = \xi - \phi(\eta) \tag{6.3}$$

where $\phi(\eta)$ is a at least once differentiable *guidance control law*, i.e. a feedback that stabilises Equation (6.1) with $\xi = \phi(\eta)$ as input. Equation (6.3) implicitly

defines the *sliding manifold*:

$$S = \{\phi(\eta),\ \xi\ :\ \mathbf{s} = \mathbf{0}\}$$

It is the goal of the sliding mode controller to reach the sliding manifold and remain there, even in the presence of disturbances and uncertainties as described by Equation (6.1) and Equation (6.2). The design of the guidance controller, $\phi(\eta)$, is not addressed in the SMC design that follows, but is simply assumed available. It could e.g. be implemented by negative proportional error signal feedback or some other suitable control methodology.

In order to describe the motion on the sliding mode we differentiate Equation (6.3) once and obtain:

$$\dot{\mathbf{s}} = \mathbf{f_b}(\eta, \xi) - \frac{\partial \phi}{\partial \eta}\mathbf{f_a}(\eta, \xi) + \mathbf{G}(\eta, \xi)\mathbf{E}(\eta, \xi)\mathbf{u} + \delta(\eta, \xi, \mathbf{u}, t) \tag{6.4}$$

When we assume a known nominal model $\hat{\mathbf{E}} = \mathbf{I_d}$ for $\mathbf{E}$ and no disturbances, i.e. $\delta = \mathbf{0}$ then it is clear that the following input makes $\dot{\mathbf{s}}$ equal to zero and maintains the system on the sliding manifold.

$$\mathbf{u} = \left[\mathbf{G}(\eta, \xi)\hat{\mathbf{E}}(\eta, \xi)\right]^{-1}\left(-\mathbf{f_b}(\eta, \xi) + \frac{\partial \phi}{\partial \eta}\mathbf{f_a}(\eta, \xi)\right) \tag{6.5}$$

The first term is responsible for cancelling dynamic forces of the system and the second term ensures that the system tracks changes in the sliding manifold. This control is also dubbed *equivalent control* [Bandyopadhyay and Sivaramakrishnan, 2006].

To ensure that the system reaches the sliding manifold the previous control law is augmented with a term to guarantee stabilisation to the manifold:

$$\mathbf{u} = \left[\mathbf{G}(\eta, \xi)\hat{\mathbf{E}}(\eta, \xi)\right]^{-1}\left(-\mathbf{f_b}(\eta, \xi) + \frac{\partial \phi}{\partial \eta}\mathbf{f_a}(\eta, \xi)\right) + \mathbf{G}^{-1}(\eta, \xi)\mathbf{v} \tag{6.6}$$

where $\mathbf{v}$ is a switching element characteristic for sliding mode control, also dubbed the *reaching law*. The following analysis will show how the reaching law can be designed to ensure that the sliding mode is reached under the presence of disturbances and uncertainties.

When Equation (6.6) is substituted into Equation (6.4) we obtain the expression for each element, $s_i \in \mathbf{S}$:

$$\dot{s}_i = e_i(\eta, \xi)v_i + \Delta_i(\eta, \xi, \mathbf{v}, t) \tag{6.7}$$

where the first term arises due to the reaching law and the second term due to the disturbances and uncertainties, in more detail:

$$\begin{aligned}
&\Delta(\eta, \xi, \mathbf{v}, t) = \\
&\delta\left(\eta, \xi, \mathbf{G}^{-1}(\eta, \xi)\hat{\mathbf{E}}^{-1}(\eta, \xi)\left(-\mathbf{f_b}(\eta, \xi) + \frac{\partial \phi}{\partial \eta}\mathbf{f_a}(\eta, \xi)\right) + \mathbf{G}^{-1}(\eta, \xi)\mathbf{v}, t\right) \\
&+ \left[\mathbf{I}_{d\times d} - \mathbf{E}(\eta, \xi)\hat{\mathbf{E}}^{-1}(\eta, \xi)\right]\left(\mathbf{f_b}(\eta, \xi) - \frac{\partial \phi}{\partial \eta}\mathbf{f_a}(\eta, \xi)\right)
\end{aligned}$$

here the first term represents dynamic disturbances and the second term represents the error in cancelling the dynamic forces and tracking the sliding mode which is due to uncertain knowledge of $\mathbf{G}(\eta, \xi)$.

We assume that the magnitude of the uncertainties and disturbances can be estimated by a continuous function and a constant such that the following inequality holds:

$$\left|\frac{\Delta_i(\eta, \xi, \mathbf{v}, t)}{e_i(\eta, \xi)}\right| \leq \varrho(\eta, \xi) + \kappa_0||\mathbf{v}||_\infty \;\; \forall 1 \leq i \leq d \tag{6.8}$$

where $\varrho(\eta, \xi) \geq 0$ and $\kappa_0 = [0, 1)$.

If we consider the following Lyapunov function candidate for each sliding variable, $s_i \in \mathbf{S}$:

$$V_i = \frac{1}{2}s_i^2 \quad \text{with} \quad \dot{V}_i = s_i\dot{s}_i$$

and insert Equation (6.7) into the expression for $\dot{V}_i$, we obtain:

$$\dot{V}_i = s_i e_i(\eta, \xi)v_i + s_i\Delta_i(\eta, \xi, \mathbf{v}, t)$$

inserting the uncertainty bound, i.e. Equation (6.8), we get:

$$\dot{V}_i \leq e_i(\eta, \xi)\left(s_i v_i + |s_i|[\varrho(\eta, \xi) + \kappa_0||\mathbf{v}||_\infty]\right) \tag{6.9}$$

hence to ensure negativity of $\dot{V}_i$, an input v_i can be chosen as:

$$v_i = -\beta(\eta, \xi)\,\mathrm{sgn}(s_i) \;\; \forall 1 \leq i \leq d \tag{6.10}$$

where the $\beta(\eta,\xi)$ function is defined as:

$$\beta(\eta,\xi) \geq \frac{\varrho(\eta,\xi)}{1-\kappa_0} + \beta_0$$

Now, inserting Equation (6.10) into Equation (6.9) and manipulating terms, we get:

$$\dot{V}_i \leq -e_i(\eta,\xi)\beta_0(1-\kappa_0)|s_i| \tag{6.11}$$

It is therefore obvious that the sliding mode controller using the control law proposed in Equation (6.10) provides global asymptotically stability for the uncertain/disturbed system.

6.2.1 Chattering and Boundary Layer Control

The previous section showed the potential of the SMC method in terms of its strong stability properties. However, the requirement of a discontinuous switching input signal is not attractive from an implementation point of view. Often the discontinuous switching can result in a phenomena known as *chattering*, consider Figure 6.1.

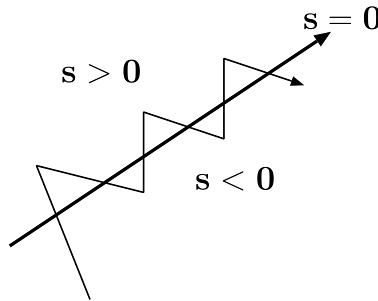


Figure 6.1: Illustration of the chattering phenomena.

In theory once the trajectory crosses the sliding manifold on the figure it should remain there, however the theory assumes that input switching can occur infinitely fast. This is not practical possible in implementations of SMC and the result is that the sliding manifold is "overshot" and a zig-zagging motion criss-crossing the manifold follows. This is clearly not desirable as it wears on the

actuators. Therefore, in applications, the switching input component is often approximated using a continuous function, e.g. by applying:

$$v_i = -\beta(\eta,\xi)\,\mathrm{sat}\left(\frac{s_i}{\epsilon_i}\right) \tag{6.12}$$

where $\mathrm{sat}(x)$ is a function approximating $\mathrm{sgn}(x)$ as:

$$\mathrm{sat}(x) = \begin{cases} x & \text{if } |x| \leq 1 \\ \mathrm{sgn}(x) & \text{if } |x| > 1 \end{cases} \tag{6.13}$$

and where $\epsilon_i > 0$ is a parameter to control the relative slope of the approximation, i.e. choosing a large ϵ_i reduces the gain close to $s_i = 0$ and a small value for ϵ_i gives a response very similar to the original $\mathrm{sgn}(\cdot)$ function. Another commonly used alternative for an $\mathrm{sgn}(x)$ approximating function is $\tanh(x)$.

The use of a continuous approximation for the switching element will have consequences for the stability properties derived in the previous subsection. These consequences will be analysed in the following for the $\mathrm{sat}(x)$ function. By inserting Equation (6.12) into Equation (6.9) we get:

$$\dot{V}_i \leq e_i(\eta,\xi)\left(-\beta(\eta,\xi)s_i\,\mathrm{sat}\left(\frac{s_i}{\epsilon_i}\right) + \varrho(\eta,\xi)|s_i| + \kappa_0\beta(\eta,\xi)|s_i|\right) \tag{6.14}$$

where from it can be seen that whenever $s_i > \epsilon$ then $\dot{V}_i$ is negative and equivalent to Equation (6.11). This means that s_i is uniformly ultimately bounded to the set $\{|s_i| \leq \epsilon, 1 \leq i \leq d\}$ called the *boundary layer*. To analyse the effect of the boundary layer on the stability of the system we first need to define class $\mathcal{K}$ functions.

Definition 6.1 (Class $\mathcal{K}$ function [Khalil, 2000])
A continuous function $\alpha : [0,a) \rightarrow [0,\infty)$ is said to belong to class $\mathcal{K}$ if it is strictly increasing and $\alpha(0) = 0$.

We assume that there exists a continuesly differentiable Lyapunov function, $V(\eta)$, associated with the sliding manifold design $\xi = \phi(\eta)$ and that there exist two class $\mathcal{K}$ functions α_1 and α_2 such that the following inequalities hold:

$$\alpha_1(||\eta||) \leq V(\eta) \leq \alpha_2(||\eta||) \tag{6.15}$$

and for the Lyapunov derivative with α_3 and γ being class $\mathcal{K}$ functions:

$$\frac{\partial V}{\partial \eta}\mathbf{f_a}(\eta,\phi(\eta+\mathbf{s})) \leq -\alpha_3(||\eta||) \quad \forall\, ||\eta|| \geq \gamma(||s||) \tag{6.16}$$

It can be seen that for some constant c the following is implied:

$$\begin{aligned} |s_i| &\leq c \;\text{ for } 1 \leq i \leq d \Rightarrow ||s|| \leq k_1 c \Rightarrow \\ \dot{V} &\leq -\alpha_3(||\eta||) \;\text{ for } ||\eta|| \geq \gamma(k_1 c) \end{aligned}$$

where k_1 is a norm dependent positive constant. We define another class $\mathcal{K}$ function α such that:

$$\alpha(r) = \alpha_2(\gamma(k_1 r))$$

and realise the following implications:

$$\begin{aligned} V(\eta) \geq \alpha_3(c) \quad &\Rightarrow \quad V(\eta) \geq \alpha_2(\gamma(k_1 c)) \\ &\Rightarrow \quad ||\eta|| \geq \gamma(k_1 c) \\ &\Rightarrow \quad \dot{V} \leq -\alpha_3(||\eta||) \leq -\alpha_3(\gamma(k_1 c)) \end{aligned} \tag{6.17}$$

from which it is clear that the set $\{V(\eta) \leq c_0 \;\text{ for } c_0 \geq \alpha(c)\}$ is positively invariant as $\dot{V}$ is negative on the boundary $V(\eta) = c_0$. Therefore if we define the set:

$$\Omega = \{V(\eta) \leq c_0\} \times \{|s_i| \leq c, \; 1 \leq i \leq d\}$$

it is clear that this set is positively invariant when we have $c > \epsilon$ and that all trajectories with initial state in Ω are bounded for $t \geq 0$.

From Equation (6.14) we know that after some finite time then $|s_i(t)| \leq \epsilon$ and therefore from Equation (6.17) that:

$$\dot{V} \leq -\alpha_3(\gamma(k_1 \epsilon)) \;\; \forall \, V(\eta) \geq \alpha(\epsilon)$$

It therefore follows that in finite time any initial trajectory will reach the positively invariant set defined by:

$$\Omega_\epsilon = \{V(\eta) \leq \alpha(\epsilon)\} \times \{|s_i| \leq \epsilon, \; 1 \leq i \leq d\}$$

Which proves that when a boundary layer is introduced the system looses asymptotically stability, but instead provides ultimately uniformly boundedness for trajectories to the set Ω_ϵ which can be made arbitrarily small by the choice of the parameter ϵ_i for the switching approximation, see Equation (6.12).

6.2.2 Reaching Laws

When applying the control law of Equation (6.10) or its continuous approximation Equation (6.12) the approach towards the sliding manifold will be at a constant rate determined by the magnitude of $\beta(\eta, \xi)$. If a different reaching response is desired then feedback law can be extended to provide more flexibility, consider for example the following reaching control law (based on [Bandyopadhyay and Sivaramakrishnan, 2006]):

$$v_i = -k_i \left(\frac{s_i}{l_i}\right)^2 - q_i s_i - \beta(\eta, \xi) \tanh\left(\frac{s_i}{\epsilon_i}\right) \tag{6.18}$$

where l, k, q, ϵ are positive design parameters. The third term is responsible for disturbance rejection, the second term provide optional proportional feedback, and the first term provide optional feedback on the squared sliding variable providing fast reaction to large errors.

It is the fact that the SMC approach reduces the design problem to designing a reaching controller for the sliding mode that makes it desirable, since the challenges associated with the non-linear dynamics of the plant is accounted for by the equivalent control introduced in Equation (6.5).

6.3 A QSS2 Implementation of Sliding Mode Control

This section contributes with a quantised state systems implementation of the SMC approach described in the previous section. The benefit is a generic software package that can be used to provide stabilising control for a large class of systems, given that the user can supply: a nominal model, uncertainty bounds, and parameters to shape the control performance in terms of reaching the sliding manifold.

At first the control structure is presented, where after design of a guidance controller based on convex performance criteria for the kinematical states is developed. Then the SMC controller for the dynamical system is developed and finally a module to dynamically adjust the parameters of the controller to local uncertainties is presented in Subsection 6.3.5 on page 100. The next section will provide simulation results to discuss performance.

6.3.1 Control Structure

The controller structure is depicted in Figure 6.2, where each block represents a software object/class in the implementation of the controller.

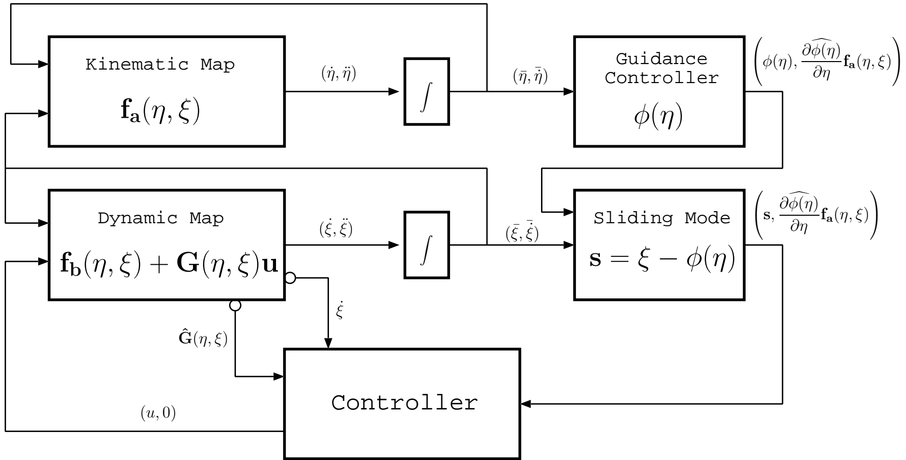


Figure 6.2: Sliding mode controller structure. Simple example with $d = 1$.

The `Kinematic` and `Dynamic map`, as well as the integrator blocks corresponds to the QSS2 software components described in Chapter 3 on page 29. This means that the user must supply a QSS2 model which is split into the two maps.

The `Guidance Controller` block provides the vector $\phi(\eta)$ that defines the time varying sliding manifold, the implementation of this block is described in the next subsection. The `Sliding Mode` block implements the sliding mode equation, i.e. Equation (6.3), and the `Controller` block is responsible for implementing the equivalent control law and the reaching control law, details will be given in the following. The final subsection of this section will describe how the presented control structure is adapted to situations where the user is able to supply state-dependent uncertainty/disturbance bounds.

6.3.2 Guidance Controller

The guidance controller must provide a feedback $\phi(\eta)$ such that:

$$\dot{\eta} = \mathbf{f_a}(\eta, \phi(\eta))$$

is stabilised at the desired end-point. Inspired by the controller presented in the previous chapter we associate with the controller a scalar valued convex objective function with global minimum in the desired reference η_r:

$$v = \mathbf{v}(\eta) \;\text{ with }\; \mathbf{v}(\eta_r) = \min_{\eta \in \mathbb{R}^d} \mathbf{v}(\eta)$$

with $\mathbf{v}(\eta) : \mathbb{R}^d \mapsto \mathbb{R}^1$. A sufficient condition for minimising this function over time, from convex optimisation theory [Boyd and Vandenberghe, 2004], is to always move in the direction of the negative gradient, therefore:

$$\phi(\eta) = -\nabla \mathbf{v}(\eta)$$

By implementing the objective function using the already developed `Qss2-Map` class, cf. Subsection 3.2.3 on page 36, the gradient information for the function is already available (cf. Equation (3.2.2 on page 34)). Hence the guidance controller can be implemented by inheriting the `Qss2Map` class and overriding the `output()` method to output the negative gradient vector rather than the function value of $\mathbf{v}(\eta)$. This new class is called `NegativeGradient`.

However, the class must also provide $\frac{\partial \phi(\eta)}{\partial \eta}\mathbf{f_a}(\eta, \phi(\eta))$ as an output for use in the calculation of the equivalent control. This information is not directly available from the `Qss2Map` class. However, by requiring $\mathbf{v}(\eta)$ to be at least twice differentiable a good numerical approximation can be obtained from:

$$\widehat{\frac{\partial \phi(\eta)}{\partial \eta}}\mathbf{f_a}(\eta, \phi(\eta)) = \frac{\nabla \mathbf{v}_k(\eta) - \nabla \mathbf{v}_{k-1}(\eta)}{\Delta \eta} \tag{6.19}$$

The domain over which this backward difference estimate is determined by $\boldsymbol{\Delta}\eta$ which in turn is determined by the quantum selected by the user for the kinematical states. Equation (6.19) is implemented in the `NegativeGradient` class.

The guidance controller presented here is just one opportunity; other classes which comply to the same interface in terms of input and output signals can be implemented and plugged into the controller structure of Figure 6.2.

6.3.3 Sliding Mode

The sliding mode block accepts the working point QSS2 trajectories $(\bar{\xi}, \bar{\dot{\xi}})$ as inputs, as well as the outputs of the guidance controller. The class implements the sliding mode equation (see Equation (6.3)) by propagating the dynamic trajectories:

$$\mathbf{s} = \left(\bar{\xi} + \bar{\dot{\xi}}\Delta\mathbf{t}\right) - \left(\phi(\eta) + \widehat{\frac{\partial\phi(\eta)}{\partial\eta}}\mathbf{f_a}(\eta, \phi(\eta))\Delta t_\phi\right)$$

where $\Delta\mathbf{t}$ is a vector of time-lapses since the last output event was received from the corresponding integrator in the model, and Δt_ϕ is the time-lapse since the last time input was received from the guidance controller.

The sliding mode block outputs the sliding variable $\mathbf{s}$ and it passes on the $\widehat{\frac{\partial\phi(\eta)}{\partial\eta}}\mathbf{f_a}(\eta, \phi(\eta))$ output from the guidance controller, which is also to be used in the sliding mode control calculation.

The above described functionality is implemented in the class called `SlidingMode` and can also be replaced with other classes if a different definition of the sliding mode than Equation (6.3) is desired in the controller structure.

6.3.4 Sliding Mode Controller

The controller accepts as input the sliding variable $\mathbf{s}$ and the change in the guidance law $\widehat{\frac{\partial\phi(\eta)}{\partial\eta}}\mathbf{f_a}(\eta, \phi(\eta))$ as inputs, and can access the input sensitivity matrix $\widehat{\mathbf{G}(\bar{\eta}, \bar{\xi})}$ and the most recent force vector $\dot{\xi}$ from the `DynamicMap`.

Whenever new input is received the controller pulls the most recent information from the `DynamicMap` and then calculates the equivalent control:

$$\mathbf{u}_e = \widehat{\mathbf{G}}^{-1}(\bar{\eta}, \bar{\xi})\left(-\widehat{\mathbf{f}}_\mathbf{b}(\bar{\eta}, \bar{\xi}) + \widehat{\frac{\partial\phi}{\partial\eta}}\mathbf{f_a}(\eta, \xi)\right)$$

where $\widehat{\mathbf{f}}_\mathbf{b}(\bar{\eta}, \bar{\xi})$ is obtained by:

$$\widehat{\mathbf{f}}_\mathbf{b}(\bar{\eta}, \bar{\xi}) = \dot{\xi} - \widehat{\mathbf{G}}(\bar{\eta}, \bar{\xi})\mathbf{u}^*$$

where $\mathbf{u}^*$ is the input calculated in the previous control calculation.

The reaching law is implemented as:

$$v_i = -k_i \left(\frac{s_i}{l_i}\right)^2 - q_i s_i - \beta_s \,\text{switch}\,(s_i) \tag{6.20}$$

where l_i, k_i, q_i are positive design parameters supplied by the user, β_s is a user supplied positive constant such that:

$$\beta_s \geq \beta(\eta, \xi) \geq \frac{\varrho(\eta, \xi)}{1 - \kappa_0} + \beta_0 \tag{6.21}$$

and finally $\text{switch}(\cdot)$ is a user defined class which implements the desired approximation of the term $\text{sgn}(s_i)$ for example $\tanh\left(\frac{s_i}{\epsilon_i}\right)$. User supplied classes must adhere to the `BoundaryLayer` interface which specifies a single method:

```
double evaluate(int index, Matrix s);
```

If no class is specified $\text{sgn}(s_i)$ is used as default. From the equivalent control and the reaching law the combined control is then calculated and applied to the system:

$$\mathbf{u} = \mathbf{u}_e + \widehat{\mathbf{G}}^{-1}(\bar{\eta}, \bar{\xi})\mathbf{v}$$

Saturation limits can also be specified for the control inputs. To summarise; the `Controller` class provides many parameters for the user to set in order to tune the performance of the controller to each specific problem. The different methods to do that is briefly listed below:

```
void setSquareGain(double k[]);
void setSquareRegion(double l[]);
void setProportionalGain(double q[]);
void setBoundaryLayer(BoundaryLayer switch);
void setSaturation(double sat[]);
void setStaticBeta(double beta_s);
```

The output of the controller, as is indicated on Figure 6.2, is kept constant between control calculations, i.e. $\dot{u} = 0$ in the output double which is provided to the Dynamic Map object (a `Qss2Map` class).

6.3.5 Dynamic Disturbance Bounds Calculation

The controller presented in the previous subsection only considered static uncertainty bounds, see Equation (6.21). This subsection describes a software class

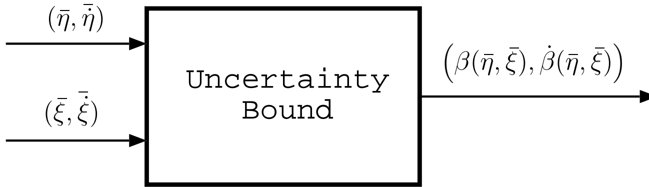


Figure 6.3: Adding dynamic uncertainty bounds to the controller structure.

that can be augmented to the presented controller structure in order to provide dynamic uncertainty bounds. Consider Figure 6.3.

By inserting such a block into the controller structure, see Figure 6.2, with output signal routed to an optional input on the `Sliding Controller` block the value for β used in the feedback calculation in Equation (6.20) can be updated dynamically.

The `UncertaintyBound` class was implemented by inheriting the `Qss2-Map` class and specifying a single output. In this way the user can specify a function $\beta(\cdot) : \mathbb{R}^{2d} \mapsto \mathbb{R}^1$ which specify the uncertainty bound. The uncertainty bound provided by the user must both apply for the dynamic disturbances $\delta(\eta, \xi)$ and uncertainty in input gain $\mathbf{E}(\eta, \xi)$ as indicated by Equation (6.8).

One can easily provide custom classes for uncertainty bound calculation with different functionality than the above if required for the specific application and insert it into the control structure, for example if one wishes to implement an adaptive uncertainty bound estimator based on monitoring performance of the control system

6.4 Simulation Results for a Deep Space Probe

In this section a simulation study of the proposed control scheme will be presented which concerns the *three-axis* mode of the Deep Space Probe (DSP) presented in Chapter 4 on page 51, cf. Figure 4.4 on page 60. Results will be for a nominal model; the objective is to describe how the SMC approach works under ideal conditions. The next chapter will provide simulation results including estimation errors, disturbances and model uncertainties in order to shed light on controller performance under non-ideal circumstances.

Referring to the kinematical model of the DSP, see Equation (4.11 on page 60), the kinematical model for the SMC controller is identified as:

$$\dot{\eta} = \frac{1}{\cos\theta_2}\begin{bmatrix} \cos\theta_2 & \sin\theta_1\sin\theta_2 & \cos\theta_1\sin\theta_2 \\ 0 & \cos\theta_1\cos\theta_2 & -\sin\theta_1\cos\theta_2 \\ 0 & \sin\theta_1 & \cos\theta_1 \end{bmatrix}\omega \text{ with } \theta = \eta,\ \omega = \xi$$

The dynamical model of the DSP without actuators is given in Equation (4.12 on page 60). We augment this model with a model of six thrusters which can provide thrust in all directions of up to 2 N with a lever arm of 50 cm. We currently assume that there are no quantisation or minimum impulse bits to consider for the thrusters, and the model becomes:

$$\dot{\xi} = \mathbf{J}^{-1}\left(-[\omega\times]\mathbf{J}\omega + \frac{1}{2}\mathbf{u}\right) \text{ with } \omega = \xi \tag{6.22}$$

Each element in the input vector $\mathbf{u} \in \mathbb{R}^3$ controls two complementary thrusters of which only one is active at a time depending on the sign of the corresponding input signal.

The QSS2 model of the DSP developed for the EKF algorithm in Chapter 4 on page 51 is reused with the updated dynamical model of Equation (6.22). If not stated otherwise the quanta used in the following simulations are 10^{-5} rad for attitude states and 10^{-7} rad/s for angular rate states.

We define the control objective to be attitude stabilisation to specified set-points $\eta_r = [\eta_{r,1}\ \eta_{r,2}\ \eta_{r,3}]^T$ and represent this control objective in the following objective function for the guidance controller:

$$v = (\eta_1 - \eta_{r,1})^2 + (\eta_2 - \eta_{r,2})^2 + (\eta_3 - \eta_{r,3})^2$$

The following subsection will show control performance under varying choices of SMC control parameters and discuss achieved performance.

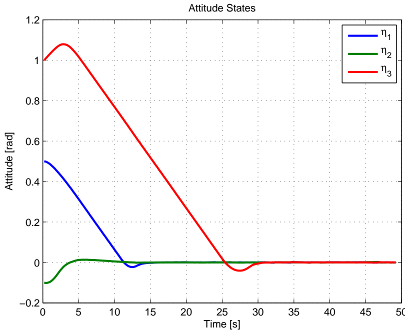
6.4.1 Results without Boundary Layer Control

Initially we will consider the simplest SMC controller possible by applying the reaching law of Equation (6.20) with parameters:

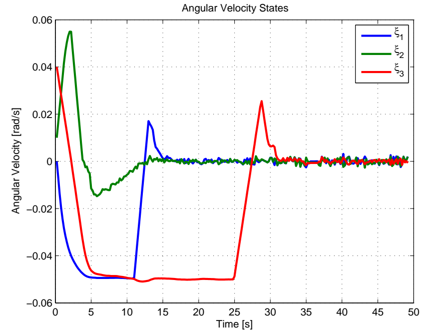
$$k_i = q_i = 0,\ \ l_i = \epsilon_i = 1 \text{ for } 1 \geq i \geq 3 \tag{6.23}$$

$$\text{switch} = \text{sgn}(\cdot), \quad \beta(\eta, \xi) = 0.05$$

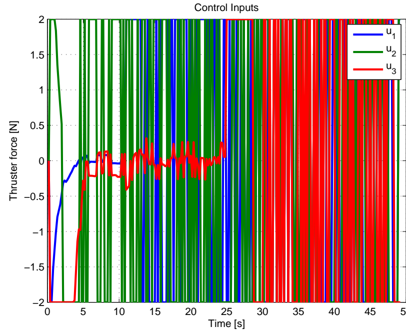
Simulation results are given in Figure 6.4 with $\eta_r = [0\ 0\ 0]^T$ and initial attitude $\eta_0 = [0.5 \quad -0.1 \quad 1]^T$. Looking a the attitude states it is clear that the controller reaches the reference, although with a little initial overshoot, and maintains it there.



(a) Attitude response



(b) Angular velocities



(c) Control input signal

Figure 6.4: Simple SMC control results, a large degree of chattering is evident.

From the angular velocities (b) it can be seen that the controller drives the angular velocities towards a constant value that is maintained until the corresponding attitude state gets close to convergence. This behaviour is also evident from the attitude states which approaches their reference at a near constant rate.

The (c) graph, however, shows that the controller exhibits chattering to a high

degree, where the control input switches between its saturation points when each state reaches its reference. This switching makes the model very slow to execute, approximately 18 s, as the QSS2 model switches operating points very often. From a propellant consumption point of view this behaviour is also not desirable.

The switching is due to the bounded final error region of the QSS2 approach to propagating differential equations, see Section 3.3 on page 38. Effectively, the input variables u_i is operating with a quantum of twice the saturation limit for each variable. As will be seen in the following simulations performance can be improved significantly if approximations to the $\mathrm{sgn}(\cdot)$ function is used in the reaching law.

6.4.2 Results with Boundary Layer Control

To investigate if better performance is obtained by switching to an approximation of the switching component all control parameters are maintained except the change to $\tanh(s_i/\epsilon_i)$ as an approximation for the switching component, the parameters are:

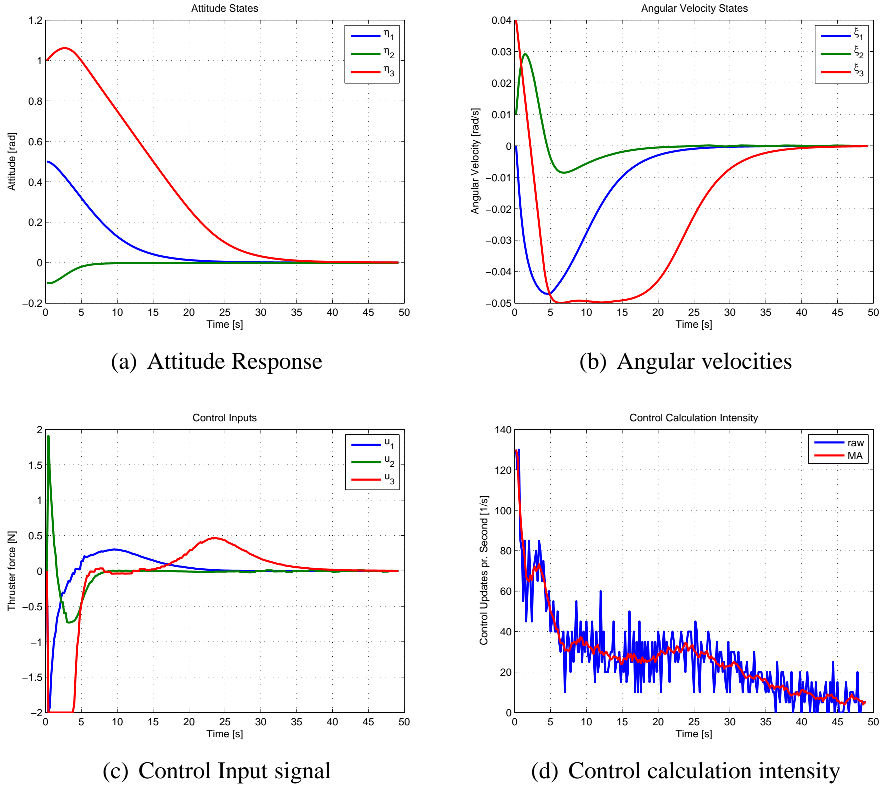
$$\begin{aligned} k_i = q_i = 0, \;\; l_i = 1, \;\; \epsilon_i = 0.2 \;\; \text{for } 1 \geq i \geq 3 \\ \text{switch} = \tanh(\cdot), \;\; \beta(\eta, \xi) = 0.05 \end{aligned} \tag{6.24}$$

Simulated responses are shown on Figure 6.5 with $\eta_r = [0\ 0\ 0]^T$ and initial attitude $\eta_0 = [0.5\ \ -0.1\ \ 1]^T$. Again the attitude states converges to zero from their initial errors; this time a little slower but without the overshoot seen previously.

The angular velocity states also shows a similar, but more smooth, response than in the previous simulation. The (c) graph clearly shows that the inputs no longer are switching as seen previously.

The (d) graph shows the intensity of control calculations; the blue line shows the frequency of controller updates (averaged over periods of 0.2 s) and the red line is a moving average filtered version of the same data with a window spanning 2 s. It can be seen that the controller demands extensive computing resources initially while stabilising the system, and that the required resources is reduced as the system operates near to its set-point.

The time required for the execution is 0.22 s, corresponding roughly 230 times real-time on a contemporary computer. A vast improvement compared to the previous execution.



(a) Attitude Response

(b) Angular velocities

(c) Control Input signal

(d) Control calculation intensity

Figure 6.5: SMC control results with approximated switching function. Chattering is no longer evident.

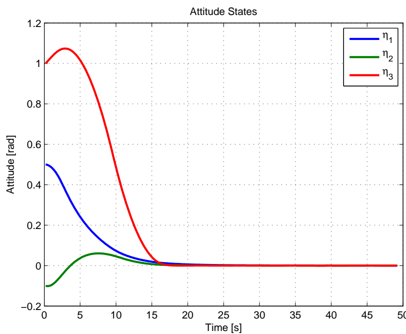
6.4.3 Performance Tuned Results

Continuing from the previous set of parameters we now wish to see if performance of the controller can be improved. We select to utilise the k_i gains (cf. Equation (6.18)) in order for the controller to react strongly to large errors, both for the error to converge quickly, but also in order to reduce the computation time that is required throughout the execution. The controller parameters are:

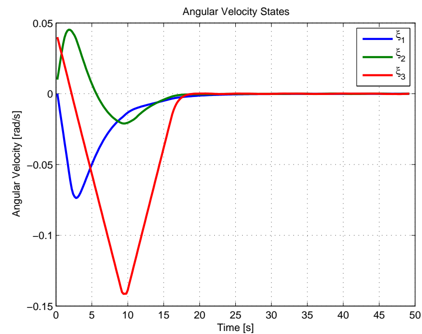
$$k_i = 0.475, \quad q_i = 0, \quad l_i = 1, \quad \epsilon_i = 0.2 \;\; \text{for } 1 \geq i \geq 3 \tag{6.25}$$
$$\text{switch} = \tanh(\cdot), \quad \beta(\eta, \xi) = 0.01$$

Results are given in Figure 6.6 with $\eta_r = [0\;0\;0\,]^T$ and the initial attitude is as

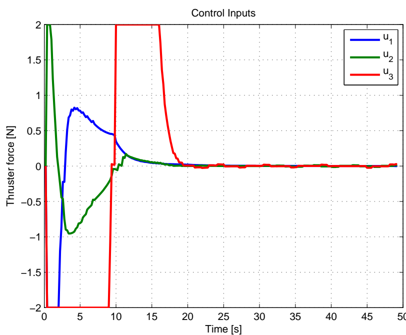
previously $\eta_0 = [0.5 \; -0.1 \; 1]^T$. It can be seen that the attitude is stabilised faster than in the previous cases.



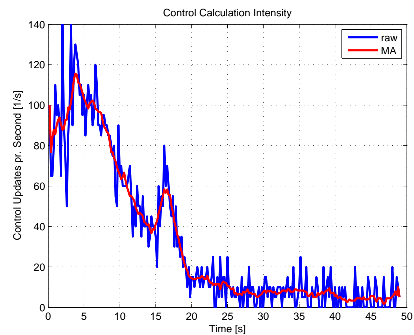
(a) Attitude response



(b) Angular velocities



(c) Control input signal



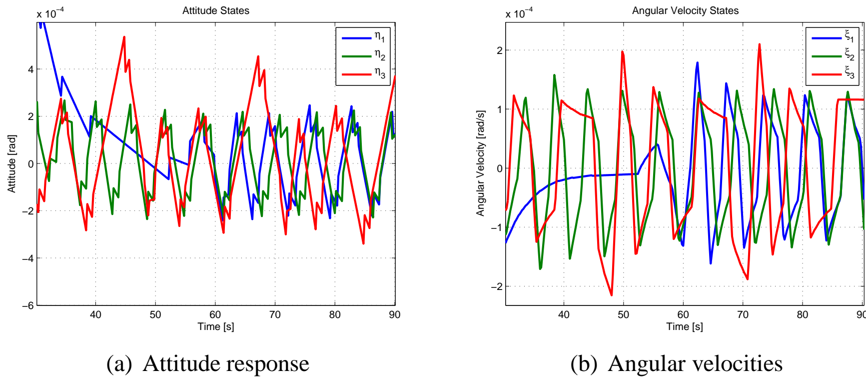
(d) Control calculation intensity

Figure 6.6: SMC control results with performance tuned reaching law.

The plot of the angular velocities (b) shows that the new reaching law does not stabilise at a constant rate, but more aggressively drives the errors to zero. This is also evident from the plot of the control inputs (c) which spend more time in saturation. The plot of calculation intensity (d) also reflects the increased performance as it can be seen that initially there is a period of high intensity which reduces when the states converges to their set points. Execution time is 0.20 s, equivalent to 250 times real-time computation.

6.4.4 Discussion of the Final Error

From all the graphs shown in this chapter so far it seems that both the attitude and angular velocity states converges nicely to zero. Figure 6.7 shows a close up plot of the states after the convergence; these plots corresponds to the controller parameters with results presented in Figure 6.6.



(a) Attitude response (b) Angular velocities

Figure 6.7: SMC control in steady state. Bounded oscillations are evident.

The plots show that the states does not converge to zero exactly but ends in bounded oscillations. This is what can be expected for a QSS2 based system; See Section 3.3 on page 38 which explains that oscillatory region is proportional to the quanta selection. In practice one can select quanta small enough that this region is insignificant. See Subsection 4.4.3 on page 63 for a similar discussion on quanta selection on estimation errors for the QSS2 based Extended Kalman Filter. Another possibility could be to implement a small dead-zone for the actuators in order to avoid excessive actuation in this region.

6.5 Chapter Summary

This chapter introduced sliding mode control for quantised state systems and developed a controller structure, and corresponding software implementation, that can stabilise a large class of non-linear systems, even in the presence of disturbances and model uncertainties.

The proposed object oriented controller structure is highly adaptable and each component can be replaced by customised elements to suit specific application requirements. The approach was verified on a deep space probe attitude control example.

The practical relevance of this will be investigated further in the following chapter, which provides results in presence of uncertainties and estimation errors. However, it is reasonably to expect that if the quanta are selected small enough to result in oscillations that are not significant compared to the uncertainty of the state knowledge then these oscillations will not have any practical significance.

The presented algorithm concerns state stabilisation, however, the proposed structure can easily be adapted to provide tracking; a suitable methodology is developed in [Khalil, 2000].

7 Evaluation of Estimation Based Control

This chapter brings together the results from the previous chapters and provides simulated results for estimation based control of the deep space probe case using the QSS based Extended Kalman Filter and the QSS based controller structures developed in the previous two chapters.

7.1 Introduction and Infrastructure for Evaluation

The previous chapters saw the development of an estimator and two control algorithms which are specifically designed to be implemented using quantised state systems. The two chapters on control algorithms provided evaluation of control performance under ideal conditions, i.e. perfect state knowledge and no disturbances or uncertainties. This chapter investigates performance under more realistic settings including state estimation errors, dynamic disturbances and uncertain model parameters. Consider Figure 7.1 which reflects the structure that will be used in the evaluation approach of this chapter.

There are two major parts in the structure; A so-called "truth model", which is implemented in *Simulink* and which simulates the physical system with all details, and a control part implemented in DEVS which consists of an internal control model, an estimator, and a controller. The two parts exchange control inputs and sensor outputs through a network and the communication is facilitated in both parts by specific objects responsible for communication and time synchronisation.

The infrastructure and the comparative study of control algorithms are the contributions of the chapter. Section 7.2 will first describe the evaluation case, and then the mechanism for interaction between the DEVS based algorithms and the *Simulink* simulation environment. Thereafter, the two following sections will provide results first for the objective directed control algorithm and then for the

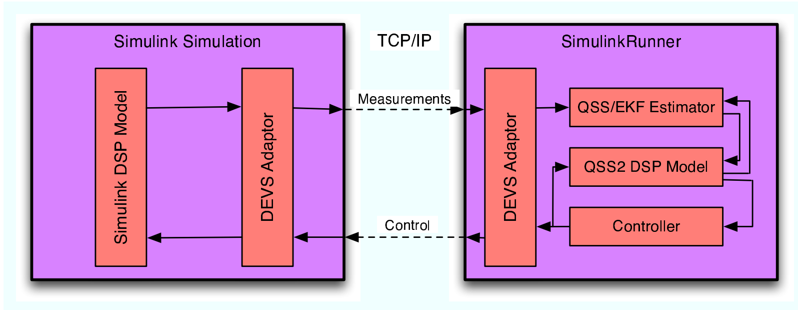


Figure 7.1: Infrastructure for estimation based evaluation. A model representing the "real" plant is simulated using *Simulink* and the estimator and controller are implemented in DEVS. Communication between the two is facilitated by a TCP/IP network link

sliding mode control algorithm.

As an example of the declarative manner in which models, controllers and estimators are constructed and configured the complete code listing for setting up the case as presented in in Section 7.4 can be found in Appendix C on page 191.

7.2 Case Study Details

The case study that will be used for evaluation in this chapter is based on the Deep Space Probe (DSP) model presented in Chapter 4 on page 51. During the DSP mission it is envisioned that it will have to conduct a fly-by of Jupiter in order to gain a gravity boost of its ΔV (Speed wrt. the target planet), consider Figure 7.2.

During the fly-by the DSP is required to autonomously maintain inertial pointing of its antenna dish towards the Earth in order to transmit telemetry throughout the whole manoeuvre. During the fly-by the DSP will experience temporary sensor unavailability, due to eclipse, and attitude disturbances because of influence from the Jovian magnetic and gravitational fields. With this scenario as inspiration the following paragraphs will give more details on the case.

Sensor Models

In Chapter 4 on page 51 the DSP used two vector observations to reconstruct atti-

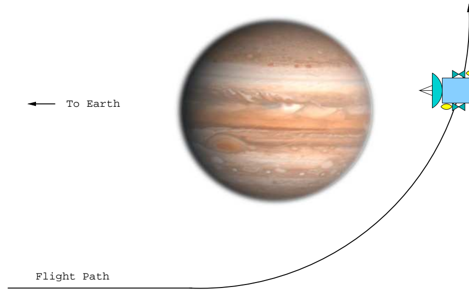


Figure 7.2: Inspiration for the case study; A Jovian fly-by.

tude and angular velocity information. These sensors are again available, but are now sampled at 5 Hz for the sun-sensor and 0.5 Hz for the star-sensor. An Inertial Measurement Unit (IMU) measuring angular velocities will also be used for the evaluation of the sliding mode controller later on. The IMU is sampled at 10 Hz and provides an accuracy represented by a standard deviation of $\sigma_{IMU} = 0.0001\ ^o/s$.

The simulations to be presented in the following will demonstrate the effect of temporarily loosing information from the sun-sensor which is imagined to be eclipsed by Jupiter, during this time it is up to the estimator to provide the best possible state-estimate despite the fact that the remaining sensors do not provide full state observability.

Disturbances

Most notably the fly-by will cause attitude disturbances in terms of gravity gradient torques and magnetic torques. For modelling simplicity only the latter will be considered here, however, from a control point of view the two type of disturbances are qualitatively similar, and results can be generalised to cover both. The magnetic torque on the spacecraft is expressed by [Fortescue et al., 2003]:

$$\tau = \mathbf{m} \times (\mathbf{A}(\theta)\mathbf{B}(t)) \tag{7.1}$$

where $\mathbf{A}(\theta)$ is a direction cosine matrix parametrised by the angular position states which describes the transformation from inertial coordinates to spacecraft body coordinates, $\mathbf{B}$ is the ambient magnetic field in inertial coordinates, given in of units Tesla, and $\mathbf{m}$ is the magnetic moment of the DSP in units of Am^2.

The magnetic moment of the DSP arises due to current loops and materials with magnetic properties. We assume that $\mathbf{m}$ consists of two components $\mathbf{m}_1$

and $\mathbf{m_2}$ which respectively represent a known magnetic moment determined from pre-launch calibrations to $\mathbf{m}_1 = [0.3 \;\; - 0.2 \; 0.7]^T$ and an unknown component $\mathbf{m_2} = [0.4 \; 0 \; 0]^T$.

The truth model in *Simulink* implements $\mathbf{m}_1 + \mathbf{m}_2$ to calculate the magnetic torque, and the sliding mode controller will make use of $\mathbf{m_1}$ as part of its dynamical uncertainty adaption. To support the latter; availability of a magnetometer is assumed which provides measurements of the local magnetic field in the body frame of the DSP at a rate of 1 Hz.

A simplified model of the Jovian magnetic field near the equatorial plane can be expressed as follows under the assumption that the field resembles that of a simple dipole field and that the field strength is about 15 times that of the near Earth environment.

$$\mathbf{B}(t) = \mathbf{B} = [0 \;\; 0 \;\; 0.6]^T \;\; mT$$

While this is a very simplified model it is adequate to demonstrate the ability of the controllers to cope with dynamical disturbances.

Model Uncertainty

The truth and control model have differing parameters in order to introduce model uncertainty; the control model implements the parameters of the dynamic model presented in Equations (4.12 on page 60) and (4.13 on page 61). The truth model implements an inertia matrix with perturbed parameters compared to the control model. The truth model inertia matrix is:

$$\mathbf{J} = \begin{bmatrix} 28.0 & -0.3 & -1.1 \\ -0.3 & 29.0 & -1.6 \\ -1.1 & -1.6 & 54.0 \end{bmatrix}$$

The model for the thruster actuation system was introduced in Section 6.4 on page 101. It is comprised of throttleable thrusters from 0 to 0.8 N with a torque arm of 50 cm. This information is used in the control model, while the truth-model has different values for the torque arm lengths:

$$\mathbf{l} = [0.48 \;\; 0.52 \;\; 0.55]^T \;\; m$$

In summary; the control and truth model differ both in inertia parameters and thrust model parameters and hence will contribute to providing a realistic measure of performance for the controllers under realistic circumstances.

7.2.1 Interconnection with Matlab/Simulink

As indicated in Figure 7.1, exchange of data between the DEVS and Simulink simulation environment is network based. This subsection provides details on the implementation of this scheme. Consider Figure 7.3 which is a class diagram of the involved classes.

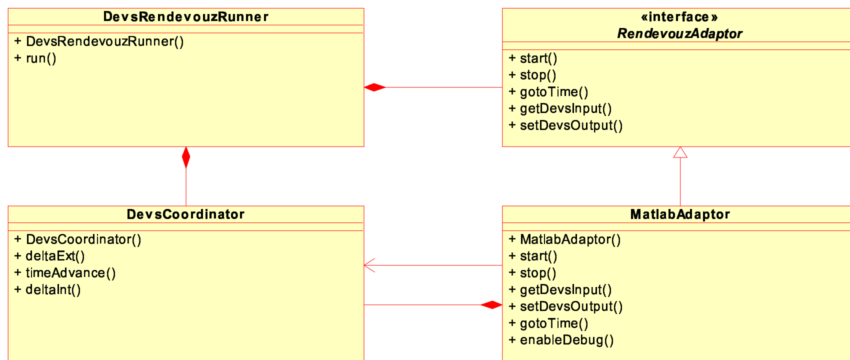


Figure 7.3: Class diagram for DEVS model with IO to/from *Simulink*.

A replacement for the `DevsRunner` class introduced in Subsection 2.2.3 on page 23 called `DevsRendezvousRunner` was developed; It is composed of a `DevsCoordinator` class which holds the system to be simulated/executed and a class that implements the `RendezvousAdaptor` interface. The runner class and the adaptor interface allows two systems to simulate/execute in lock-step by each taking a specified forward step in time and then exchange inputs and outputs, before taking another step to the next rendezvous point. See Figure 7.4 for an overview of the protocol for advancing the simulations.

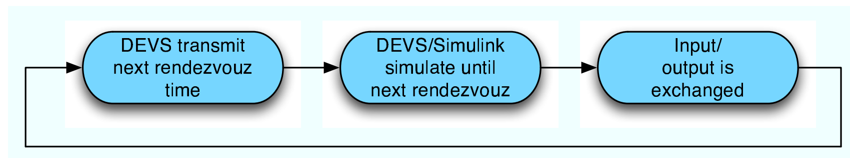


Figure 7.4: The steps in the rendezvous protocol.

The `MatlabAdaptor` class implements this interface and communicates through socket-based TCP/IP networking with a running *Simulink* simulation,

which on its part implements a specific *Matlab* S-function taking care of communication on the *Simulink* side and implements a sample time to match the specified forward step time. This has been the mechanism used in the remainder of the chapter to close the loop between the control system implemented in DEVS and the "truth model" implemented in *Simulink*.

For a concrete application the next step after verification through simulations would be to write a new implementation for the `RendezvousAdaptor` interface which communicates with the sensors and actuators on the spacecraft. This way the control system can transition from a simulation environment to the application environment without changing a single line of code in the control system software.

7.3 Optimising Control Results

This section provides results for the control algorithm developed in Chapter 5 on page 69. The rendezvous interval that is used to synchronise the truth model simulation in Simulink and the control/estimation software in DEVS has been set to 0.05 s. The QSS2 implementation of the DSP model utilises quanta of $\Delta Q = 10^{-5}$ rad for the attitude states and $\Delta Q = 10^{-6}$ rad/s for angular velocity states. These values corresponds to the values chosen in Chapter 4 on page 51 in order to be insignificant compared to the estimation error.

7.3.1 Simulation Results - Optimising Control

This section shows results of the optimising controller applied on the case described above. A performance function to provide inertial attitude stabilisation was found as:

$$v_c(\theta,\omega) = 30\omega_1^2 + 30\omega_2^2 + 50\omega_2^2 + 5(30(\theta_1 - r_1)^2 + 30(\theta_2 - r_2)^2 + 50(\theta_3 - r_3)^2)$$

where $\mathbf{r} = [r_1\ r_2\ r_3]$ are the attitude references for each corresponding Euler angle. The leading factor before each term is the inertia of the corresponding axis in order to provide a weighting that corresponds to the difficulty of turning around the corresponding axis. The leading factor, 5, in front of all the attitude related terms is an empirically found relative weighting between keeping small angular rates and reaching the target attitude quickly.

Other parameters for the controller are the saturation limits of ± 0.8 N for each thruster, and the maximum allowed rate of change of the control signals which were found to be:

$$\dot{\mathbf{u}}_{max} = -\dot{\mathbf{u}}_{min} = [0.3\ 0.3\ 0.3]$$

In this case the values were found by trial and error rather than considering physical limitations of the thrusters, i.e. actuator dynamics is considered fast enough to track this signal. Finally, a control cost vector was found by a few simulation iterations to provide good performance:

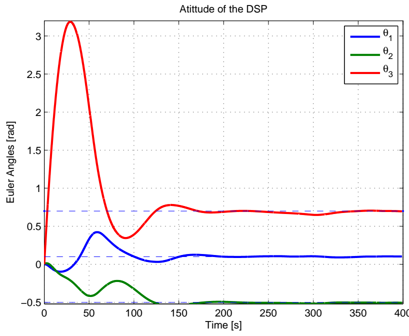
$$\mathbf{P} = \begin{bmatrix} 0.15 & 0 & 0 \\ 0 & 0.15 & 0 \\ 0 & 0 & 0.15 \end{bmatrix} \tag{7.2}$$

The simulations results for the case with these controller parameters can be seen in Figure 7.5 for an attitude reference of: $r = [0.1 \quad -0.5 \quad 0.7]^T$ and initialisation conditions of zero attitude and a significant roll around the axis of most inertia. Further, from time 200 s to 300 s no sun-sensor measurements are available due the sensor being eclipsed by Jupiter.

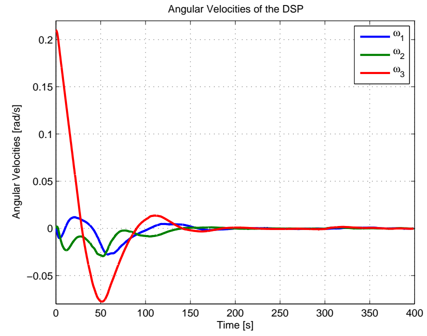
The (a) and (b) graphs on the figure show the attitude and angular velocities reported by the truth model throughout the simulation; it can be seen that the controller effectively reduces the angular velocities and positions the probe at the correct attitude within 200 s. During the following period where the sun-sensor is unavailable small drifts in the attitude, graph (a), can be seen, specially for the θ_3 state and at 300 s, where the sensor becomes available again, the attitude states are again stabilised at their references.

The (c) and (d) graphs show the estimation error by the QSS/EKF filter for attitude and angular velocity states, respectively. The initial error is quickly reduced on the first sample on both graphs and continues to reduce gradually until 200 s where the sun-sensor becomes unavailable. Hereafter it is clear that the attitude estimates, graph (c), begins to drift over the next 100 s due to the limited observability with the available star-sensor. At time 300 s the sun-sensor again becomes available and the estimate quickly converges again.

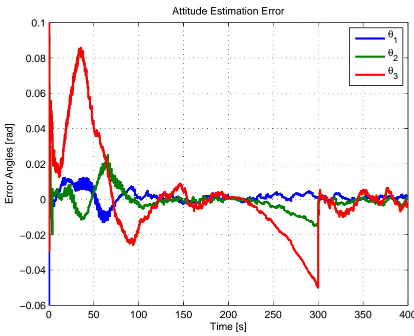
The (e) graph shows actuation signals to the thrusters. Initially there is a large response in order to cancel the roll-rate whereafter all actuation signals converge close to zero. After the sun-sensor becomes available after the blinded period ending at time 300 s, it can be seen that the thrusters increase activity in order to re-align the craft to the proper attitude.



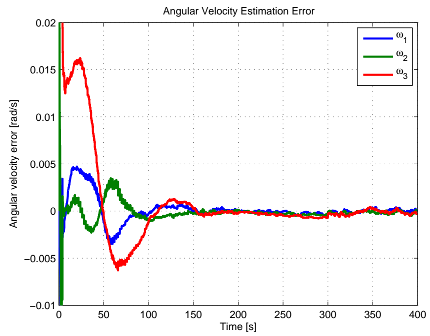
(a) Truth model attitude



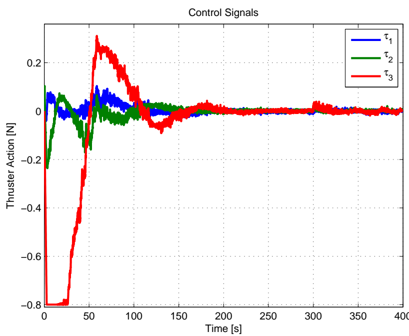
(b) Truth model angular velocities



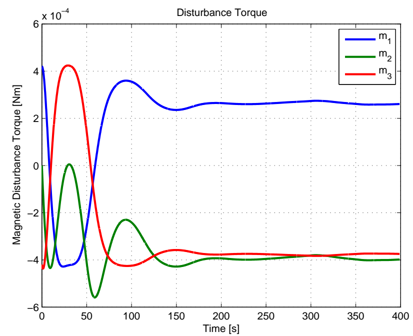
(c) Attitude estimation error



(d) Angular velocities estimation error



(e) Control input signals



(f) Magnetic disturbance

Figure 7.5: Jupiter gravity assist results with the optimising controller.

The (f) graph shows the disturbance torque due to the magnetic disturbance. It can be seen how the directionality of the disturbance is linked to changes in attitude of the craft.

In summary the estimation and control algorithm combined perform well. However, a number of points are worth further discussion; from the graph showing actuation signals it can be seen that the signals have a high-frequency low amplitude switching component. This behaviour was also observed on the simulations carried out in Chapter 5 on page 69. This behaviour is due to the nature of the control law which switches between positive and negative values of the input slope, see Equation (5.8 on page 73), this behaviour is not particularly desirable for this case since it means that propellant is consumed.

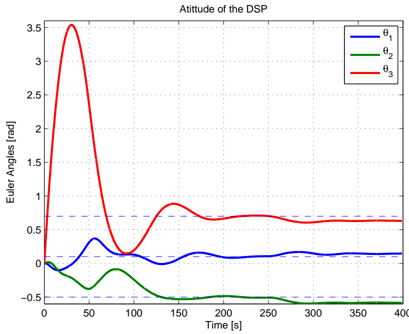
Second point for discussion is the final error; from the graphs it is clear that there is some motion close to the reference values. Zooming on the plots it can be seen that this corresponds with amplitudes of the estimation error. Further, since the magnetic disturbance has an amplitude that is relative small compared to the estimation errors it is difficult to validate the robustness properties towards this disturbance.

7.3.2 Results with Exaggerated Disturbances

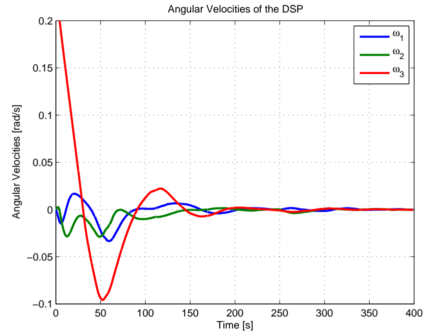
In order to address the two points discussed above a new simulation case will be presented where a low-pass filter is applied to the control signals before being applied to the truth-model. The idea is to see if this is a practical remedy to reduce propellant consumption while not affecting performance. The filter will have a cut-off frequency of 0.3 Hz, which is chosen slightly higher than the maximum frequency at which the controller can alternate between control signal saturation limits.

Secondly, we will let the sun-sensor be available throughout the whole simulation, but between time 200 s and 300 s the magnetic disturbance will be multiplied with a factor of 25 to investigate the controller response to this disturbance. The results are presented in Figure 7.6.

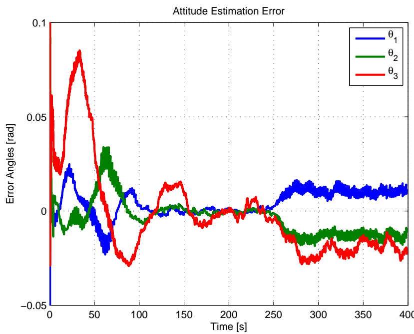
Observing first the (e) graph it can be seen that the control inputs are now much more smooth than in the corresponding graph on Figure 7.5. Apart from removing the high frequency oscillations the control input curves are equivalent and it can also be observed that performance of the controller is equivalent to the previous case. In summary; the addition of the low-pass filter clearly is a benefit



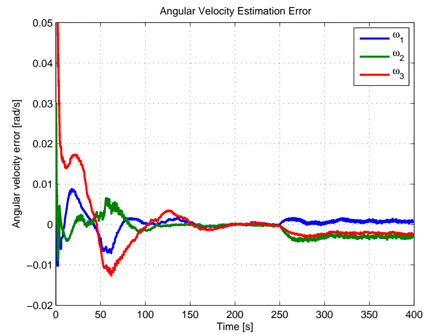
(a) Truth model attitude



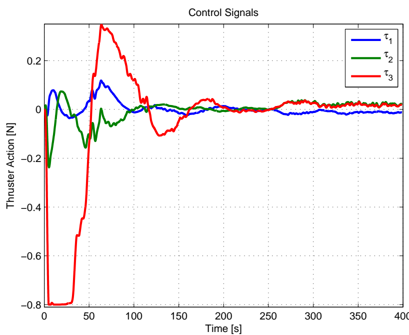
(b) Truth model angular velocities



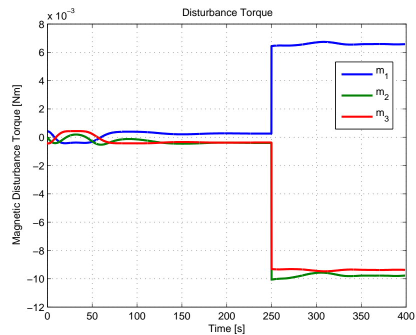
(c) Attitude estimation error



(d) Angular velocities estimation error



(e) Control input signals



(f) Magnetic disturbance

Figure 7.6: Results with filtered inputs and exaggerated disturbance. Note the smoother control input signals.

in terms of implementation and has no significant effect on overall performance.

If the results are observed after time 200 s it is clear from the (f) graph that there is a sudden increase in the disturbance input and errors start to develop for both velocity and attitude states. It can also be seen that the controller reacts to the disturbance and manages to reduce angular velocities, graph (b), to zero again and stabilise the attitude states (graph (a)), although with a final static error. The (c) and (d) also clearly show that the estimator cannot provide unbiased estimates in the presence of this very low-frequent disturbance.

The amplitude of the final error is due to Pareto optimality in the performance function between the objective of driving the state performance function to zero and at the same time drive control inputs to zero. Reducing control cost will reduce the final error, but also provide a less dampened response to the initial error.

7.4 Sliding Mode Control Results

This section describes evaluation results obtained for the sliding mode controller structure. Results are presented for the case above with exaggerated disturbances in order to evaluate robustness properties of the control scheme.

We define the control objective to be attitude stabilisation to specified set-points $\eta_r = [\eta_{r,1}\ \eta_{r,2}\ \eta_{r,3}]^T$ and represent this control objective in the following objective function for the guidance controller (cf. Subsection 6.3.2 on page 98):

$$v = (\eta_1 - \eta_{r,1})^2 + (\eta_2 - \eta_{r,2})^2 + (\eta_3 - \eta_{r,3})^2$$

A dynamic disturbance bound calculation, as proposed in Subsection 6.3.5 on page 100, was implemented to provide bounds on the magnetic disturbance input described by Equation (7.1). The bound is calculated using the known magnetic moment of the craft, $\mathbf{m}_1$, and the ambient magnetic field as measured by the on-board magnetometer, $\mathbf{B_m}$, i.e. the bound is given by:

$$|\tau| = 1.5|\mathbf{m_1} \times \mathbf{B_m}| \tag{7.3}$$

where the leading factor, 1.5, ensures additional robustness towards unknown magnetic moments (e.g. $\mathbf{m}_2$ which is part of the truth model, but not the control model). The settings of remaining controller parameters can be seen by the API calls below:

```
smc.setSaturation(new double[]{0.8, 0.8, 0.8});
smc.setProportionalGain(new double[]{0.06,0.06,0.06});
smc.setBoundaryLayer(new TanhLayer(0.1));
```

Otherwise initial conditions in the truth model and QSS/EKF estimator is as in the previous section.

7.4.1 Simulation Results - Sliding Mode Control

Figure 7.7 provides results for the sliding mode controller on the case with exaggerated disturbance inputs.

From graphs (a) and (b) it can be seen that performance is excellent in terms of reaching the attitude reference and maintaining it even in the presence of the disturbance signal. Graphs (c) and (d) also show that the estimator performs well.

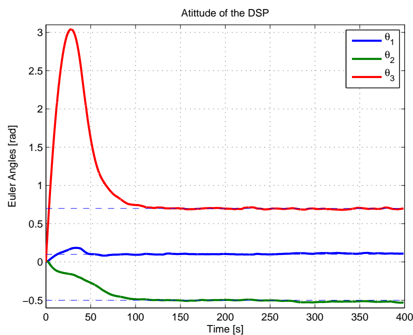
However, looking at the (e) graph it is clear that the control signal is not desirable for a real system due to the extreme switching that is evident. This phenomenon is not due to chattering, but due to the coupling between the estimation error and the equivalent control, cf. Equation (6.5 on page 91). This will be investigated further in the following.

7.4.2 Effect of Estimation Error on Equivalent Control

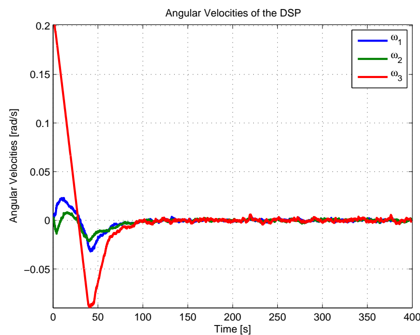
The results presented in Figure 7.8 are for a simulation where only the equivalent control is applied to the system, i.e. the control will attempt to bring all velocity states to zero.

It can be seen that the control is effective in doing so, but even in steady-state there is significant control switching and perturbations to velocity states as a consequence. This control action is driven by the estimation errors in the angular velocity states.

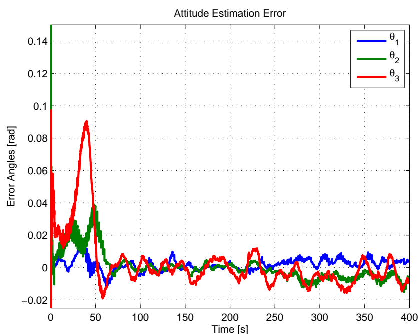
Hence, to improve the situation, accuracy of the state estimation must be increased, which will lead to reduced noise-driven actuation. This will be achieved by adding the inertial measurement unit sensor described in subsection 7.2. Filtering the control signal (as done previously for the optimising controller) is not an option since it will introduce chattering, due to the unmodeled delay in applying the control signal [Khalil, 2000].



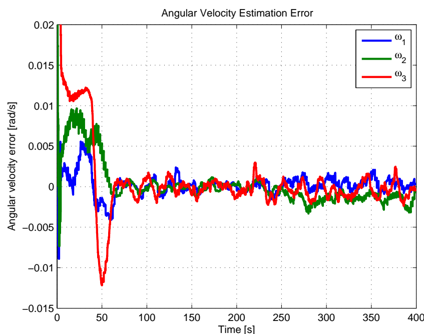
(a) Truth model attitude



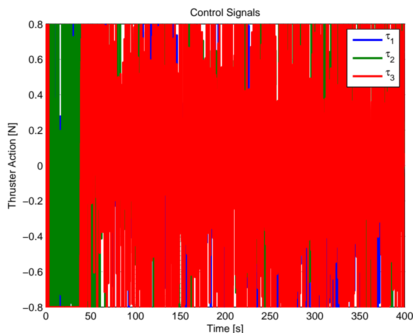
(b) Truth model angular velocities



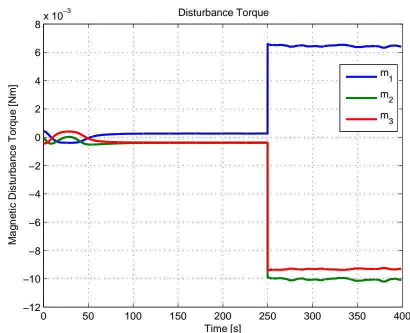
(c) Attitude estimation error



(d) Angular velocities estimation error

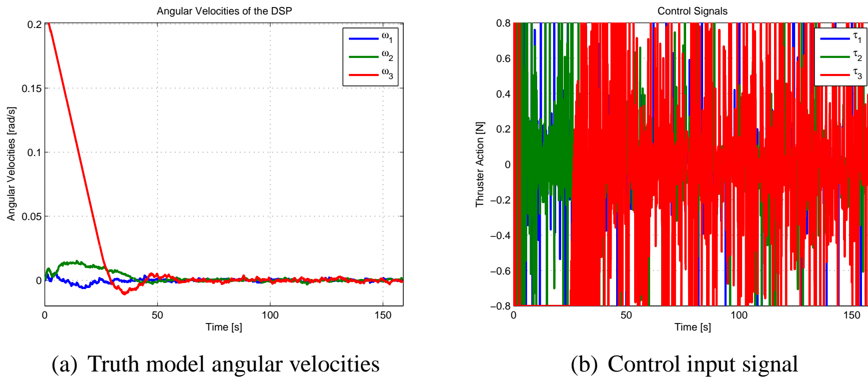


(e) Control input signals



(f) Magnetic disturbance

Figure 7.7: Sliding mode results. The control signals are clearly undesirable.



(a) Truth model angular velocities

(b) Control input signal

Figure 7.8: Results with only Equivalent Control. Control switching caused by estimation error is evident

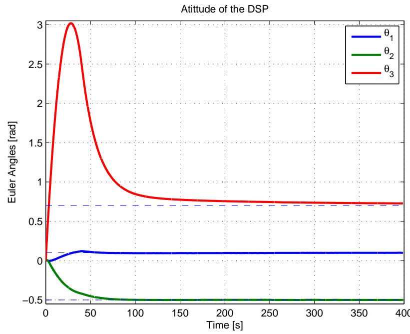
7.4.3 Sliding Mode Control with IMU

Figure 7.9 provides results for the sliding mode controller on the case with exaggerated disturbance, with an added IMU sensor for improved estimation accuracy.

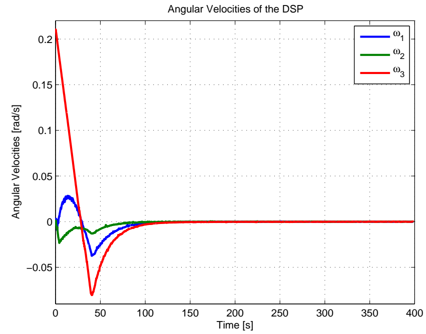
It can be seen from the top graphs that performance in terms of reaching time is equivalent to the non-IMU case. However, in steady state almost no perturbations are visible on the states and after time 250 s no reaction to the disturbance is evident at all. It is noted, however, that the θ_3-state is slow to converge fully to its reference.

Observing the middle graphs it is seen that for the θ_3-state estimator convergence is slow, which was reflected in the attitude response seen on the graph above. The reason for this is believed to be that with the very accurate IMU sensor added to the configuration the Extended Kalman Filter has become very confident, i.e. it has very low co-variance traces in steady state operation. It is possible that the situation can be improved by injecting additional process noise. Angular velocity estimation errors can be seen to correlate with the control input signal, which is natural since there is some difference between the truth and control model used due to uncertainties. Also, it can be seen that the estimation error increases when the exaggerated disturbance is in effect.

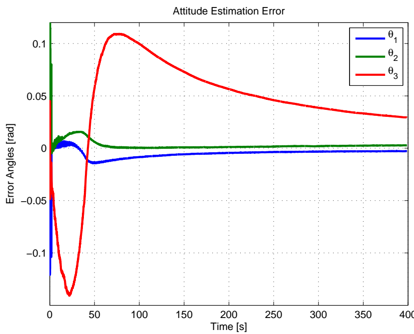
Looking at the bottom-left graph it is clear that the control signal is now better suited for use on a real system with less noise driven actuation. However, some



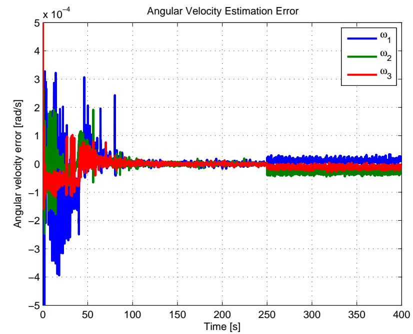
(a) Truth model attitude



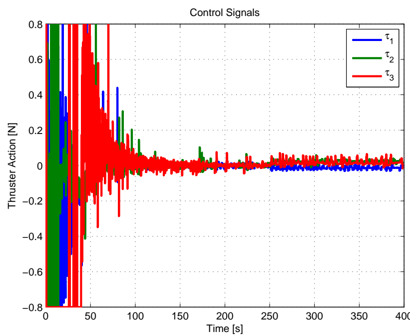
(b) Truth model angular velocities



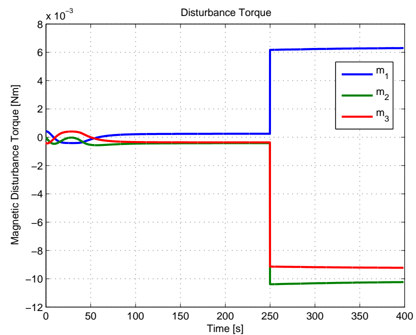
(c) Attitude estimation error



(d) Angular velocities estimation error



(e) Control input signals



(f) Magnetic disturbance

Figure 7.9: Results with IMU. Control switching is significantly reduced compared to previous results.

switching is still observed, especially during the first 100 s of the simulation.

In part this switching can be explained by chattering which is due to the unmodelled sample-and-hold delay of the control signal applied to the truth model. This can be reduced by increasing the rendezvous frequency. A significant part of the switching, however, is ascribed to attitude estimation error which changes rapidly during the first 100 s and causes the guidance controller to update the angular velocity reference often. This part can be reduced by increasing attitude estimation accuracy, e.g. by adding a high performance star-tracker.

In summary; it is concluded that the sliding mode controller offers very good performance in terms of reaching time and disturbance rejection, but it is very sensitive, in terms of desirability of the control input signal, to the estimation error.

7.5 Chapter Summary

This chapter described an evaluation case for evaluating the algorithms developed in the previous chapters under conditions that resemble those experienced by a real-life control problem, i.e. effects of disturbances, uncertainties, estimation errors, and synchronisation.

Using the case the controller structures developed in the past two chapters was evaluated when driven by estimates obtained from the QSS/EKF estimator developed previously. A truth model was implemented in Simulink and input/output signals between the Simulink model and the DEVS based control/estimation algorithms were facilitated by a special adaptor class to provide rendezvous based data exchange.

The optimising control scheme showed to be able to guide the attitude to the desired reference, but it was not very robust against disturbances. The sliding mode controller showed excellent performance in terms of reaching time and disturbance rejection, but is sensitive to estimation errors.

Part III

Hybrid Systems, Simulation and Control

Hybrid Systems and QSS Based Simulation

8

This chapter introduces a control oriented specification of hybrid system models and contributes with a quantised state systems approach for simulating and executing such models. The approach features conservative event detection and can be initialised declaratively from hybrid system specifications expressed in a declarative language, in this case based on the eXtended Markup Language (XML).

8.1 Introduction and Motivation

Hybrid system models capture both continuous and discrete behaviour of a given system and hence possess more power of expression compared to pure continuous models or pure discrete models. However, due to this expressiveness and the associated difficulty of obtaining strong theoretical results there is not today a unified definition of exactly what a hybrid system is, or a unified notation and terminology for such systems.

Instead there are many different hybrid system model definitions tailored to specific problem domains or solutions strategies, e.g. control of piece-wise-linear systems [Bemporad and Morari, 1999], control of piece-wise-affine systems [Habets et al., 2006], or model verification [Henzinger, 1996].

This chapter introduces a definition of hybrid system models which is suitable for supervisory control applications, as will be demonstrated in the next chapter. The definition, given in Section 8.2, is close to the definition given in [Branicky et al., 1998], but differs in some aspects. Associated with the specification is a dedicated language formulated in eXtended Markup Language (XML), which allows users to declare hybrid system models in a format comprehensible to humans, which can also be processed by computer algorithms.

Section 8.3 contributes with a method to execute hybrid system models, as specified in XML, as part of a control system. The key point here is to provide conservative detection of discrete location transitions, and secondly to provide an algorithm that can co-exist with an estimation algorithm which also manipulates the state. Section 8.4 gives some details on the implementation of the approach and the translation from XML files to software entities.

Finally, Section 8.5 provides a full declaration for a hybrid system model known as *Raibert's Hopper* [Back et al., 1993] and provides simulation results for the state evolution of the system based on the developed algorithm.

The work on a control oriented hybrid systems specification and associated XML description is published in [Laursen et al., 2005, Alminde et al., 2006a, Alminde et al., 2006b] as part of a multi-disciplinary framework for working with hybrid system models called Simulation, Observation and Planning in Hybrid Systems (SOPHY). Appendix A on page 173 provide the hybrid system model definition used in SOPHY from which the following definition is extracted. The Appendix also covers composition of hybrid systems which is not covered in the main dissertation.

8.2 Hybrid System Models

This section introduces the formal definition of a hybrid system that will be used in the remainder of this dissertation. In our terminology we will associate the term *location* to cover each of the different continuous systems embedded in the hybrid system model, and shifts between locations are called *transitions*, which are taken when a *transition condition* is true. During transitions between locations the state can be discretely altered by *reset conditions*, and finally transitions can be triggered by *input events* and each transition can emit *output events* when taken.

Definition 8.1 (Hybrid System Model)
A hybrid system model is an 8-tuple:

$$\mathcal{H} = (Q, X, U, Y, E, \mathcal{F}, \mathcal{G}, \mathcal{T})$$

With spaces defined as:
$Q = \{q \mid 1 \leq q \leq s\}$*: is the set of location indexes with cardinal number* $s \in \mathbb{Z}^+$
$X \subseteq \mathbb{R}^n$*: is the continuous state-space with dimension* $n \in \mathbb{Z}^+$
$U \subseteq \mathbb{R}^m$*: is the continuous input-space with dimension* $m \in \mathbb{Z}^+$

$Y \subseteq \mathbb{R}^o$: *is the continuous output-space with dimension* $o \in \mathbb{Z}^+$
$E = \left\{ e | e \in 2^\Sigma \right\}$: *is the set of possible input/output event labels, where* Σ *is a set of labels*

and related maps:
$\mathcal{F} : Q \times X \times U \rightarrow \dot{X}$: *is a mapping onto the tangent bundle* $\dot{X}$ *of* X
$\mathcal{G} : Q \times X \times U \rightarrow Y$: *is a continuous output map*
$\mathcal{T} : Q \times X \times U \times E \rightarrow Q \times X \times E$: *is a transition map*

Remark 8.1.1
Time is not explicitly given in the definition of the system; however, with no loss of generality the model can include an extra state in the continuous map to represent explicit time

Remark 8.1.2
The map $\mathcal{F}$, *as defined above, allows Ordinary Differential Equations (ODE), but not differential algebraic or partial differential equations*

At any point in time the future evolution of a given hybrid system model can be described by its current location, state values, and input values. This we define as the hybrid state of the system and the history of the hybrid state evolution is the hybrid trajectory.

Definition 8.2 (Hybrid State)
At any time the total state of a hybrid system model is described by the triple:

$$\mathcal{S} = (q \in Q, \mathbf{x} \in X, \mathbf{u} \in U)$$

Definition 8.3 (Hybrid Trajectory)
Associated with an interval of time: $t = [t_0; t_f]$ *is the hybrid trajectory:*

$$\mathcal{T}(t) = (q(t), \mathbf{x}(t))$$

where $q(t)$ *is a piece-wise constant function and* $x(t)$ *is a piece-wise continuous function.*

The maps, as described in the definition of hybrid system models, are described very generally. The next definition provides more structure by associating a specific map to each discrete location. This provides a more operational terminology for specifying hybrid system models and implementing corresponding representations in software.

Definition 8.4 (Location Indexed Maps)
The forcing map and output map are refined with more structure, as follows, in order to allow sub-maps to be specified for each discrete location:

$$\begin{aligned}\mathcal{F} &= \left\{\{f_q\}_{q\in Q} : q \times X \times U \rightarrow \dot{X}\right\}\\ \mathcal{G} &= \left\{\{g_q\}_{q\in Q} : q \times X \times U \rightarrow Y\right\}\end{aligned}$$

A similar exercise is undertaken for the transition map by representing $\mathcal{T}$ by transitions mapping between specific locations, and with well-defined transition domain and reset condition.

Definition 8.5 (Transition Map)
The transition map $\mathcal{T}$ is composed of transition relations between specified locations:

$$\mathcal{T} = \left\{\{t_r\}_{r\in\{1,\dots,p\}} | Q \times X \times U \times E \rightarrow Q \times X \times E\right\}$$

where each transition relation is a 6-tuple:

$$\tau_r = \left(q_1, q_2, j_r(\cdot), \mathbf{r}_r(\cdot), e_{r,in} \in 2^{\Sigma}, e_{r,out} \in 2^{\Sigma}\right)$$

where:
$q_1 \in Q$*: is the source location*
$q_2 \in Q$*: is the destination location*
$j_r(\cdot) : \mathcal{X} \times \mathcal{U} \rightarrow \{true, false\}$*:* transition equation *which triggers the transition*
$\mathbf{r}_r(\cdot) : \mathcal{X} \times \mathcal{U} \rightarrow Q \times X$*: is an algebraic reset equation of the state*
$e_{r,in}$*: is an input event that causes the transition to trigger*
$e_{r,out}$*: is an output event that is emitted when the transition is taken*

Remark 8.5.1
When a transition relation, $j_r(\cdot)$, turns true or an input event label is received which is contained in $e_{r,in}$ the corresponding reset condition and location jump is always taken.

8.2.1 Graphical Representation of Hybrid System Models

With the definitions from above it is possible to give a graphical representation of a hybrid system model. Consider Figure 8.1 on the facing page which represents a hybrid system model with two discrete locations and three transition relations.

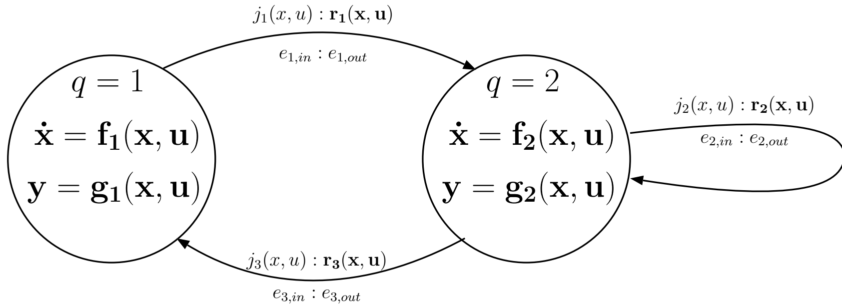


Figure 8.1: A graphical representation of a hybrid system model.

Each location is identified by the value of q and each has distinct state and output maps. Transition relations are drawn as directed edges from source location to destination location. Above each transition relation the transition equation is written followed by the state-reset equation. Under the transition relation the set of input labels that can trigger the transition is written followed by the set of labels that are emitted when the transition is taken. Later in this chapter a concrete example of a hybrid model definition and associated graphical representation will be presented.

8.2.2 XML Specification of Hybrid Systems

In order for hybrid system models to be declared as input to computer algorithms as part of a declarative approach a format must be available which is both human readable and at the same time interpretable for a computer. Therefore a specification using eXtended Markup Language has been developed that maps the hybrid system model definition as described above. XML is a declarative language that is well suited for the purpose.

The XML format for hybrid systems definitions is described in an XML Document Type Definition (DTD) which describe the allowed tags, their allowed hierarchy and multiplicity. Table 8.1 gives and overview of the tags defined in the DTD and their meaning. The complete DTD specification is given in Section A.5 on page 180.

The table illustrates how the specification is divided in three sections; one section giving general information such as model name, parameters and constants,

Element	Data contained in element
SophySystem	
name	The name of the subsystem
documentation?	Text describing the subsystem model
hints?	Parameters to be passed on to on-line system
hint+(name)	Name (in attribute) and value (string) pair for hint
constants?	Collection of constants to be substituted in equations
constant+(name)	Name (in attribute) and value pair for the constant
states	
state*(documentation)	Name of state, optional documentation in attribute
inputs	
input*(documentation)	Name of input, optional documentation in attribute
outputs	
output*(documentation)	Name of output, optional documentation in attribute
locations	
location+	
name	The name of the location
documentation?	Documenting text for this location
diffequation*(state)	A mathematical expression giving the differential equation for each state
outputmap*(output)	Mathematical expression involving state variables that evaluates to an output value for this output
transitions?	
transition+	
name	The name of the transition
documentation?	Documenting text for this transition
domain	Logical expression involving state and input variables
reset	Reset associated with this transition
destination	Name of destination location
statereset*(state)	Expression to reset the state indicated as attribute
inputevent*	Names of events that are emitted when transitioning
outputevent*	Names of events that can trigger this transition

Table 8.1: Tags and their meaning in XML descriptions of hybrid system models. '?' means 0 or 1 corresponding tag required, '*' is zero to many and '+' means at least one. Tags in parenthesis, following a tag definition, are tags that must be included within the tag being defined [Consortium, 2006].

a second section defining the state-, input-, and output spaces, and a final section providing details for each location in the model. It can also be seen that optional documentation tags and attributes are included to provide clear binding between model elements and documentation. This can e.g. be used in a graphical model viewer, where the documentation can pop-up when the mouse points to a specific location or transition.

A full example of a model defined in this format can be seen in Section A.6 on page 182. The following shows an example specification of a single location for a system of a free-falling object and a transition associated with the event of the object hitting the ground.

```
<location>
  <name>FreeFall</name>
  <diffequation state="Position"> Speed</diffequation>
  <diffequation state="Speed"> -9.82</diffequation>
  <outputmap output="Position"> Position </outputmap>
  <outputmap output="Speed"> Speed </outputmap>

  <transitions>
    <transition>
      <name> GroundImpact</name>
      <domain>Position &lt; 5</domain>
      <reset>
        <destination> Crashed </destination>
        <statereset state="Speed"> 0 </statereset>
      </reset>
    </transition>
  </transitions>
</location>
```

8.3 Hybrid System Execution in DEVS/QSS

This section proceeds to describe a method based on DEVS and QSS that simulates or executes a hybrid system models as defined above. A QSS approach to simulation of hybrid system models was presented in [Kofman, 2004], but this work extends the approach by adding state resets and modifying it to allow use in execution environments (rather than just simulation), i.e. it can be used as part of an on-line control system where the state variables can also be manipulated by an estimator. Further, software is developed that automatically translates a hybrid system model specified in the XML into executable code.

The challenge in simulating or executing a hybrid system model is state-event detection, i.e. accurately detecting when a transition equation $j_r(\cdot)$ turns true, at which time, exactly, the related transition must be executed for a consistent result. Many methods for hybrid simulation based on time discrete methods check the transition equations each time step and if a condition has become true they use bisection or interpolation to *roll-back* the simulation to the event time [Barton and Lee, 2002, Taylor and Kebede, 1997, Lieu et al., 1999]. Clearly, this is not suitable

for a method that supports use as a execution system; in this case causality must be preserved.

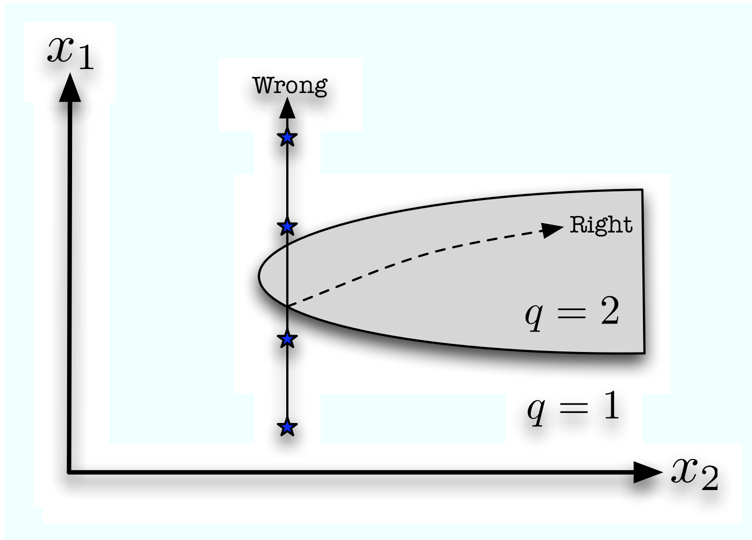


Figure 8.2: The problem of detecting events consistently in sampled systems. Blue points represent discrete sample points.

Another problem in discrete time approaches to propagating the state of hybrid system models is missed state-event detections; if the time-steps taken by the algorithm are too large it may fail to activate the roll-back procedure, consider Figure 8.2. Here the dots represent the points in state-space corresponding to the time instances where the propagation algorithm performs computation and the shaded area is a transition domain; depending on the exact times of evaluation the system may evolve along different trajectories.

The simulation architecture to be presented in the following paragraphs makes use of the DEVS/QSS approach to provide consistent and conservative event detection by extrapolating the current state and identifying future state-events times.

8.3.1 Hybrid Execution/Simulation Architecture

Figure 8.3 depicts the software architecture that will be used to implement hybrid system models in DEVS/QSS in terms of a block diagram.

The two maps, $\mathbf{f}$ and $\mathbf{g}$, are based on the QSS2 function map described in

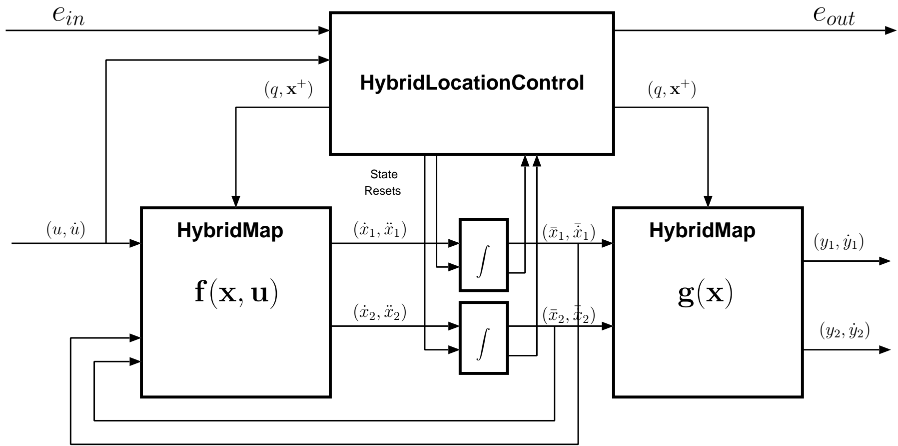


Figure 8.3: Block diagram of architecture for simulating and executing hybrid system models. Example with two states, one input, and two outputs.

Subsection 3.2.2 on page 33, but are extended to contain multiple embedded maps and has an extra input port which receives information on the current location, q, and the state value following a transition $\mathbf{x}^+$. When a message is received on this port the `Qss2HybridMap` switches to the associated set of equations for the new location and recalculates output values and their derivatives.

The integrators each has two inputs; the first to receive derivate values from the function map and the second to receive state reset events either from an estimator or from the `Qss2HybridLocationControl` class which handles hybrid transitions.

The first output of the integrator is as previously, i.e. a first order state model that is updated when the quantum criteria, Equation (3.6 on page 33) is fulfilled. The new second output outputs the full second order state model, see Equation (3.5 on page 33) whenever the integrator block receives new input. This new output is used in the `Qss2HybridLocationControl` class to conservatively detect state events and execute corresponding transitions. This is explained in more detail in the following.

8.3.2 Hybrid Location Control

The `Qss2HybridLocationControl` (HLC) is the most important class in terms of simulating/executing hybrid system models and will be described more in depth in the following. The responsibilities of the class is to:

- conservatively predict location changes times
- execute location changes and associated resets
- react to incoming events and communicate output events on location changes

Of these the first responsibility is the most daunting and will be described in the following. The core point is to predict when to schedule location changes, i.e. to predict when the state values in the simulation will enter a region of the state-space where the transition equation $j_r(\cdot)$ of Definition (8.5) turns true. We consider $j_r(\cdot)$ as a logical relation of inequalities, e.g.:

$$j_3(\mathbf{x}, \mathbf{u}) = (x_1 < 5) \wedge (3x_2 + 2x_3 > 4) \tag{8.1}$$

In order to predict location changes the shift of logic states of all inequalities, which are part of all transitions in a given location, must be maintained and predicted into the future. To this end the HLC class uses two inner classes to represent transitions and inequalities respectively, see Figure 8.4.

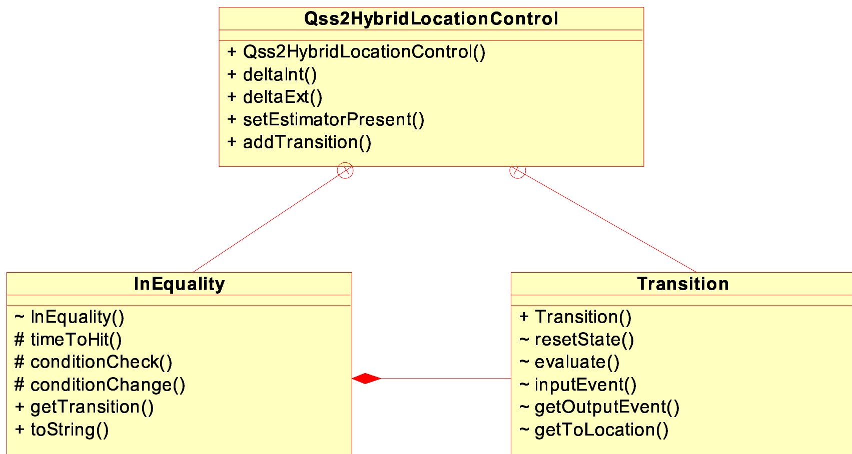


Figure 8.4: The HLC class and its two inner classes representing inequalities and transitions.

An `InEquality` object is created for each inequality in the hybrid system model, like e.g. $x_1 < 5$ from Equation (8.1), and assigned a unique id. The `timeToHit()` method (to be described soon) is then called on all `InEquality` objects which are relevant for the current location and the HLC schedules an internal DEVS event at the least positive time.

At the execution of internal event the logic state of the triggering `InEquality` object is updated and it is checked if the full composite logical expression, e.g. Equation (8.1) is fulfilled, using the `evaluate()` method. If the transition is enabled the HLC class utilises the methods in the `Transition` class to perform a state reset and emit the correct output labels. If the logical expression is not fulfilled; the HLC class does nothing but allows continued continuous state evolution until the next inequality changes logic state.

Conservative Event Detection

As described above the HLC classes requires each inequality to report when in the future it will change logic state. This is possible since with the `Qss2Map` and `Qss2Integrator` the trajectory is known into the future as it is given by the parabolic model maintained in the integrator. The HLC receives this trajectory from each integrator, as described above.

However, in the following we will require that the inequalities are scalar equations which are linear functions of the state and input, respectively. Thus we require that each inequality can be put on the following form, where, for simplicity, $\mathbf{z} = [\mathbf{x}^T \ \mathbf{u}^T]^T$ is the concatenation of the state and input vector respectively:

$$\begin{aligned} l_{id} &: \ \mathbf{a^T z} - b > 0, \ \text{ or} \\ l_{id} &: \ \mathbf{a^T z} - b < 0 \end{aligned} \tag{8.2}$$

Where l_{id} is an assigned unique identification to track the inequality. For a numerical implementation on a computer we make no distinction between the operator pair $>$ and $\geq$, as well as the pair$<$ and $\leq$.

The `timeToHit()` method must now determine when the inequality turns true; given that $\mathbf{z}$, $\dot{\mathbf{z}}, \ddot{\mathbf{z}}$ are known from the integrators, we know the state will evolve along the following parabolic path until the next event where new information will be received:

$$\mathbf{z}(\tau) = \frac{1}{2}\ddot{\mathbf{z}}(t_0)\tau^2 + \dot{\mathbf{z}}(t_0)\tau + \mathbf{z}(t_0)$$

where t_0 indicates the time where the last information was received from the integrators, by inserting this equation into the lefthand side side of Equation (8.2) and setting the expression equal to zero, we obtain the characteristic equation:

$$\frac{1}{2}\mathbf{a}^{\mathbf{T}}\ddot{\mathbf{z}}(t_0)\tau^2 + \mathbf{a}^{\mathbf{T}}\dot{\mathbf{z}}(t_0)\tau + \mathbf{a}^{\mathbf{T}}\mathbf{z}(t_0) - b = 0 \tag{8.3}$$

which is a standard second order equation in the variable of τ that can be solved deterministically. Positive roots of this expression indicate times in the future where the inequality will change logic state under the current trajectory of $\mathbf{z}$. If there are two positive roots the least it chosen and if all roots are negative, meaning that the state is moving away from the transition, then positive infinity is returned by the `timeToHit()` method.

Discussion of General Inequalities

A limitation in the approach from above is the restriction that the inequalities must be on the form of Equation (8.2) which only accepts linear expressions. An alternative approach, which has not been pursued, could be to allow general non-linear expressions and then use a numerical approach, e.g. Newton iteration, to determine the least positive root.

This approach has not been pursued for two reasons; the linear form introduced in Equation (8.2) is adequate for the hybrid system models investigated in the remainder of this dissertation and secondly the Newton iteration algorithm does not guarantee convergence in all cases [Kreyzig, 1999], which makes it ill suited for use in an on-line system.

8.3.3 Handling Estimator State Resets

When a hybrid location change results in a state-reset then all inequalities for the new location are evaluated to determine their initial condition and if the set of fulfilled inequalities triggers a transition upon initialisation then the HLC class makes sure that the transition is taken immediately.

However, when the hybrid system model is executed with an estimator in the loop which can also trigger reset events in the integrators a mechanism must be place in the HLC class to detect this and evaluate if the state-change caused any of the equalities to shift logic state. The HLC class checks the trajectory inputs from the integrators and when it is detected that they have been reset, it reinitialises all inequalities and checks for enabled transitions.

This approach is computationally expensive each time the estimator provides a new update, but it is necessary to maintain a consistent logic state of the inequalities that make up the transition equation.

8.4 Declaring Models for Simulation/Execution

The previous section described the infrastructure for simulating/executing hybrid system models, and this section now proceeds to give some detail on the function calls required to set up a hybrid system model using the infrastructure, as well as describe the automated process of translating a model defined in an XML document, into an executable object.

8.4.1 Application Programmers Interface

The integrator blocks and the maps on Figure 8.3 are set up in the same manner as a standard QSS2 simulation, cf. Subsection 3.2.3 on page 36, with the exception that the constructor for each map accepts an array of equation-sets rather than a single set, and has a parameter to indicate which equations set is the initial set. Declaration of the HLC is done with the following call:

```
Qss2HybridLocationControl(int noInputs, int noStates,
                          int noLocations, int currentLocation)
```

Which creates the HLC object and initialises the number of inputs, states, locations and the initial location. Transitions between locations are registered with the object using the following call:

```
addTransition(int fromLocation, int toLocation,
              String transitionEq, String[] reset,
              ArrayList<String> inEvents,
              ArrayList<String> outEvents)
```

Where the first two arguments are the source and destination locations of the transition, respectively. `TransitionEq` is the transition equation, i.e. $j_r(\cdot)$, `reset` is an array of of algebraic expressions of the state and input which describe the state-reset for each state to be executed when the transition is taken. `inEvents` are the event labels that can trigger the transition and `outEvents` are the event labels emitted by the model when the transition is taken - events in the system are represented by text strings.

If the hybrid system model is to be used in on-line control where the states can be reset by other mechanisms than the HLC object this must be indicated to the HLC class. This is indicated by issuing the call:

```
setEstimatorPresent()
```

8.4.2 Translation from XML to Executable

A special class, `XMLModelFactory`, has been developed which parses XML documents as described in Subsection 8.2.2 and translates them into DEVS coupled model following the block diagram of Figure 8.3. This entails:

- creating required maps, integrators and the HLC object
- setting up DEVS connections between the ports of all blocks
- adding all transitions including resets and input/output event specifications

The ability to declare models in an XML document that closely resembles the mathematical model of the system clearly supports the declarative approach pursued in this dissertation, and makes it easier to move from model towards an implementation.

8.5 Simulation of Raibert's Hopper

To illustrate the applicability of the specification of hybrid system models and simulation/execution software developed in this chapter a simulation case study will be presented involving a one legged robot known as Raibert's Hopper, which have previously been simulated in the literature, see e.g. [Back et al., 1993].

8.5.1 Model Description

Raibert's Hopper is a one-legged jumping robot with motion confined to the vertical axis, see Figure 8.5 for a sketch of the physical system.

It consists of a body comprised of two pressurised tanks; P_{init} is a low-pressure tank used to extend the leg during free-flight and P_{th} is a high pressure tank used to boost the robot off the ground. A valve controls which tank is con-

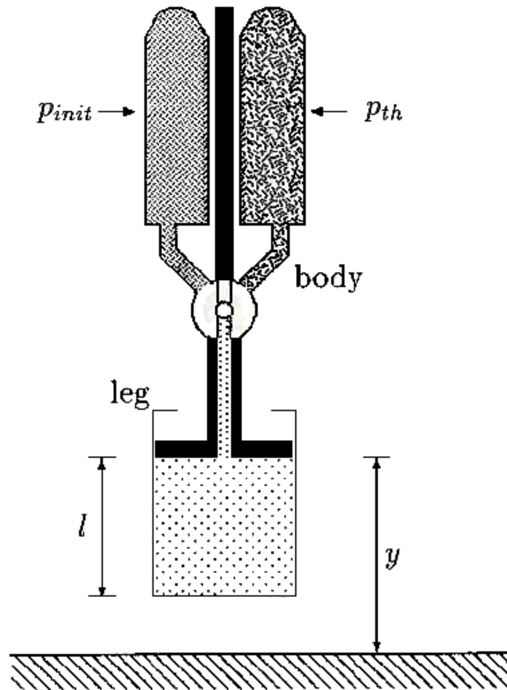


Figure 8.5: Schematic drawing of Raibert's Hopper [Back et al., 1993].

nected to the pneumatic cylinder leg. The system is modelled using four states:

$$\mathbf{x} = [y \;\; \dot{y} \;\; t \;\; \eta_2]^T$$

where y is the position, $\dot{y}$ is the velocity, t is a time-elapse state used in the thrust phase of motion (to be explained), and η_2 is a resettable parameter used in the decompression phase (to be explained) to model the force exerted by the compressed gas in the pneumatic cylinder following the thrust phase. A hybrid model of the system is formulated, see Figure 8.7 for a graphical representation of the associated hybrid system model and Figure 8.6 for illustrations of the state of the physical system in each location.

The behaviour in each of the four locations can be described as:

Flight: Here the hopper is not in contact with the ground and the low-pressure tank is connected to the piston ensuring it is fully extended. The equations of

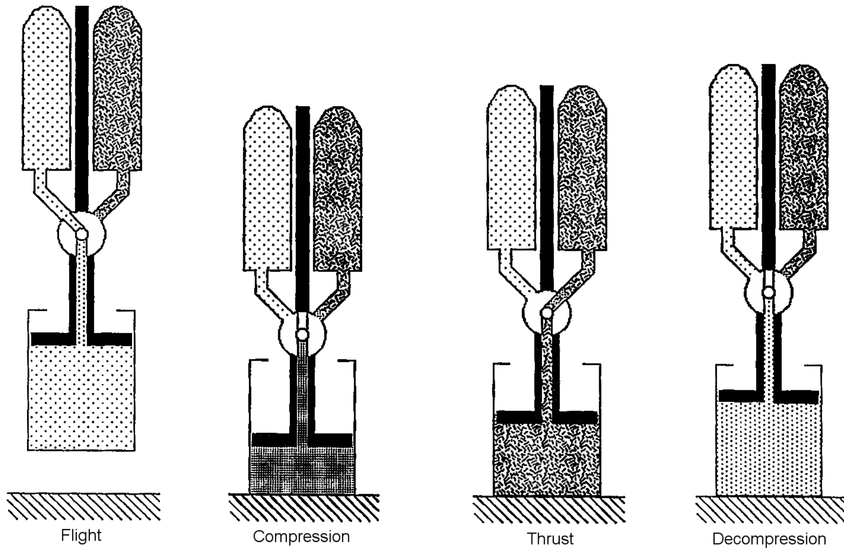


Figure 8.6: Illustration of the state of the physical system in the four different locations: Flight, compression, thrust, and decompression [Back et al., 1993].

motion are:

$$\mathbf{f_1} : \begin{bmatrix} \dot{y} \\ \ddot{y} \\ \dot{t} \\ \dot{\eta_2} \end{bmatrix} = \begin{bmatrix} \dot{y} \\ -g \\ 0 \\ 0 \end{bmatrix}$$

Where g is the acceleration due to gravity. Air drag is assumed negliable. When the bottom of the piston touches the ground (the constant l indicates the height where this happens) the system enters the **Compression** location.

Compression: here the valve is closed and the pressure in the cylinder builds up as it is compressed and exerts an upward directed force, characterised by the constant, η. Kinetic friction, with friction coefficient γ, also acts to reduce energy. The equations of motion are:

$$\mathbf{f_2} : \begin{bmatrix} \dot{y} \\ \ddot{y} \\ \dot{t} \\ \dot{\eta_2} \end{bmatrix} = \begin{bmatrix} \dot{y} \\ \eta/y - \gamma\dot{y} - g \\ 0 \\ 0 \end{bmatrix}$$

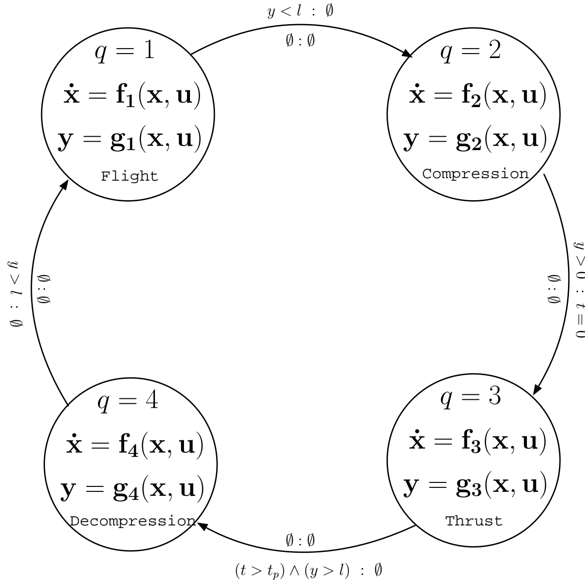


Figure 8.7: Graphical presentation of a hybrid model for Raibert's Hopper with all transitions between locations.

When the spring-like effect of the compression causes the vertical velocity to reach zero the hopper transitions to the **Thrust** location and resets the internal timer state t.

Thrust : In the **Thrust** location the high-pressure tank (p_{th}) gets connected to the piston through the valve for a duration specified by the constant t_p, alternatively the hopper can also transition from the location if the piston becomes fully extended within the active thrust period. The equations of motion are:

$$\mathbf{f_3} : \begin{bmatrix} \dot{y} \\ \ddot{y} \\ \dot{t} \\ \dot{\eta}_2 \end{bmatrix} = \begin{bmatrix} \dot{y} \\ \tau - \gamma \dot{y} - g \\ 1 \\ 0 \end{bmatrix}$$

where τ is the thrust force.

Decompression: At this point if the bottom of the piston has already left the ground the transition to the flight phase is taken immediately, otherwise the hopper continues upwards movement due to the upward speed and the acceleration caused by the force exerted by the compressed gas in the piston. This

force is proportional to the achieved height at the time the valve was closed, i.e. $\eta_2 = \tau y_t$, where y_t is the height when leaving the thrust phase. η_2 is calculated by the reset condition leading to the location. The equations of motion are:

$$\mathbf{f_4} \quad : \quad \begin{bmatrix} \dot{y} \\ \ddot{y} \\ \dot{t} \\ \dot{\eta_2} \end{bmatrix} = \begin{bmatrix} \dot{y} \\ \eta_2/y - \gamma\dot{y} - g \\ 0 \\ 0 \end{bmatrix}$$

In each location the output-map is just an identity map of the states. A full XML declaration of the hybrid system model can be seen in Section A.6 on page 182 of Appendix A, which also gives numerical values for the constants in the model.

8.5.2 Simulation Results

Here simulation results are shown for Raibert's Hopper in order to demonstrate the functioning DEVS/QSS software for simulating/executing hybrid system models. At first an example will be given where the hopper exhibits stable hopping motion and thereafter an example where the energy supplied to the system is not enough to sustain hopping motion resulting in the system becoming "stuck" in one location. For both cases the initial state-space values are:

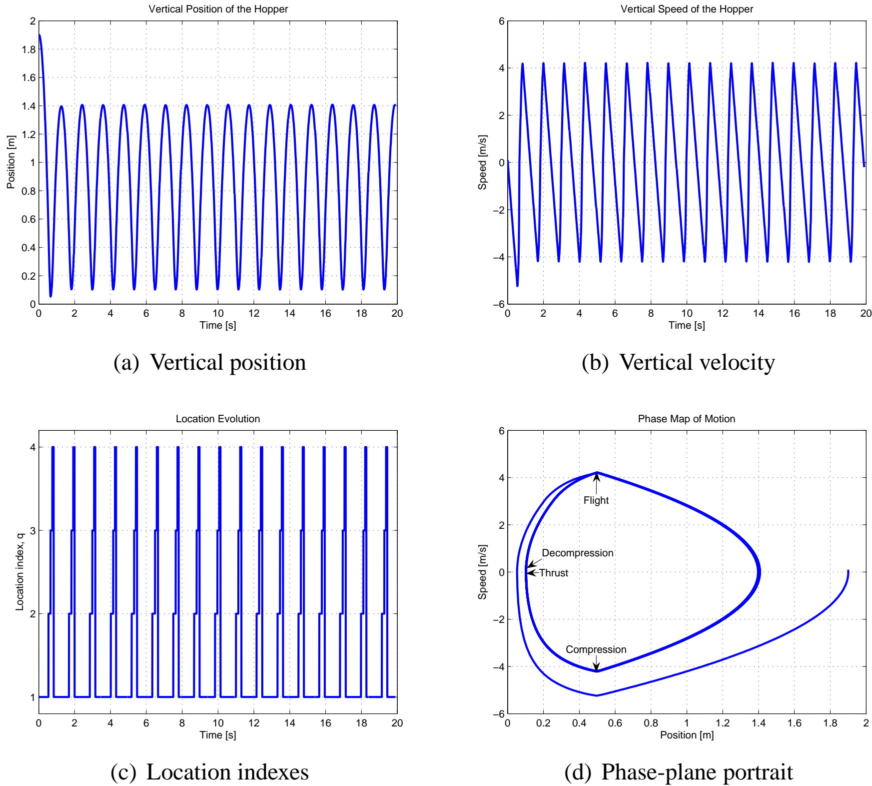
$$\mathbf{x} = [y \;\; \dot{y} \;\; t \;\; \eta_2]^T = [1.9 \;\; 0.1 \;\; 0 \;\; 0]^T$$

Results for the first case are presented on the four graphs on Figure 8.8 and correspond to the following set of parameters for the system:

$$g = -9.82\frac{m}{s^2} \quad l = 0.5m \quad t_r = 0.1s \quad \tau = 40\frac{m}{s^2} \quad \gamma = 0.2 \quad \eta_1 = 8\frac{m^2}{s^2}$$

The (a) and (b) graphs show that after the first hop the motion settles at a very stable orbit. The (c) graph shows the location indexes and it can be seen how the systems transitions according to the diagram of Figure 8.7 in a continuing sequence.

The (d) graph is a phase-plane portrait and it is easy to see how the system finds a stable orbit following initialisation. The graph also shows the points in the orbit where the system transitions between locations. Such stable periodic



(a) Vertical position

(b) Vertical velocity

(c) Location indexes

(d) Phase-plane portrait

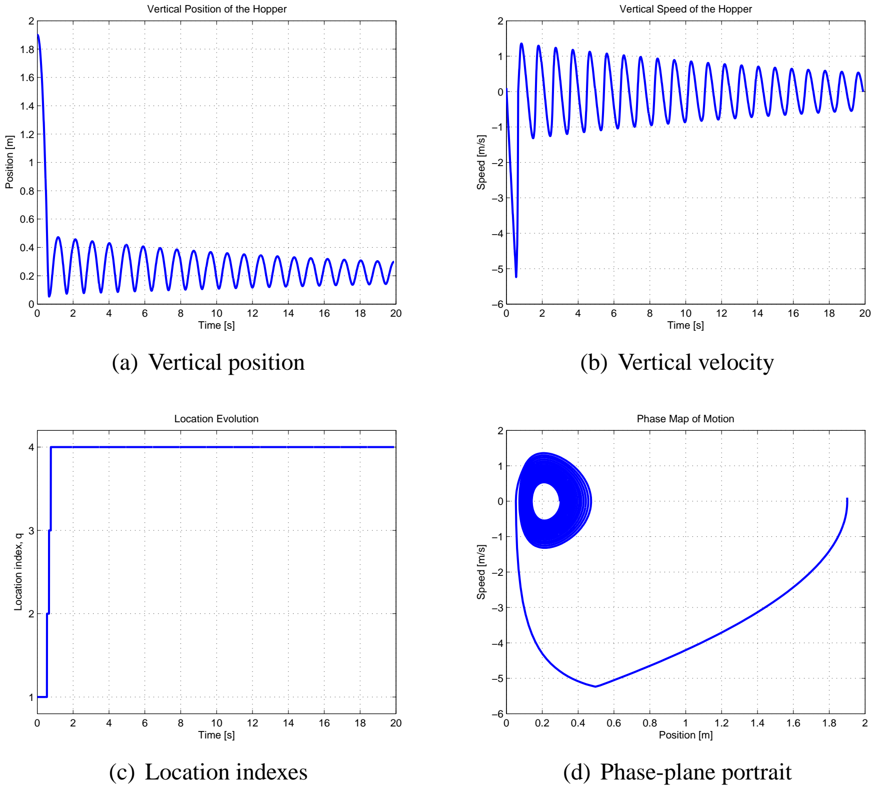
Figure 8.8: Raibert's Hopper under stable hopping motion.

behaviour as seen here is expected from analytical analysis of the system [Vakakis and Burdick, 1990].

Results for the second case, where the upward thrust force has been reduced, are presented on the four graphs on Figure 8.9 and correspond to the following set of parameters for the system:

$$g = -9.82\frac{m}{s^2} \quad l = 0.5m \quad t_r = 0.1s \quad \tau = 20\frac{m}{s^2} \quad \gamma = 0.2 \quad \eta_1 = 8\frac{m^2}{s^2}$$

This time, as evident from the (c) graph, the hopper gets stuck in the **Decompression** location since the supplied thrust is not enough for the system to achieve flight. In this case the spring-effect and the friction causes the system to exhibit a decaying oscillation, which is clearly evident from the (a) and (b) graphs, as well



(a) Vertical position (b) Vertical velocity

(c) Location indexes (d) Phase-plane portrait

Figure 8.9: Raibert's Hopper - Getting stuck in decompression.

as the phase-portrait of the (d) graph.

Performance Discussion

The real-time performance of the software has been analysed using the first case, presented above, by extending the simulation period to 1000s and comparing it to a real-time clock. Execution time on a contemporary lap-top was 2.25s, meaning that the simulation runs at approximately 444 times real time - these result are for QSS2 quanta of $\Delta q = 0.001$. Profiling the simulation shows that 67% of the time is spent in the DEVS framework, 15% is spent evaluating maps, 11% in the HLC class, and 7% in the integrators.

The numbers above gives a feel for the performance, but cannot be applied to predict about the computing resources required by other models as it depends

highly on the quantum selected, the dimension of the state-space and the number of inequalities in each location. For this example the high overhead (Time spent in the DEVS-framework) is due to the simplicity of the model, which makes the evaluation in the maps very efficient.

8.6 Matlab Comparison

In order to demonstrate the value of conservative event detection a simple example has been setup where the QSS based solution is compared to an implementation of Raibert's Hopper implemented in a Matlab script. The Matlab script uses the same equations, transition and constants as in the previous section and implements these in a simulation loop that performs simulation actions according to the sequence:

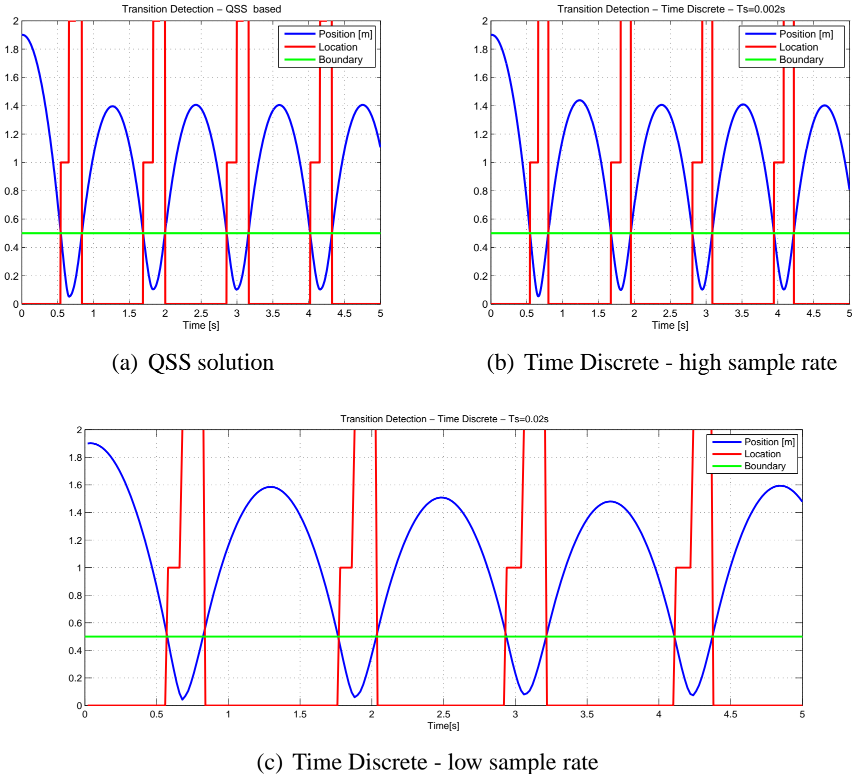
1. Propagate states in current location
2. Check transitions equations and shift location if required
3. Perform reset on entering new location if location changed
4. Proceed from top

Results for this comparison can be seen on Figure 8.10, where graph (a) gives the QSS solution for the first 5 seconds with the blue line being the position, the red line is location indexes (0=free flight, 1=compression, 2=thrust, 3=decompression) and the green line indicates the transition boundary from the free flight location to the compression location. It can be seen from the figure that all three lines meet in the transition points at 0.5m as one would expect. The quantisation for all states is 0.0025.

Graph (b) shows results from the Matlab script with a sample period of 0.002s which is fast enough to give a similar result with transitions detected accurately enough not to qualitatively affect the simulation.

Finally, graph (c) shows what happens when the sample period is set to 0.02s. Here it is visually evident that the new location is only entered at the following sample. The reduced accuracy also leads to qualitative differences in behaviour; in the previously two examples motion stabilises with top points of the parabola at 1.4m, now the general height has increased and seems to vary from cycle to cycle.

The general increase of the top-point is due to the fact that the thrust periods becomes longer due to the transition detection delay. The variance of height is due the variance of the transition detection delay in each for the various transi-



(a) QSS solution

(b) Time Discrete - high sample rate

(c) Time Discrete - low sample rate

Figure 8.10: Transition detection comparison to Matlab.

tion events. Increasing the sample period further results in progressively larger qualitative differences compared to graph (a) and (b).

For a simulation lasting 500s, Matlab takes 1.05s to complete the simulation, while the QSS implementation takes 0.91s. However, these numbers are as much an artifact of the underlying run time system as the specific algorithms in used by the two approaches.

This simple comparative study has demonstrated how conservative event detection, as implemented in the QSS method of this chapter, helps to provide more accurate simulations.

8.7 Chapter Summary

This chapter contributed with a mathematical model of hybrid dynamical systems which is designed to be widely applicable for control systems. Software based on the DEVS framework and quantised state systems was developed for simulation and on-line execution of hybrid system models. The QSS approach allowed state events to be detected conservatively.

Further, a XML file format was specified which makes it easy for a user to supply a hybrid system model (compliant to the presented specification), which is then automatically translated into an executable object. This approach was explored through the Raibert's Hopper Example.

Finally, a simple comparative study demonstrated some of the benefits of the QSS method with conservative event estimation as compared to naive implementation of a hybrid model of Raibert's Hopper.

9 Towards Declarative Hybrid Supervisory Control

This chapter provides an example of hybrid supervisory control based on the Deep Space Probe case. With this example in mind a methodology for declarative hybrid supervisory control is proposed. The proposed methodology builds on the results presented in this dissertation, but still requires some elements to be fully implemented.

9.1 Introduction

During the course of this dissertation a number of tools have been developed to support simulation, estimation and control of dynamic systems based on discrete event interactions between software objects and the notion of quantised state systems. These tools are depicted on Figure 9.1 as the lower three layers in the figure.

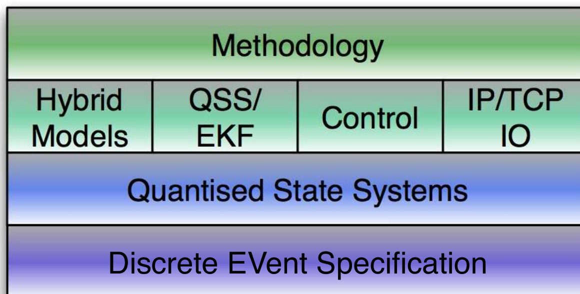


Figure 9.1: Layers of capabilities as developed in the dissertation, a new methodology layer will be added to on top.

This chapter will conclude the dissertation by proposing a methodology based on the developed capabilities for *Hybrid Supervisory Control*. Hereby we mean control of a continuous plant according to a hybrid model describing desired closed loop performance.

The proposed methodology is not fully integrated in the software developed during the dissertation and descriptions of the remaining work to implement the methodology are given. Referring to Figure 1.6 on page 14 the work in this chapter fits in the `Configuration` package.

The next section will at first provide an example of hybrid supervisory control, whereafter Section 9.3 on page 154 will describe the proposed methodology and the required work to implement it from the current state of development.

9.2 Hybrid Supervisory Control Example

The previous chapter introduced hybrid system models and described methods to simulate/execute them using tools based on quantised state systems. Before formulating a methodology, in the next section, for hybrid supervisory control, this section will provide an example of how the tools developed in the previous chapter can be utilised to implement such a hybrid supervisory control system.

The example will utilise the optimising controller developed in Chapter 5 and will be evaluated in the same manner as in Chapter 7, where control inputs drives a "truth model" implemented in Simulink, which also produces measurements for the control model implemented in DEVS/QSS.

9.2.1 Scanning Motion for the Deep Space Probe Case

The following example is based on the Deep Space Probe case as it was presented in Chapter 7 on page 109, where the DSP motion was considered as it conducted a Jovian fly-by. For the following example we will impose the requirement that the DSP, during the fly-by, must perform a scanning motion with one of its sensors; we will require the Euler angle θ_2 to scan the following interval, while the other two Euler angles must be held constant:

$$\theta_1 = 0.1 \ \wedge \ -0.4 < \theta_2 < -0.2 \ \wedge \ \theta_3 = 0.7$$

This behaviour is described using the hybrid model depicted in Figure 9.2

using two discrete locations.

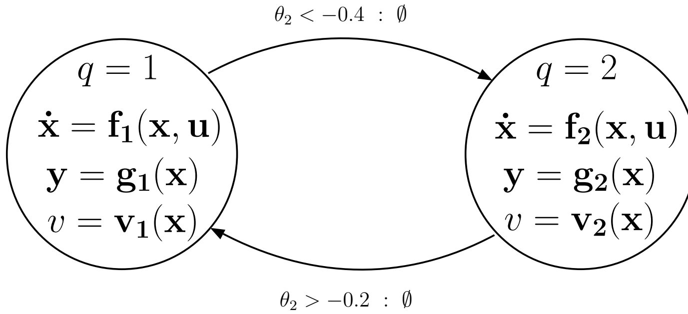


Figure 9.2: Hybrid model for the extended DSP Case

Here $\mathbf{f}_1(\cdot)$ and $\mathbf{f}_2(\cdot)$ represent identical dynamical models, as there are no changes in dynamics for the two locations. Further, $\mathbf{g}_1(\cdot)$ and $\mathbf{g}_2(\cdot)$ represent the measurement models for the sun- and star sensor, which are also identical in both locations. Both the dynamics and the measurement models for the DSP case are stated in Subsection 4.4.1 on page 59.

The functions $\mathbf{v}_1(\cdot)$ and $\mathbf{v}_2(\cdot)$ are performance functions for the control algorithm. We will make use of the optimising controller and adopt the performance function used in Section 7.3 on page 114:

$$v(\theta, \omega) = 30\omega_1^2 + 30\omega_2^2 + 50\omega_2^2 + 5(30(\theta_1 - r_1)^2 + 30(\theta_2 - r_2)^2 + 50(\theta_3 - r_3)^2)$$

with the reference vector, $\mathbf{r} = [r_1 \ r_2 \ r_3]^T$, being determined by the current location:

$$\mathbf{r} = \begin{cases} [0.1 \ -0.5 \ 0.7]^T & \text{if} \quad q = 1 \\ [0.1 \ -0.1 \ 0.7]^T & \text{if} \quad q = 2 \end{cases}$$

With these reference vectors it is certain that the controller in each location will move towards the transition boundary for each location as specified by the transition equations depicted on Figure 9.2.

Supervisory Control Implementation

The hybrid performance function is implemented using the the `HybridMap` class described in Subsection 8.3.1 on page 134 and the transitions are handled using the `HybridLocationControl` class described in Subsection 8.3.2 on page 136. These classes have been introduced into the code developed for the original DSP optimising control case as described in Section 7.3 on page 114.

9.2.2 Simulation Results

Simulations have been run with the supervisory controller described above under the same conditions as in Chapter 7 on page 109, i.e.:

- In closed loop with a "truth" model in Simulink
- Model uncertainties:
  - uncertain moments of inertia
  - uncertain lever-arm vector
- Presence of a parasitic magnetic moment
- Sensor noise

The results of the simulation can be seen on Figure 9.3 from a run that is initialised with conditions of zero Euler angles and an angular velocity corresponding to a slow roll about the major axis of inertia.

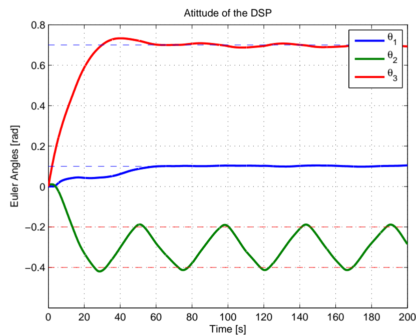
Graph (a) shows how the desired attitude motion of the θ_2 axis is obtained, i.e. a side scanning motion, while the other two Euler angles are held constant. It can be seen that there is some cross-coupling of the motion meaning that the controller must exert effort to maintain the constant Euler angles of θ_1 and θ_3. Graph (b) shows the corresponding angular velocity.

Graphs (c) and (d) show the estimation error of the QSS/EKF filter for the attitude and angular velocity respectively. Graph (e) shows the control input and it is clear that the direction of actuation of the 2nd axis changes as the `HybridLocationControl` class commands location transitions. Finally, graph (f) depicts the disturbance from the parasitic magnetic moment.

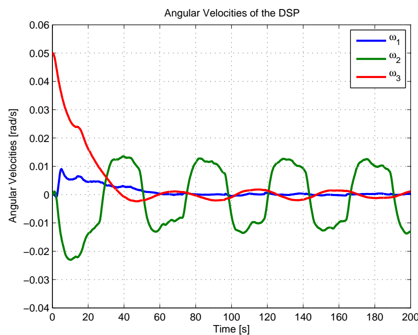
9.3 Towards a Methodological Approach

For complex systems it is desirable to be able to build hybrid supervisory control systems declaratively, i.e. automatically based on descriptions of the system and desired behaviour. The previous example showed an example of a relatively simple system implemented by declaring each part of the system (as DEVS objects in source code) and setting up interconnections.

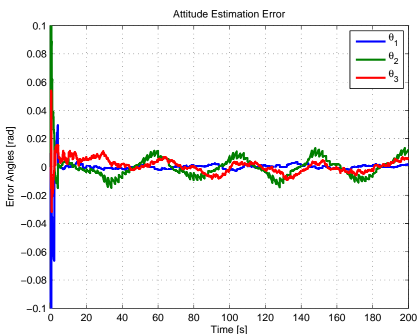
Chapter 8 on page 127 showed how run-time models of hybrid systems could be constructed declaratively from a specification file, in this case using XML. A



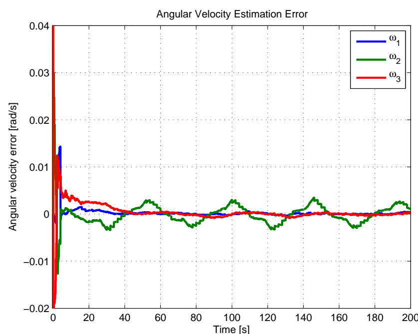
(a) Truth model attitude



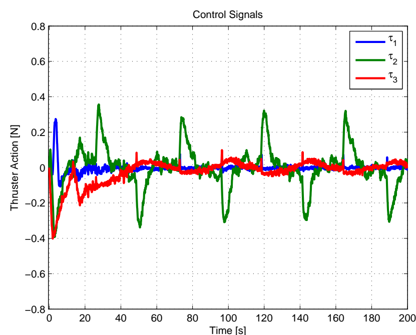
(b) Truth model angular velocities



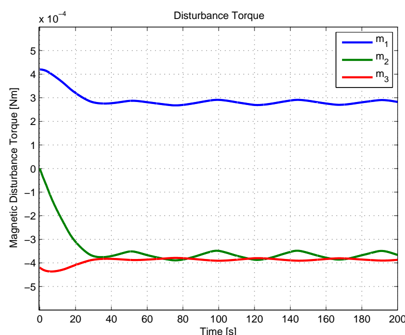
(c) Attitude estimation error



(d) Angular velocities estimation error



(e) Control input signals



(f) Magnetic disturbance

Figure 9.3: Jupiter gravity assist with side-scanning motion.

similar approach should be taken towards specification and automatic construction of complete control systems, as discussed in the Introduction chapter (see Chapter 1 on page 1).

This section proposes a methodology for hybrid supervisory control and points out the remaining work required to implement this methodology based on the tools developed so far during this dissertation.

9.3.1 Proposed Methodology

Figure 9.4 depicts the steps in the proposed methodology for hybrid supervisory control. In each step a part of the control system is declared and composed with the result of the previous step.

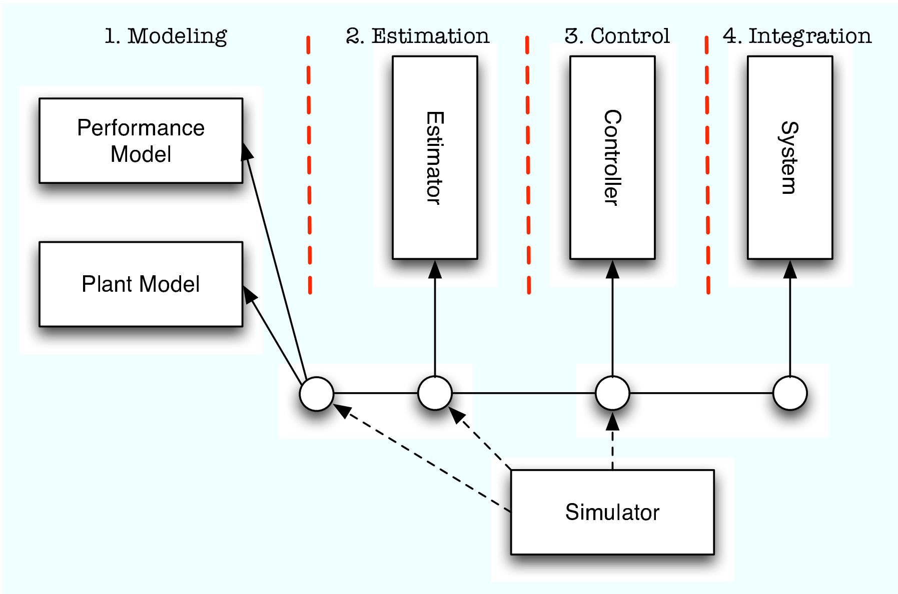


Figure 9.4: Methodology for declarative control design using the tools developed in this thesis. Circles represent software composition, arrows points to the entities being composed, and dotted lines indicate temporary composition.

Each step in the process is described briefly in the following:

Modeling: Here a model of the plant and the performance to achieved by the

plant are declared by the control engineer.

Estimation: An estimator is declared with appropriate parameters and is composed with the model and performance specification from the previous step.

Control: Here a controller type is chosen and declared with proper parameters specific for the type, and composed to the system.

Implementation: Finally, the composite is composed with the run-time system in order to take control of the real plant. Tuning based on achieved performance can be conducted.

Through all the steps it is possible to verify the status of development by composing the composite at that point with a suitable simulation tool; Subsection 2.3.1 on page 26 mentions some of the developed DEVS tools that can be instrumental in this process, e.g. tools to feed a DEVS model at step 2 with measured data from the physical plant to evaluate estimator performance.

At a first glance; the methodology looks very similar to the traditional process of designing and implementing model-based control; however, there are a number of important differences:

1) Each step results in a piece of software that is carried forward in the process unchanged (except for parameters that may be tuned) and integrated with the results of the other steps. This means that when the design is done, so is the software that implements it. A more traditional approach is to test designs using various simulations tools and then when satisfied the actual control code is written to mimic the implementation in the tool used to verify the design.

2) In each step we are not required to implement solutions manually, instead we declare them, e.g. in the *control* step, we declare what algorithm to use, set parameters, and compose it to our system. The alternative would be to implement a given controller manually on each new problem. This allows quick adoption of complicated solution strategies to the problem and quick adaption to e.g. new system parameters. This is possible due to the goal oriented nature of the controllers treated in Chapter 5 and 6.

9.3.2 Required Work to Implement the Proposed Methodology

Given the tools developed during this dissertation, i.e. various Quantised State Components implemented in DEVS, it is possible to encapsulate models, estimation algorithms and control algorithms in contained communicating software

objects.

The example given in Section 9.1 of this chapter demonstrates a use of these objects to implement a hybrid supervisory control system which implements all the steps described in the methodology, however, the example have proceed from step to step by adding new elements to the source code and recompiling. To be a truly *declarative* approach further work is required.

Declarative System Description

Instead of declaring the control system structure by instantiating objects in a compiled language, Java in this, case the control structure should be described in a declarative language separate from the implementation language. The XML approach described in Section 8.2 on page 128 implements this approach for hybrid models.

To extent this approach to cover the whole methodology XML descriptions should be available for each step in the methodology, i.e. one file for the model, one for performance, one for the estimator, one for control and one for the composition. Finally, one file should describe the configuration of the control system, i.e. point to the relevant specific files declaring each part of the system.

First results for a composition architecture along these thoughts have been published in [Alminde et al., 2006b], and a specific example on the use of these results in an architecture that makes use of of QSS/DEVS based controller is described in [Alminde et al., 2007c]. However, systematic adoption and development into a plug'n'play control framework requires further work.

Online Composition

With a system declaration available as just described it is possible to build a complete control structure from declaration including setting up all the relevant bindings, i.e. communication channels, between the objects implementing the entities being declared.

This process can either take place as a sequential build process or the individual components can be instantiated in parallel and then be put into the declared structure by subsequent on-line manipulation.

The first approach is the one implemented with the QSS/DEVS tools described during this dissertation, but as already discussed in Section 2.4 on page 27 it is an interesting avenue of further work to develop from the DEVS formalism an evolved encapsulation mechanism which is more rich in information about itself

and in methods to allow on-line manipulation of the objects.

9.4 Chapter Summary

This chapter has provided an example demonstrating how the components developed throughout the dissertation using DEVS and quantised state systems can be used to implement a hybrid supervisory control system. Furthermore, a methodology to handle such systems declaratively has been proposed and it has been discussed what remaining work lies ahead to implement this methodology.

Part IV

Closure

10 Concluding Remarks

This final chapter summarises the results of the thesis and evaluates them against the research objectives formulated in the Introduction Chapter of the thesis. Conclusions from the work is drawn, and, finally, a number of areas worth further work are presented and discussed.

10.1 Summary of the Results

The following provides a chapter by chapter summary of the contents and results reported in this thesis.

Chapter 1: Introduction
The introductory chapter proposed the development of a declarative control system to enable advanced control and estimation algorithms to be used as solutions to real-life challenges with reduced development effort. This idea was transformed into three research objectives that were pursued in this thesis. The three objectives are restated here:

Research Objective 1: *"To provide and demonstrate a framework that allows control/estimation algorithms and plant models to be described independently and then be composed at run time"*

Research Objective 2: *"To demonstrate the applicability of object oriented design to the domain of control systems software for on-line execution"*

Research Objective 3: *"To demonstrate and evaluate a Quantised State Systems approach to control systems software in contrast to typical sample driven implementations"*

Further, an overview of related work was presented and the structure of the thesis and its scientific contributions were summarised.

Chapter 2: Discrete Event Systems
The chapter introduced the DEVS representation, which has been used throughout the thesis to implement the methods and algorithms developed. In summary a DEVS model is made up of atomic and coupled models contained in a top-level coupled model that can be executed by a runner object.

A software framework was developed in Java which implements the specified DEVS capabilities. Further, issues concerning the communication model used in DEVS was discussed. The merit of the DEVS approach when implementing control systems software is the encapsulation and compositionality that it can provide for software components, such that they can be reused in various contexts with no changes.

Chapter 3: Quantised State Systems
Initially it was argued that numerical simulation algorithms are an important part of advanced control and estimation approaches. An alternative to well known discrete-time methods was presented, which relies on quantisation of the states. The merit of this approach is automatic adjustment of the required number of integration steps to the level of change experienced by the solution trajectory - and further a decoupling of states in the calculations that can exploit sparsity.

A specific QSS based algorithm, the QSS2 algorithm, was presented in detail. The algorithm relies on a first order quantisation of the state and maintains internally Jacobian matrices for the system being propagated.

The properties of the QSS2 algorithm was discussed and demonstrated through both simple illustrative examples and a more intensive simulation study of an autonomous underwater vehicle. It was demonstrated that the QSS2 algorithm has performance and robustness features that makes it interesting for use in control applications. To the author's best knowledge the simulation study currently is the most comprehensive study of a higher dimensional non-linear system being simulated with the QSS2 algorithm.

Chapter 4: Kalman Filter Estimation in QSS
The chapter introduced the QSS/EKF filter which is an extended Kalman filter

implemented for use with quantised state systems. A case study concerning attitude determination for a deep-space probe demonstrated that the QSS/EKF filter performs almost identically to the traditional sample-based EKF.

Contrary to the original EKF algorithm, the QSS/EKF alternative does not require analytical expressions for the state and measurements Jacobians respectively, but instead Jacobians are provided at no additional computational cost by the QSS2 algorithm used for state propagation. For systems where it is impractical or impossible to analytically derive expressions for the Jacobian, or where such expressions becomes very computationally expensive the QSS/EKF algorithm provides an interesting alternative to the conventional EKF algorithm.

Secondly, the QSS/EKF filter is a reusable implementation that effectively encapsulates the algorithm and only requires the user to specify the model of the system and associated measurements as a QSS2 model. The QSS/EKF algorithm and the model can then be composed at run-time.

Chapter 5: Optimising Control of QSS Systems
The chapter introduced two algorithms for control of a class of non-linear multiple-input-multiple-output systems based on QSS2 models of the system and a QSS2 description of a control objective function which is minimised by the choice of control input slopes by the controller.

It was shown that stability of the proposed method must be viewed in the framework of switched/hybrid systems and depends on the users choice of control objective function and control cost matrix. Further, it was shown that if a quadratic control objective function is chosen the control strategy is equivalent to the *min-skew-projection* strategy described in [Pettersson and Lennartson, 1997].

In practice the method can be applied to a large number of systems, e.g. motion control system with inherent dynamical dampening, where the control objective it to guide the kinematical states to a given set-point.

The method was demonstrated using simulations of an autonomous underwater vehicle, based on a nominal model and full state knowledge, and it was demonstrated that the method was successful in controlling the system. Both the single objective and multiple objective control variant of the method was demonstrated successfully.

Chapter 6: Sliding Mode Control in QSS Systems
The chapter introduced sliding mode control for quantised state systems and de-

veloped a controller structure, and corresponding software implementation, that can stabilise a large class of non-linear systems, even in the presence of disturbances and model uncertainties.

The proposed controller structure is highly adaptable and each component can be replaced by customised elements to suit specific application requirements. The approach was verified on a deep space probe attitude control example.

The presented algorithm concerns state stabilisation, however, the proposed structure can easily be adapted to provide tracking; a suitable methodology is developed in [Khalil, 2000].

Chapter 7: Evaluation of Estimation Based Control
The chapter described a case for evaluating the algorithms developed in the preceding chapters under conditions that resembles those experienced by real-life control problems, i.e. including effects of disturbances, uncertainties, estimation errors, and synchronisation issues.

Using the case the controller structures developed in the previous two chapters were evaluated when driven by estimates obtained from the QSS/EKF estimator. A truth model was implemented in Simulink and input/output signals between the Simulink model and the DEVS based control/estimation algorithms were facilitated by a special adaptor class to provide rendezvous based state synchronisation.

The optimising control scheme showed itself to be able to guide the attitude to the desired reference, but it was not very robust against disturbances. The sliding mode controller showed excellent performance in terms of reaching time and disturbance rejection, but is sensitive to estimation errors.

Chapter 8: Hybrid Systems and QSS Based Simulation
The chapter contributed with a mathematical model of hybrid dynamical systems which is designed to be widely applicable for control systems. Further, software based on the DEVS framework and quantised state systems was developed for simulation and on-line execution of hybrid models.

Next, a XML file format was specified which makes it easy for a user to supply a hybrid model (compliant to the presented specification), which is then automatically translated into an executable object. This approach was explored through the Raibert's Hopper Example. The QSS approach allowed state events to be detected conservatively.

Finally, a simple comparative study demonstrated some of the benefits of the

QSS method with conservative event estimation as compared to naive implementation of a hybrid model of Raibert's Hopper.

Chapter 9: Towards Declarative Supervisory Control
The chapter provided an example demonstrating how the components developed throughout the thesis using DEVS and quantised state systems can be used to implement a hybrid supervisory control system. Furthermore, a methodology to handle such systems declaratively was proposed and it was discussed what remaining work lies ahead to implement this methodology.

10.2 Conclusions on Research Objectives and Contributions

To evaluate the contributions of the thesis they are related to the research objectives that were stated in Chapter 1 on page 1. The first objective was formulated as:

Research Objective 1: *"To provide and demonstrate a framework that allows control/estimation algorithms and plant models to be described independently and then be composed at run time"*

This thesis has addressed this issue and developed the QSS/EKF estimation algorithm which is set-up by composing it with a QSS2 model describing the plant, and declaring uncertainty parameters. Further, two distinct control algorithms were developed; one based on local optimisation and one based on sliding mode control. Both rely on composition with a QSS2 model of the plant and a desired control objective.

The *DevsRendendevouzAdaptor* interface introduced in Subsection 7.2.1 on page 113 allows The complete control software implementation to be composed with different system, e.g. a simulator or the plant hardware, without modification other than adoption of a Java interface to implement input and output functionality.

Further, the chapter on hybrid systems has demonstrated the idea of using XML descriptions of hybrid systems model, which closely reflects the mathematical model, as a tool for automatic declarative composition of hybrid system models for simulation and execution.

Research Objective 2: *"To demonstrate the applicability of object oriented design to the domain of control systems software for on-line execution"*

Object oriented programming has been used throughout the work presented in the thesis and provides a high degree of modularity and encapsulation of the implemented functionality. The sliding mode control chapter is a good example of this where it was possible to inherit the `Qss2Static` class and modify its behaviour to form the guidance controller of the sliding mode controller structure.

Again the *DevsRendendevouzAdaptor* interface introduced in Subsection 7.2.1 on page 113 demonstrates the value of the object oriented approach by providing the flexibility to use the control software in different environments by only implementing a specific input/output handler for the environment.

Research Objective 3: *"To demonstrate and evaluate a Quantised State Systems approach to control systems software in contrast to typical sample driven implementations"*

Quantised state systems and specifically the QSS2 algorithm have been used extensively throughout the thesis and it has been shown that it is a paradigm that fits well with the two research objectives listed above. The thesis contributes with novel algorithms which can be used with quantised state systems in applications involving hybrid simulation, estimation and control.

This thesis and associated publications demonstrates the first use of the QSS2 algorithm in estimation problems, and the first direct exploitation of the QSS2 algorithm and its structure for control algorithms. Previous control work, e.g. [Kofman, 2003] has investigated the Quantised State Approach as an implementation method for control laws based on traditional analytical methods for linear systems, i.e. as an alternative to difference equations.

It is still not as easy to apply advanced control theory as it is to remove red eyes from a photograph in Photoshop - as the ultimate goal was formulated in the introduction; however, the approach pursued in this thesis has lead to algorithm implementations based on quantised state system which allows on-line composition of model-elements (dynamics and performance) with algorithms for estimation and control, without having the control engineer to manually write any control code.

10.3 Recommendations for Future Work

This final section puts attention to a number of issues that the author feels are important to discuss as possible avenues of further work within the topics treated in this thesis.

10.3.1 Real-Time Issues

There has been no stringent treatment of real-time issues which is highly relevant for any control system. In Chapter 7 and Chapter 9 the algorithms were tested in closed loop with a high-fidelity simulation model. These tests did not indicate real-time issues, but clearly for applications where the computational delays are significant, relative to the physical system being controlled, issues can develop.

An analysis of these issues for the DEVS and QSS based methods is challenging as the computational delays will vary in a non-deterministic way. This is contrary to traditional sample-based methods were delays often can be measured and incorporated in the control design.

One intriguing idea worth further investigation is the use of the quantums, used in the QSS2 algorithm, as a feedback quantity in order to regulate the current CPU-usage to a preallocated share of the total CPU time. In this manner the control system will apply more fine-grained control when the system is close to resting and apply a more coarse control during set-point transitions.

10.3.2 Control Beans

Emphasis has been put on encapsulating algorithms in a manner such that they can be used in a number of applications with no changes, DEVS and the QSS approach have been instrumental in this approach. Future work should integrate changes in the DEVS specification as discussed in Section 2.4 on page 27 concerning formalisation of non-subscription based communication between components. Further, the implementation of the DEVS framework should consolidate a rich Application Programmers Interface (API) for the `DevsAtomic` class which not only covers DEVS execution functionality, but also the capability for tools to interrogate the capabilities of each object.

This would lead to a kind of *"controlBeans"* specification, which can be utilised by tools that declaratively builds solution implementations to user specified prob-

lems. This idea is closely related to the use of *JavaBeans* [Sun_Microsystems, 2007b] used by tools that automatically builds code for graphical user interfaces from a user supplied specification of the desired result and *Java Enterprise Beans* [Sun_Microsystems, 2007a] used in a similar fashion for business applications.

10.3.3 The Configuration Layer of the Declarative Control System

The introduction talked about a required `Configuration` module in the Declarative Control System Architecture, see Subsection 1.1.2 on page 3, with the following responsibility:

Configuration this module is responsible for analysing the user supplied models and set up relations (at run time) between model elements and algorithms, both estimation and control. This includes choosing which algorithms are best suited to the problem.

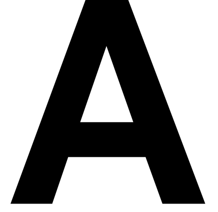
This module has not received much attention during this thesis as focus has been on developing methodologies of the more basic architectural components. However, Chapter 8 on page 127 on hybrid systems introduced the use of XML files to declare hybrid system models and Chapter 9 on page 151 proposed a methodology to extent this approach to cover full specification of control systems - some of these ideas has been described as as part of the *Simulation, Observation and Planning in Hybrid Systems (SOPHY)* project [Alminde et al., 2006b,Laursen et al., 2005] were the use of XML files to describe both dynamical model components, hardware interfaces, and composite system structures was explored, but it remains to fully integrate this work with the QSS approach.

Consolidating this approach and using the XML formats as a back-end for a graphical user interface similar to e.g. *Simulink* in appearance will provide a convenient mechanism for control engineers to provide system descriptions and a platform for algorithms to analyse the specified system and suggest to the user which *controlBeans* objects should be chosen for automatic composition with the model to form suitable solution strategies.

Part V

Appendices

Hybrid Systems and XML Specifications



This note formally defines hybrid systems and their compositions as interpreted within the Sophy working group - A distinction is made between deterministic and probabilistic systems. Also, specifications for the associated XML file format is given as well as an example hereupon.

A.1 Definition of a Hybrid System

In the following $\mathbb{R}^n$ will denote the n-dimensional Euclidean space and $\mathbb{Z}^+$ will denote the smallest inductive set, i.e. the positive integers. A hybrid system is an 8-tuple:

$$\mathcal{H} = (Q, X, U, Y, E, \mathcal{F}, \mathcal{G}, \mathcal{T}) \tag{A.1}$$

Where:
$Q = \{q \in \mathbb{Z}^+ | 1 \leq q \leq s\}$: is a set of location indexes with cardinal number $s \in \mathbb{Z}^+$
$X = \{\{x | x \in X_q\}_{q \in Q} | X_q = \mathbb{R}^{n_q}\}$: is the state-space with dimension $n_{q \in Q} \in \mathbb{Z}^+$
$U = \{\{u | u \in U_q\}_{q \in Q} | U_q = \mathbb{R}^{m_q}\}$: is the input-space with dimension $m_{q \in Q} \in \mathbb{Z}^+$
$Y = \{\{y | y \in Y_q\}_{q \in Q} | Y_q = \mathbb{R}^{o_q}\}$: is the output-space with dimension $o_{q \in Q} \in \mathbb{Z}^+$
$E = \left\{e | e \in 2^{\Sigma}\right\}$: is the set of possible input/output event labels, where Σ is a set of labels

$\mathcal{F} : Q \times X \times U \mapsto \dot{X}$: is the forcing functions on the continuous state-space
$\mathcal{G} : Q \times X \times U \mapsto Y$: is a continuous output map
$\mathcal{T} : Q \times X \times U \times E \mapsto Q \times X \times E$: is a transition map

Remarks

- Time is not explicitly given in the definition of the system, however, with no loss of generality the modeller can include an extra state in the continuous map to represent explicit time
- In most practical applications the dimensions of the state-, input-, and output-spaces will not change with different $q \in Q$
- The map $\mathcal{F}$, as defined above, allows Ordinary Differential Equations (ODE), but not e.g. differential algebraic or partial differential equations

A.2 Specialised Specifications

The above definition is abstract and contains little information about how the maps are to be implemented in practice or how the initial state is defined. This section imposes restrictions on the above definition in order to define a Hybrid Deterministic System (HDS) and a Hybrid Probabilistic System (HPS), and gives examples of how concrete specifications can be implemented under these restrictions.

A.2.1 Hybrid Deterministic System (HDS)

A HDS imposes the following restrictions on the above definition:

- The maps, $\mathcal{F}, \mathcal{G}$, and $\mathcal{T}$, must be deterministic functions of the state and input
- At any time the total state of the HDS is defined by the triple: $\mathcal{S} = (q \in Q, x \in X_q, u \in U_q)$
- The initial state of a HDS is defined by: $\mathcal{S}_0 = (q_0 \in Q, x_0 \in X_{q_0}, u_0 \in U_{q_0})$
- If the total state is indexed with $q \in Q$, e.g. $\mathcal{S}_q$, it means that the location is fixed, thus: $\mathcal{S}_q = X_q \times U_q$

To define a HDS the initial total state must be included in the definition, further to make specification of the HDS more convenient the maps $\mathcal{F}$ and $\mathcal{G}$ will be defines as sets of functions with index $q \in Q$ and the transition map will be broken up into a set of different maps:

$$\mathcal{H}^{HDS} = (Q, X, U, Y, E, \mathcal{F}, \mathcal{G}, \mathcal{T}, \mathcal{S}_o) \tag{A.2}$$

where:
Q, X, U, Y, E: are defined as before

$\mathcal{F} = \left\{ \{f_q\}_{q \in Q} | \{q\} \times X_q \times U_q \mapsto \dot{X}_q \right\}$: is the set of forcing functions on the continuous state-space
$\mathcal{G} = \left\{ \{g_q\}_{q \in Q} | \{q\} \times X_q \times U_Q \mapsto Y_q \right\}$: is the set of continuous output maps
$\mathcal{T} = \left\{ \{t_r\}_{r \in \{1,..,p\}} | Q \times X \times U \times E \mapsto Q \times X \times E \right\}$: are transition maps indexed from 1 to p

Where each transition is described as a 4-tuple:

$$\tau_r = \left(j(\mathcal{S}_q), r(\mathcal{S}_q), e_{in} \in 2^{\Sigma}, e_{out} \in 2^{\Sigma} \right) \tag{A.3}$$

where:
$j(\mathcal{S}_q) : \mathcal{S}_q \mapsto \{true, false\}$: is the *transition domain* which triggers the transition when true
$r(\mathcal{S}_q) : \mathcal{S}_q \mapsto Q \times X$: is an algebraic reset equation of the state
e_{in}: is an input event that causes the transition to trigger
e_{out}: is an output event that is emitted when the transition is taken

In this definition the use of the location indexed state $\mathcal{S}_q$ rather than $\mathcal{S}$ makes it convenient to group transitions, τ_r, according to source location. For purposes of implementation the transition domain must be specified as a number of logical combined inequalities, example:

$$j(\mathcal{S}_q) = j_1(\mathcal{S}_q) > 0 \ \wedge \ (j_2(\mathcal{S}_q) > 0 \ \vee j_3(\mathcal{S}_q) > 0) \tag{A.4}$$

A.2.2 Hybrid Probabilistic System (HPS)

A HPS imposes the following restrictions on the above definition:

- The maps $\mathcal{F}$ and $\mathcal{G}$ are deterministic functions in the same way as for the HDS but may accept inputs that are functions of stochastic processes, i.e.

  $$\mathcal{F} : f_q(x, u) = f'_q(x, u, w) | w = W \sim P$$

  where w is the output of a stochastic process W with a distribution given by the probability density function P.

- $\mathcal{T}$ may contain a $j(\mathcal{S}_q)$ that generates a transition in case $u \leq j(\mathcal{S}_q)$, where $j(\mathcal{S}_q)$ is an appropriate likelihood function and u is random variable with a constant probability in the interval $[0; 1]$, i.e. the output of a uniformly distributed process.

A.2.3 Constant Dimension Systems (CDS)

In many applications of hybrid systems the state, input and output space remains the same in all locations, these we will call Constant Dimension Systems (CDS). For these system we can associate fixed vectors to represent the spaces, as given by:

- state-vector: $\mathbf{x} = X_{q \in Q}$
- input-vector: $\mathbf{u} = U_{q \in Q}$
- output-vector: $\mathbf{y} = Y_{q \in Q}$

A HDS or HPS system can also be CDS at the same time, with appropriate changes in notation.

A.3 Composition of Hybrid Systems

In the following section we define the parallel composition of two hybrid systems, $\mathcal{H}_1$ and $\mathcal{H}_2$, as shown in Figure A.1, defining a new hybrid system $\mathcal{H}_3$. Composing two hybrid systems into one entails that:

- Input, $\mathbf{u}_{\mathcal{H}_3}$, and output, $\mathbf{y}_{\mathcal{H}_3}$, to the composed system is selected.
- Mapping functions, that maps the input to the composed system and the output from the two hybrid systems, $\mathbf{y}_{\mathcal{H}_1}$ and $\mathbf{y}_{\mathcal{H}_2}$, to the input to the two hybrid systems, $\mathbf{u}_{\mathcal{H}_1}$ and $\mathbf{u}_{\mathcal{H}_2}$, and the output from the composed system are selected.

The parallel composition offers the possibility of modelling a complex hybrid system as a number of sub-models instead of a monolithic hybrid system. The composition is parallel in the sense that the execution of the models happens concurrently.

The composition of two hybrid systems is defined over the domain:

$$||_M : \mathcal{H} \times \mathcal{H} \rightarrow \mathcal{H}, \tag{A.5}$$

such that the inputs to the composition are two hybrid systems:

$$\mathcal{H}_1 = (Q_1,\, X_1,\, U_1,\, Y_1,\, \Sigma_1,\, \mathcal{F}_1,\, \mathcal{G}_1,\, \mathcal{T}_1) \tag{A.6}$$

$$\mathcal{H}_2 = (Q_2,\, X_2,\, U_2,\, Y_2,\, \Sigma_2,\, \mathcal{F}_2,\, \mathcal{G}_2,\, \mathcal{T}_2) \tag{A.7}$$

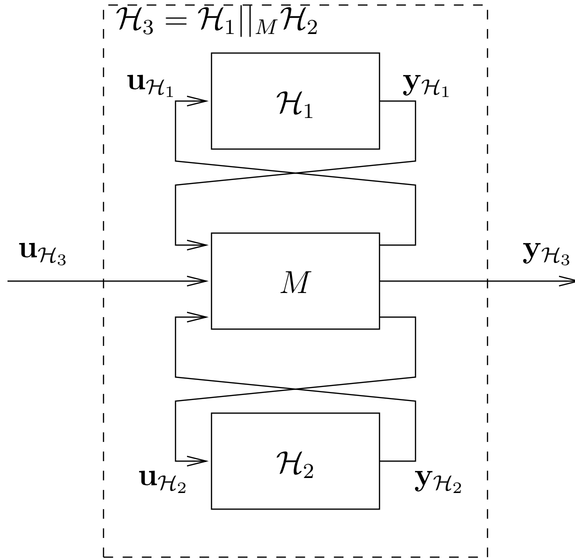


Figure A.1: Composition of two hybrid systems

and the composition operator is defined by a set of matrices:

$$\mathcal{M} = \{\mathbf{M}_{(q_1,\, q_2)} \mid (q_1,\, q_2) \in Q_1 \times Q_2\}, \tag{A.8}$$

that for each possible combination of hybrid locations in the two systems maps the output from both hybrid systems and the input vector to the composed system $\mathbf{u}_{\mathcal{H}_3}$ to the old input vectors $\mathbf{u}_{\mathcal{H}_1}$ and $\mathbf{u}_{\mathcal{H}_2}$ and the output vector from the composed system in the following way:

$$\begin{bmatrix} \mathbf{u}_{\mathcal{H}_1} \\ \mathbf{u}_{\mathcal{H}_2} \\ \mathbf{y}_{\mathcal{H}_3} \end{bmatrix} = \mathbf{M}_{(q_1,\, q_2)} \begin{bmatrix} \mathbf{y}_{\mathcal{H}_1} \\ \mathbf{y}_{\mathcal{H}_2} \\ \mathbf{u}_{\mathcal{H}_3} \end{bmatrix}. \tag{A.9}$$

Two hybrid systems are M-composable, if for all modes $q_1 \in Q_1$ and $q_2 \in Q_2$ there exists a solution to

$$\begin{bmatrix} \mathbf{u}_{\mathcal{H}_1} \\ \mathbf{u}_{\mathcal{H}_2} \\ \mathbf{y}_{\mathcal{H}_3} \end{bmatrix} = \mathbf{M}_{(q_1,\, q_2)} \begin{bmatrix} \mathbf{y}_{\mathcal{H}_1} \\ \mathbf{y}_{\mathcal{H}_2} \\ \mathbf{u}_{\mathcal{H}_3} \end{bmatrix} = \mathbf{M}_{(q_1,\, q_2)} \begin{bmatrix} \mathcal{G}_1(q_1,\, \mathbf{x}_{\mathcal{H}_1},\, \mathbf{u}_{\mathcal{H}_1}) \\ \mathcal{G}_2(q_2,\, \mathbf{x}_{\mathcal{H}_2},\, \mathbf{u}_{\mathcal{H}_2}) \\ \mathbf{u}_{\mathcal{H}_3} \end{bmatrix} \tag{A.10}$$

given as

$$\mathcal{M}\left((q_1, q_2), \begin{bmatrix} \mathbf{x}_{\mathcal{H}_1} \\ \mathbf{x}_{\mathcal{H}_2} \end{bmatrix}, \mathbf{u}_{\mathcal{H}_3}\right) = \begin{bmatrix} \mathbf{u}_{\mathcal{H}_1} \\ \mathbf{u}_{\mathcal{H}_2} \\ \mathbf{y}_{\mathcal{H}_3} \end{bmatrix}, \tag{A.11}$$

or in other words there may not be any algebraic loops in the composition.

A.3.1 Composition of CDS Systems

The above general discussion of composition will require a $\mathbf{M}$ to be defined for any combination of locations in the two systems. The following will define the composition of two CDS systems needing only one $\mathbf{M}$ matrix. While losing some expressivity, the benefits are:

- Only one composition matrix must be specified (maintain sanity of the engineer at all costs)
- Communication channels can be set up statically for a distributed system

The hybrid composition is defined as:

$$\mathcal{H}_1^{CDS}||_M\mathcal{H}_2^{CDS} = \mathcal{H}_3^{CDS} = (Q_3, \mathbf{x}_3, \mathbf{u}_3, \mathbf{y}_3, \Sigma_3, \mathcal{F}_3, \mathcal{G}_3, \mathcal{T}_3), \tag{A.12}$$

with two input systems:

$$\mathcal{H}_1^{CDS} = (Q_1, \mathbf{x}_1, \mathbf{u}_1, \mathbf{y}_1, \Sigma_1, \mathcal{F}_1, \mathcal{G}_1, \mathcal{T}_1) \tag{A.13}$$

$$\mathcal{H}_2^{CDS} = (Q_2, \mathbf{x}_2, \mathbf{u}_2, \mathbf{y}_2, \Sigma_2, \mathcal{F}_2, \mathcal{G}_2, \mathcal{T}_2) \tag{A.14}$$

and a composition matrix of real numbers:

$$\begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{y}_3 \end{bmatrix} = \mathbf{M} \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \mathbf{u}_3 \end{bmatrix} \tag{A.15}$$

with the following restrictions to avoid algebraic loops:

- there must only be zeros between $\mathbf{y}_1$ and $\mathbf{u}_1$
- there must only be zeros between $\mathbf{y}_2$ and $\mathbf{u}_2$

The dimensions of $\mathbf{u}_3$ (m) and $\mathbf{y}_3$ (o) is implicitly defined by the dimensions of $\mathbf{M}$, the inputs and outputs. The following spaces then follows from the composition operation:

$Q_3 = \{q_3 : q_3 \in Q_1 \times Q_2\}$: is the location set of the composed hybrid system
$\mathbf{x}_3 = [\mathbf{x}_1^t \ \mathbf{x}_2^t]^t$: is the new of continuous state space with dimension $n_3 = n_1 + n_2$
$E_3 = \left\{e | e \in 2^{(\Sigma_1 \cup \Sigma_2)}\right\}$: is the set of possible input/output events

The maps of the composed system is defined as:
$\mathcal{F}_3 : Q_3 \times \mathbf{x}_3 \times \mathbf{u}_3 \rightarrow \dot{\mathbf{x}_3}$: is the forcing functions on the continuous state space

$$\mathcal{F}\left(q_3, \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}, \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix}\right) = \begin{bmatrix} \mathcal{F}_1(proj_{Q_1} q_3, \mathbf{x}_1, \mathbf{u}_1) \\ \mathcal{F}_2(proj_{Q_2} q_3, \mathbf{x}_2, \mathbf{u}_2) \end{bmatrix} \tag{A.16}$$

$\mathcal{G}_3 : Q_3 \times \mathbf{x}_3 \times \mathbf{u}_3 \rightarrow \mathbf{y}_3$: is the continuous output map

$$\mathcal{G}\left(q_3, \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix}\right) = \begin{bmatrix} \mathcal{G}_1(proj_{Q_1} q_3, \mathbf{x}_1, \mathbf{u}_1) \\ \mathcal{G}_2(proj_{Q_2} q_3, \mathbf{x}_2, \mathbf{u}_2) \end{bmatrix} \tag{A.17}$$

$\mathcal{T}_3 : Q_3 \times \mathbf{x}_3 \times \mathbf{u}_3 \times 2^{\Sigma_3} \rightarrow Q_3 \times \mathbf{x}_3 \times 2^{\Sigma_3}$: is the transition map

$$\mathcal{T}\left(q_3, \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix}, e\right) = \begin{bmatrix} \mathcal{T}_1(proj_{Q_1} q_3, \mathbf{x}_1, \mathbf{u}_1, e) \\ \mathcal{T}_2(proj_{Q_2} q_3, \mathbf{x}_2, \mathbf{u}_2, e) \end{bmatrix} \tag{A.18}$$

Remarks

- As seen from the composition; Spaces are simply merged and $\mathbf{M}$ distributes information to $\mathcal{H}_1^{CDS}$ and $\mathcal{H}_2^{CDS}$ as appropriate.
- As the new $\mathcal{H}_3^{CDS}$ itself is a hybrid CDS then the composition is closed.

A.4 Overview of XML Tags for Defining a Hybrid System

This section described the *SophySystem* DTD-document which describes who a subsystem must be declared. The table below shows the structure of the DTD document. Note that: * - indicates zero or more, + indicates 1 or more, ? - indicates 0 or 1, also identifiers in parenthesis indicates a attribute for the corresponding element.

Element	Data contained in element, if any
SophySystem	
name	The name of the subsystem
documentation?	Text describing the subsystem model
hints?	Collection of "hints" - see below
hint+(name)	Name (in attribute) and value (string) pair for hint
constants?	Collection of constants to be substituted in equations
constant+(name)	Name (in attribute) and value (string) pair for the constant
states	
state*(documentation)	Name of state, optional documentation using attribute
inputs	
input*(documentation)	Name of input, optional documentation using attribute
outputs	
output*(documentation)	Name of output, optional documentation using attribute
locations	
location+	
name	The name of the location
documentation?	Documenting text for this location
diffequation*(state)	A mathematical expression giving the differential equation for each state
outputmap*(output)	A mathematical expression involving the state variables that evaluates to an output value for this output
transitions?	
transition+	
name	The name of the transition
documentation?	Documenting text for this transition
domain	A logical expression involving the state and input variables
reset	Reset associated with this transition
destination	Name of destination location
statereset*(state)	Mathematical expression to reset the state indicated as attribute
inputevent*	The name of an event that is emitted when transitioning
outputevent*	The name of an event that can trigger this transition

The following gives a few additional explaining notes on the various elements from the table:

- **hints**: are optional symbol/value pairs that can provide meta information to the Sophy environment. Currently used for development and experimentation.
- **diffequation**: must be the corresponding state equation for the state indicated as attribute
- **outputmap**: must be the corresponding algebraic equation for the output indicated as attribute
- A transition becomes enabled when the **domain** evaluates to true
- **statereset**: is an algebraic equation of the state and input which calculates the new state value in the new location

A.5 Document Type Definition for a Hybrid System

The following is the DTD specification for a hybrid systems model:

```
<?xml version='1.0' encoding='utf-8' ?>

<!ELEMENT SophySystem (name, documentation?, hints?, constants?, states,
                       inputs, outputs, locations)>
```

```
<!-- META DATA *******************************************************
    -->
<!ELEMENT name (#PCDATA)>

<!ELEMENT documentation (#PCDATA)>

<!ELEMENT constants (constant+ )>
  <!ELEMENT constant (#PCDATA)>
      <!ATTLIST constant symbol CDATA #REQUIRED>

<!ELEMENT hints (hint+)>
  <!ELEMENT hint (#PCDATA)>
    <!ATTLIST hint hintname CDATA #REQUIRED>

<!-- IO and states ***************************************************
     -->
<!ELEMENT states (state*)>
  <!ELEMENT state (#PCDATA)>
      <!ATTLIST state documentation CDATA #IMPLIED>

<!ELEMENT inputs (input*)>
  <!ELEMENT input (#PCDATA)>
      <!ATTLIST input documentation CDATA #IMPLIED>

<!ELEMENT outputs (output*)>
  <!ELEMENT output (#PCDATA)>
      <!ATTLIST output documentation CDATA #IMPLIED>

 <!-- Locations and Dynamics*******************************************
      -->
<!ELEMENT locations (location+)>
  <!ELEMENT location  (name, documentation?, diffequation*, outputmap*,
                        transitions?)>
  <!ELEMENT diffequation (#PCDATA)>
    <!ATTLIST diffequation state CDATA #REQUIRED>
  <!ELEMENT outputmap (#PCDATA)>
    <!ATTLIST outputmap output CDATA #REQUIRED>
  <!ELEMENT transitions (transition+)>
    <!ELEMENT transition (name, documentation?, domain, reset,
        inputevent*,
              outputevent*)>
      <!ELEMENT domain (#PCDATA)>
      <!ELEMENT reset (destination, statereset*)>
        <!ELEMENT destination (#PCDATA)>
        <!ELEMENT statereset (#PCDATA)>
          <!ATTLIST statereset state CDATA #REQUIRED>
      <!ELEMENT inputevent (#PCDATA)>
      <!ELEMENT outputevent (#PCDATA)>
```

A.6 Example of Subsystem Specification

The following provides an example of a model described using the *SophySystem* specification.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE SophySystem SYSTEM "SophySystem.dtd">
<SophySystem>
    <name>RaibertsHopper</name>
    <documentation>
      This file implements a model of the one legged robot called
      "Raibert's Hopper". The model is documented in "Hybrid Systems
       - Lecture Notes in Computer Science 736", Springer, 1993
    </documentation>

    <constants>
       <constant symbol="G">9.82</constant>
       <constant symbol="Tp">0.35</constant>
       <constant symbol="L">0.5</constant>
       <constant symbol="GAMMA">0.1</constant>
       <constant symbol="TAU">40</constant>
       <constant symbol="ETA">8</constant>
    </constants>

    <!-- *****************************************************************
        -->
    <states>
       <state>Position</state>
       <state>Speed</state>
       <state documentation="Only used in the thrust location">Time</state
           >
       <state documentation="Only used in the decompression location">ETA2
           </state>
    </states>

    <inputs> </inputs>

    <outputs>
        <output>Position</output>
        <output>Speed</output>
    </outputs>

    <locations>
    <!-- ************************************************************* -->
        <location>
          <name>Flight</name>
          <documentation>
              In this location the hopper is in free flight
          </documentation>

          <diffequation state="Position"> Speed</diffequation>
```

```
    <diffequation state="Speed"> -G</diffequation>
    <diffequation state="Time"> 0</diffequation>
    <diffequation state="ETA2"> 0</diffequation>

    <outputmap output="Position"> Position </outputmap>
    <outputmap output="Speed"> Speed </outputmap>

    <transitions>
      <transition>
        <name> Ground Impact</name>
          <domain>Position &lt; L</domain>
          <reset>
            <destination> Compression </destination>
          </reset>
      </transition>
    </transitions>
  </location>

  <!-- ************************************************************
      -->
  <location>
    <name>Compression</name>
    <documentation>
      The foot now touches the ground and the gas is being
          compressed.
     </documentation>

    <diffequation state="Position"> Speed</diffequation>
    <diffequation state="Speed"> ETA/Position-GAMMA*Speed-G</
        diffequation>
    <diffequation state="Time"> 0</diffequation>
    <diffequation state="ETA2"> 0</diffequation>

    <outputmap output="Position"> Position </outputmap>
    <outputmap output="Speed"> Speed </outputmap>

    <transitions>
      <transition>
        <name> Deaccelerated</name>
        <domain>Speed &lt; 0</domain>
        <reset>
          <destination> Thrust </destination>
          <statereset state="Time"> 0 </statereset>
        </reset>
      </transition>
    </transitions>
  </location>

  <!-- ************************************************************
      -->
  <location>
    <name>Thrust</name>
    <documentation>
```

```
          The high pressure tank valve is opened and the hopper
          is propelled upwards
        </documentation>

        <diffequation state="Position"> Speed</diffequation>
        <diffequation state="Speed"> TAU−GAMMA∗Position−G</diffequation>
        <diffequation state="Time"> 1</diffequation>
        <diffequation state="ETA2"> 0</diffequation>

        <outputmap output="Position"> Position </outputmap>
        <outputmap output="Speed"> Speed </outputmap>

        <transitions>
          <transition>
            <name> Thrust timer expires</name>
              <domain>(Time &gt; Tp)||(Position &gt; L)</domain>
              <reset>
                <destination> Decompression </destination>
                <statereset state="ETA2"> Position∗TAU </statereset>
              </reset>
            </transition>
         </transitions>
      </location>

       <!−− ************************************************************
            −−>
       <location>
          <name>Decompression</name>
          <documentation>
          Thrust is over but the foot is not yet clear of the ground,
              thus the
          gas still provides a small upwards force
          </documentation>

          <diffequation state="Position"> Speed</diffequation>
          <diffequation state="Speed"> ETA2/Position−GAMMA∗Speed−G</
              diffequation>
          <diffequation state="Time"> 0</diffequation>

          <outputmap output="Position"> Position </outputmap>
          <outputmap output="Speed"> Speed </outputmap>

        <transitions>
          <transition>
            <name> Ground Cleared</name>
            <domain>Position &gt; FootHeight</domain>
            <reset>
              <destination> Flight </destination>
            </reset>
          </transition>
        </transitions>
      </location>
   </locations>
```

```
</SophySystem>
```

B Software Overview

This Appendix provides an overview of the software co-developed with this thesis

B.1 Obtaining the Software

The software is property of Aalborg Univeristy. If interested in using it for any purpose contact `dimon@es.aau.dk` or `jakob@es.aau.dk` to obtain the software and negotiate terms and conditions of use.

In addition the following software libraries must be installed in order to compile and run the software:
`http://sourceforge.net/projects/jep/` - Java Equation Parser (JEP)
`http://jmathtools.sourceforge.net/doku.php` - jmathPlot for plotting

B.2 Software Structure

The software repository consists of a number of Java-packages, the package/directory structure is as follows:

```
|-- TextDocs
|-- XMLFiles
|-- resources
|-- rpe
`-- devs
    |-- control
    |-- customfunctions
    |-- devsCore
    |-- discreteTimeIntegrators
    |-- equation
    |-- estimation
    |-- examples
```

```
    |   ‘-- thesis
    |-- hybrid
    |-- io
    |-- qss
    |-- qss2
    ‘-- tools
```

The following gives a very condensed description of the contents of each package:

`TextDocs`: contains a note on idead for improvement of the software.

`XMLDocs`: contains document type definitions and examples of hybrid systems specified as XML files, e.g. the model of Raibert's Hopper.

`ressources`: contains a number of comma separated data files which is used as input for some of the examples in the `devs.examples` package.

`rpe` contains some out-of-tree updates to the JEP library that is used to increase performance.

`devs.control`: contain all controller implementations, i.e. the optimisation based controller and the sliding mode controller classes.

`devs.customFunctions`: contains implementations of various mathematical functions that is used with JEP in the examples.

`devs.devsCore`: the implementation of the Discrete EVent Specification.

`devs.discreteTimeIntegrators`: contains an implementation for the forward Euler algorithm for time discrete integrations.

`devs.equation`: contains classes used to describe and evaluate equation sets for the various map implementations.

`devs.estimation`: contains the Extended Kalman Filter implementation.

`devs.example`: contains various examples testing/exploring functionality.

`devs.example.thesis`: contains code for seeting up all the examples that has been included in the thesis.

`devs.hybrid`: contains classes for simulating and executing hybrid systems both for QSS and QSS2 based models.

`devs.io`: various classes to facilitate input and output to/from the DEVS eviroment, e.g. matlab connections, plotting, file IO.

`devs.qss`: implements the QSS algorithm for quantised state systems - not discussed during the thesis.

`devs.qss2`: implements the QSS2 algorithm for quantised state systems, which

has been used extensively throught the thesis.

`devs.tools:` contains various tools; a `Connections` class to help automate setting up DEVS connections for complex models and the `XMLModelFactory` to build hybrid system models from XML specifications.

B.3 Getting Started

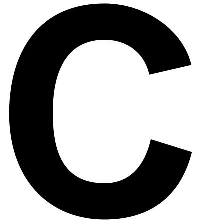
To get started with the code it is recommended to first take at look at some of the simple examples in the `devs.examples` package. For examples:

`Qss2Test`: sets up a simple 2nd order model in QSS2, simulates it using a standard simulation *runner*, and plots the results.

`CartOnPlane:` sets up a model of a cart driving on a plane and composes it with an optimizing controller, simulates the results and plots it.

`DeepSpaceProbeIAV2007:` sets up a model of a deep space probe, composes it with the QSS/EKF filter and runs the model with input from datafiles with results from a truth-model.

To develop new software use the example as templates and make sure to read the java-doc comments documenting the class interfaces.



Code Examples

This appendix contains an examples of declaring models/controllers/estimators using the DEVS/QSS framework. The objective of including this in the thesis is to demonstrate the declarative *manner in which a system is set up.*

C.1 Code Example for Deep Space Probe Case

```
package DevsSophy.Examples.thesis;

import DevsSophy.Qss.*;
import DevsSophy.Tools.*;
import DevsSophy.control.*;
import DevsSophy.customfunctions.*;
import DevsSophy.estimation.*;
import DevsSophy.io.*;
import Jama.Matrix;
import DevsSophy.DevsCore.*;

/** This example sets up a model for the Deep Space Probe case study of
 * chapter 7 and composes it with an Extended Kalman Filter and a
 * Sliding mode controller which includes a dynamic uncertainty bound
 * estimator. The complete model is set up to run with Matlab through
 * socket communication
 * @author Lars Alminde (alminde@es.aau.dk)
 */
public class DSPEvalSMC {
  //Main method with all the action
  public static void main(String[] args){

    //Declare the Devs-Context and loading constants and custom function
    //-----------------------------------------------------------------------
    DevsContext con= new DevsContext(0,10e-9,true);
    con.addConstant("J1=30, J2=30, J3=50, Xdsp=1e6, Ydsp=0, Zdsp=0");
    con.addConstant("Xsun=0, Ysun=0, Zsun=0, Xstar=0, Ystar=1e6, Zstar=5e6");
    con.addFunction("rotate321",new Rotate321()); //3-2-1 Euler rotation
    con.addFunction("DspGyro",new DspGyroCoupling()); //GyroDynamics

    //Declare a Sample-Hold filter for EKF
    //-----------------------------------------------------------------------
    DevsAtomic filter=new DSPEvalSampleFilter(con, "Filter",true);

    //Declare the dynamics and kinematics in as an equationsset and
    //set up at corresponding Qss2Map
    //-----------------------------------------------------------------------
    RpeEquationSet eqSet = new RpeEquationSet(
      new String[]{"u1","u2","u3","z1","z2","z3","z4","z5","z6"});
    eqSet.addContext(con);
    eqSet.addEquation("" +
```

```
    "(1/cos(z2))*(cos(z2)*z4+sin(z1)*sin(z2)*z5+cos(z1)*sin(z2)*z6)");
eqSet.addEquation("(1/cos(z2))*(cos(z1)*cos(z2)*z5-sin(z1)*cos(z2)*z6)");
eqSet.addEquation("(1/cos(z2))*(sin(z1)*z5+cos(z1)*z6)");
eqSet.addEquation("DspGyro(1,z4,z5,z6)+0.5*(0.0334*u1+0.0006*u2+0.0007*u3)");
eqSet.addEquation("DspGyro(2,z4,z5,z6)+0.5*(0.0006*u1+0.0334*u2+0.0010*u3)");
eqSet.addEquation("DspGyro(3,z4,z5,z6)+0.5*(0.0007*u1+0.0010*u2+0.0200*u3)");
eqSet.lock();

//initial values of input
double[] op=new double[]{0,0,0,0.1,-1,0,0,-0.03,0.15};

//construct the map and differentiate wrt. init. value
Qss2Static t = new Qss2Static(con,"Static",eqSet,2);
t.differentiate(op,0.01);

//State declarations
//-----------------------------------------------------------------------
Qss2ResetIntegrator int1 = new Qss2ResetIntegrator(con,"int1", op[3], 1e-5);
Qss2ResetIntegrator int2 = new Qss2ResetIntegrator(con,"int2", op[4], 1e-5);
Qss2ResetIntegrator int3 = new Qss2ResetIntegrator(con,"int3", op[5], 1e-5);
Qss2ResetIntegrator int4 = new Qss2ResetIntegrator(con,"int4", op[6], 1e-6);
Qss2ResetIntegrator int5 = new Qss2ResetIntegrator(con,"int5", op[7], 1e-6);
Qss2ResetIntegrator int6 = new Qss2ResetIntegrator(con,"int6", op[8], 1e-6);

//Declare extended Kalman filter
//-----------------------------------------------------------------------
//Declare measurement equations
RpeEquationSet measurements = new RpeEquationSet(
  new String[]{"e1","e2","e3","p","q","r"});
measurements.addContext(con);
measurements.addEquation(
  "rotate321(1,e1,e2,e3,(Xsun-Xdsp),(Ysun-Ydsp),(Zsun-Zdsp))");
measurements.addEquation(
  "rotate321(2,e1,e2,e3,(Xsun-Xdsp),(Ysun-Ydsp),(Zsun-Zdsp))");
measurements.addEquation(
  "rotate321(3,e1,e2,e3,(Xsun-Xdsp),(Ysun-Ydsp),(Zsun-Zdsp))");
measurements.addEquation(
  "rotate321(1,e1,e2,e3,(Xstar-Xdsp),(Ystar-Ydsp),(Zstar-Zdsp))");
measurements.addEquation(
  "rotate321(2,e1,e2,e3,(Xstar-Xdsp),(Ystar-Ydsp),(Zstar-Zdsp))");
measurements.addEquation(
  "rotate321(3,e1,e2,e3,(Xstar-Xdsp),(Ystar-Ydsp),(Zstar-Zdsp))");
measurements.lock();

//Construct and differentiate measurement map
Qss2Static measureMap=new Qss2Static(con,"Measurements",measurements,0);
measureMap.differentiate(new double[]{op[3],op[4],op[5],op[6],op[7],op[8]},0.1);

//Initial paramters for the EKF
Matrix P=new Matrix(new double[][]{{0.1,0,0,0,0,0},{0,0.1,0,0,0,0},
  {0,0,0.1,0,0,0},{0,0,0,1e-4,0,0},{0,0,0,0,1e-4,0},{0,0,0,0,0,1e-4}});
Matrix Q=new Matrix(new double[][]{{0,0,0,0,0,0},{0,0,0,0,0,0},
  {0,0,0,0,0,0},{0,0,0,1,0,0},{0,0,0,0,1,0},{0,0,0,0,0,1}}).times(1e-8);

//Construct the filter and register states
EKF ekf=new EKF(con, "ekf",0.02 ,2, P, Q, measureMap,false);
ekf.registerState(new Qss2ResetIntegrator[]{int1,int2,int3,int4,int5,int6});

//Register Sun-sensor
Matrix Rss2=new Matrix(new double[][]{{3e-4,0,0},{0,3e-4,0},{0,0,3e-4}});
ekf.registerMeasurement(new int[]{0,1,2},Rss2);

//Register Star-Sensor
Matrix Rss=new Matrix(new double[][]{{3e-6,0,0},{0,3e-6,0},{0,0,3e-6}});
ekf.registerMeasurement(new int[]{3,4,5},Rss);

//Class for sampling outputs at 100Hz (only viasualisation after simulation)
//-----------------------------------------------------------------------
Qss2SampledOutput sample=new Qss2SampledOutput(con,"Sample",6,0.01,true,true);

//Declare Sliding Mode Controller
//-----------------------------------------------------------------------
```

```
//Declare performance Metric
RpeEquationSet per = new RpeEquationSet(new String[]{"z1","z2","z3"});
per.addContext(con);
per.addEquation("0.5*(z1-0.1)^2+0.5*(z2+0.5)^2+0.5*(z3-0.7)^2");
per.lock();

//set up the 'negativegradieant' function
NegativeGradient controlStage1=new NegativeGradient(con, "nGrad", per);
controlStage1.differentiate(new double[]{0.5, -0.1, 1},0.000001);

//set up the 'SlidingMode' class
SlidingMode controlStage2=new SlidingMode(con, "SlidingMode",3);

//set up the 'SlidingController'
SlidingController controlStage3=new SlidingController(con, "SlidingControl", 3, 3, new double
    []{0.01, 0.01, 0.01}, t);
controlStage3.setFullMap();
controlStage3.setSaturation(new double[]{0.8, 0.8, 0.8});
//controlStage3.setProportionalGain(new double[]{0.06,0.06,0.06});
//controlStage3.setSquareGain(new double[]{0.475,0.475,0.475});
controlStage3.setBoundaryLayer(new TanhLayer(1));

//Set up the custon uncertainty class - hardcoded with relevant equations
Uncertainty unc=new Uncertainty(con,"Uncertainty");

//Declare the DEVS coordinator and add all model components to it
//-------------------------------------------------------------------------
DevsCoordinator coord=new DevsCoordinator(con,"Coord",9,9,15);
coord.addAtomic(new DevsAtomic[]{filter, t, int1, int2, int3, int4, int5});
coord.addAtomic(new DevsAtomic[]{int6, ekf, measureMap, sample,unc});
coord.addAtomic(new DevsAtomic[]{controlStage1,controlStage2,controlStage3});

//Set up all the runtime connections between the DEVS Components
//-------------------------------------------------------------------------
//External inputs - from matlab
coord.addInput(1,filter,1);
coord.addInput(2,filter,2);
coord.addInput(3,filter,3);
coord.addInput(4,filter,4);
coord.addInput(5,filter,5);
coord.addInput(6,filter,6);
coord.addInput(7,filter,7);
coord.addInput(8,filter,8);
coord.addInput(9,filter,9);

//Maps to integrators and visa verca
Connections.mapToInt(coord,t,new DevsAtomic[]{int1,int2,int3,int4,int5,int6},0);
Connections.intToMap(coord,new DevsAtomic[]{int1,int2,int3,int4,int5,int6},t,3);
Connections.intToMap(coord,new DevsAtomic[]{int1,int2,int3,int4,int5,int6},measureMap,0);
Connections.intToMap(coord,new DevsAtomic[]{int1,int2,int3,int4,int5,int6},sample,0);

//Controller connections
Connections.intToMap(coord,new DevsAtomic[]{int1,int2,int3},controlStage1,0);
Connections.intToMap(coord,new DevsAtomic[]{int4,int5,int6},controlStage2,0);
Connections.intToMap(coord,new DevsAtomic[]{int1,int2,int3},controlStage1,0);

coord.addConnection(controlStage1,1,controlStage2,4);
coord.addConnection(controlStage1,2,controlStage2,5);
coord.addConnection(controlStage2,1,controlStage3,1);
coord.addConnection(controlStage2,2,controlStage3,2);
coord.addConnection(unc,1,controlStage3,3);
coord.addConnection(filter,3,unc,1);

coord.addConnection(controlStage3,1,t,1);
coord.addConnection(controlStage3,2,t,2);
coord.addConnection(controlStage3,3,t,3);

//EKF connections
coord.addConnection(t,7,ekf,1);
coord.terminateConnection(t,8);
coord.addConnection(filter,1,ekf,2);
coord.addConnection(filter,2,ekf,3);
```

```
    coord.addConnection(ekf,1,int1,2);
    coord.addConnection(ekf,2,int2,2);
    coord.addConnection(ekf,3,int3,2);
    coord.addConnection(ekf,4,int4,2);
    coord.addConnection(ekf,5,int5,2);
    coord.addConnection(ekf,6,int6,2);

    coord.terminateConnection(measureMap,1);
    coord.terminateConnection(measureMap,2);
    coord.terminateConnection(measureMap,3);
    coord.terminateConnection(measureMap,4);
    coord.terminateConnection(measureMap,5);
    coord.terminateConnection(measureMap,6);

    //Outputs to Matlab
    coord.addOutput(controlStage3,1,1);
    coord.addOutput(controlStage3,2,2);
    coord.addOutput(controlStage3,3,3);
    coord.terminateConnection(controlStage3,4);

    coord.addOutput(sample,1,4);
    coord.addOutput(sample,2,5);
    coord.addOutput(sample,3,6);
    coord.addOutput(sample,4,7);
    coord.addOutput(sample,5,8);
    coord.addOutput(sample,6,9);

    //Make runner and do simulation
    //-----------------------------------------------------------------
    //Construct Matlab Adaptor for localhost port 8189
    RendevouzAdaptor ada=new MatlabAdaptor(8189);

    //Set up rendevouzrunner for 400s of execution
    DevsRendevouzRunner runner=new DevsRendevouzRunner(coord,ada,399);

    //Start the execution (will await Matlab)
    runner.run(DevsRunner.NOREALTIME, 0.05);
  }
}
```

Bibliography

[Alamir, 2006] Alamir, M. (2006). *Stabilization of Nonlinear Systems Using Receding-horizon Control Scheme*. Springer.

[Alminde et al., 2006a] Alminde, L., Bendtsen, J. D., and Stoustrup, J. (2006a). *A Quantized State Approach to On-line Simulation for Spacecraft Autonomy*. AIAA. In 2006 Modeling and Simulation Technologies Conference Proceedings. American Institute of Aeronautics and Astronautics, Keystone, Colorado, August 2006.

[Alminde et al., 2007a] Alminde, L., Bendtsen, J. D., and Stoustrup, J. (2007a). *A Quantised State Systems Approach for Jacobian Free Extended Kalman Filtering*. IFAC. Submitted for IAV2007.

[Alminde et al., 2007b] Alminde, L., Bendtsen, J. D., Stoustrup, J., and Pettersen, K. Y. (2007b). *Objective Directed Control using Local Minimisation for an Autonomous Underwater Vehicle*. IFAC. In proceedings of IAV2007, 3-5. September 2007, Toulouse, France.

[Alminde et al., 2006b] Alminde, L., Laursen, K. K., and Bendtsen, J. D. (2006b). *A Reusable Software Architecture for Small Satellite AOCS Systems*. European Space Agency. In proceedings of Small Satellites Systems and Services 2006, Chia Laguna, Italy, 25.-29. September 2006.

[Alminde et al., 2007c] Alminde, L., Laursen, K. K., and Bendtsen, J. D. (2007c). *Sophy: A tool for Declarative Control*. n/a. Submitted for review for an international journal on control architectures.

[Ascher and Petzold, 1998] Ascher, U. M. and Petzold, L. R. (1998). *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics.

[Back et al., 1993] Back, A., Guckenheimer, J., and Myers, M. (1993). *A Dynamical Simulation Facility for Hybrid Systems*. Springer-Verlag. In Lecture Notes in Computer Science no. 736 p. 255-267.

[Bandyopadhyay and Sivaramakrishnan, 2006] Bandyopadhyay, B. and Sivaramakrishnan, J. (2006). *Discrete-Time Sliding Mode Control*. Springer.

[Barton and Lee, 2002] Barton, P. I. and Lee, C. K. (2002). *Modeling, Simulation, Sensitivity Analysis, and Optimization of Hybrid Systems*. ACM. ACM Transactions on Modelling and Computer Simulation, Vol 12, No. 4, October 2002, pages 256-289.

[Bemporad and Morari, 1999] Bemporad, A. and Morari, M. (1999). *Control of Systems Integrating Logic, Dynamics, and Constraints*. Elsevier. Journal of Automatica, no. 35, p. 407-427.

[Bernard et al., 1999] Bernard, D., Doyle, R., Riedel, E., Rouquette, N., Wyatt, J., Lowry, M., and Nayak, P. (1999). *Autonomoy and Software Technology on NASA's Deep Space One*. IEEE. IEEE Intelligent Systems Journal, May/June 1999.

[Bernard et al., 2000] Bernard, D. E., Gamble, E. B., Jr., Rouquette, N. F., Smith, B., Tung, Y.-W., Muscettola, N., Dorias, G. A., Kanefsky, B., Kurien, J., Millar, W., Nayak, P., Rajan, K., and Taylor, W. (2000). *Remote Agent Experiment DS1 Technology Validation Report*. NASA.

[Bhanderi, 2005] Bhanderi, D. D. V. (2005). *Spacecraft Attitude Determination with Earth Albedo Corrected Sun Sensor Measurement*. Aalborg University. PhD thesis.

[Boyd and Vandenberghe, 2004] Boyd, S. and Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press.

[Branicky, 1994] Branicky, M. S. (1994). *Stability of switched and hybrid systems*. IEEE. In proceedings of the 33rd IEEE Conference on Decision and Control, Florida, USA, December.

[Branicky et al., 1998] Branicky, M. S., Borkar, V. S., and Mitter, S. K. (1998). *A Unified Framework for Hybrid Control: Model and Optimal Control Theory*. IEEE. IEEE Transactions on Automatic Control, Vol 43, NO. 1, January 1998.

[Callear, 2003] Callear, D. (2003). *Prolog programming for students : with expert systems and artifical intelligence topics*. Thomson Learning.

[Carlo et al., 2000] Carlo, R. A., Branicky, M. S., Pettersson, S., and Lennartson, B. (2000). *Perspectives and Results on the Stabillizability of Hybrid Systems*. IEEE. In proceedings of the IEEE Vol. 88, No. 7, July 2000.

[Cassandras and Lafortune, 1999] Cassandras, C. G. and Lafortune, S. (1999). *Introduction to Discrete Event Systems*. Kluwer Academic Publishers.

[Consortium, 2006] Consortium, W. W. W. (2006). *XML Specification*. WWWC.

http://www.w3.org/XML/.

[Dvorak et al., 2004] Dvorak, D., Bollela, G., Canham, T., Carson, V., Champlin, V., Giovannoni, B., Indictor, M., Meyer, K., Murray, A., and Reinholtz, K. (2004). *Project Golden Gate: Towards Real-Time Java in Space Missions*. IEEE. Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04).

[Dvorak et al., 2000] Dvorak, D., Rasmussen, R., Reeves, G., and Sacks, A. (2000). *Software Architecture Themes in JPL's Mission Data System*. IEEE. In proceedings of IEEE Aerospace Conference, March 2000, p. 259-267.

[Fortescue et al., 2003] Fortescue, P., Stark, J., and Swinerd, G. (2003). *Spacecraft Systems Engineering*. Willey, 3 edition.

[Fossen, 2002a] Fossen, T. I. (2002a). *Marine Control Systems - Guidance, Navigation and Control of Ships, Rigs and Underwater Vehicles*. Marine Cybernetics.

[Fossen, 2002b] Fossen, T. I. (2002b). *Marine GNC Toolbox for matlab*. Marine Cybernetics. available at: www.marinecybernetics.no.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994). *Design Patterns: Elements of Reuseable Object-Oriented Software*. Addison-Wesley.

[Gene F. Franklin and Emami-Naeini, 1994] Gene F. Franklin, J. D. P. and Emami-Naeini, A. (1994). *Feedback Control of Dynamic Systems*. Addison Wesley.

[Grewal and Andrews, 1993] Grewal, M. S. and Andrews, A. P. (1993). *Kalman Filtering Theory and Practice*. Prentice Hall.

[Habets et al., 2006] Habets, L., Collins, P., and van Schuppen, J. (2006). Reachability and control synthesis for piecewise-affine hybrid systemson simplices. *IEEE Transactions on Automatic Control*, 51:938–948.

[Healey and Lienard, 1993] Healey, A. J. and Lienard, D. (1993). *Multivariable Sliding Mode Control for Autonomous Diving and Steering of Unmanned Underwater Vehicles*. IEEE. Journal of Oceanic Engineering, Vol. 18, No. 3, July 1993.

[Heemels et al., 2006] Heemels, W. P. M. H., Siahaan, H. B., Juloski, A. L., and Weiland, S. (2006). *Control of Quantised Linear Systems: an l_1-optimal approach*. IEEE. In proceedings of the 2006 American Control Conference,

Denver, USA, June 4-6, 2003.

[Henzinger, 1996] Henzinger, T. A. (1996). *The Theory of Hybrid Automata*. IEEE. Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on , 27-30 July 1996 Pages:278 - 292.

[Henzinger et al., 2003] Henzinger, T. A., Horowitz, B., and Kirsch, C. M. (2003). *Giotto - A Time-Triggered Language for Embedding Programming*. IEEE. Proceedings of the IEEE, vol. 91, no. 1, January 2003.

[Horowitz et al., 2003] Horowitz, B., Liebman, J., Ma, C., Koo, T. J., Sangiovanni-Vincentelli, A., and Sastry, S. S. (2003). *Platform-Based Embedded Software Design and System Integration for Autonomous Systems*. IEEE. Proceedings of the IEEE, vol. 91, no. 1, January 2003.

[Johnson and Brown, 1998] Johnson, S. A. and Brown, G. M. (1998). *An Overview of the Fault Protection Design for the Attitude Control Subsystem of the Cassini Spacecraft*. IEEE. In proceedings: American Control Conference 1998.

[Khalil, 2000] Khalil, H. K. (2000). *Non-linear Systems*. Prentice Hall, 3 edition.

[Kofman, 2002] Kofman, E. (2002). *A Second Order Approximation for DEVS Simulation of Continous Systems*. SIAM. Journal of Simulation, issue 78, p. 76-89.

[Kofman, 2003] Kofman, E. (2003). *Discrete Event Based Simulation and Control of Continuous Systems*. Facultad de Ciencias Exactas, Ingeniería y Agrimensura Universidad Nacional de Rosario. PhD-thesis.

[Kofman, 2004] Kofman, E. (2004). *Discrete Event Simulation of Hybrid Systems*. SIAM. SIAM Journal on Scientific Computing. 25(5). pp 1771-1797.

[Kofman et al., 2001] Kofman, E., Lee, J. S., and Zeigler, B. P. (2001). *DEVS Representation of Differential Equation Systems: Review of Recent Advances*. IEEE. Proceedings of ESS'01 pp. 591-595.

[Koo et al., 2005] Koo, T., Dubey, A., Wu, X., and Su, H. (2005). *Computation Platform for Automatic Analysis of Embedded Software Systems Using Model Based Approach*. Springer. Lecture Notes in Computer Science - Volume 3707/2005.

[Kreyzig, 1999] Kreyzig, E. (1999). *Advanced Engineering Mathematics*. John Wiley and Sons, 8 edition.

[Laursen et al., 2005] Laursen, K. K., Pedersen, M. F., Bendtsen, J. D., and Al-

minde, L. (2005). *The SOPHY Framework: Simulation, Observation and Planning in Hybrid Systems*. IEEE. Fifth International Conference on Hybrid Intelligent Systems (HIS05), p 457-462.

[Lieu et al., 1999] Lieu, J., Liu, X., Koo, T.-K. J., Sinopoli, B., Sastry, S., and Lee, E. A. (1999). *A Hierarchical Hybrid System Model and Its Simulation*. IEEE. Proceedings of the 38th Conference on Decision and Control, Phoenix, December 1999, page 3508-3513.

[Lunze, 1994] Lunze, J. (1994). *Qualitative Modelling of Linear Dynamical Systems with Quantized State Measurements*. Pergamon. Automatica, Vol. 30, no. 3, pp. 417-431, 1994.

[Maciejowski, 2002] Maciejowski, J. M. (2002). *Predictive Control with Constraints*. Prentice Hall.

[Nutaro, 2005] Nutaro, J. (2005). *Constructing Multi-point Discrete Event Integration Schemes*. INFORMS. Proceedings of the 5th Winter Simulation Conference (WSC), 4-7/12 2005, Orlando, USA.

[Oort et al., 2006] Oort, E., Chu, Q., and Mulder, J. (2006). *Robust Model Predictive Control of a Feedback Linearized F-16/MATV Aircraft Model*. AIAA. In 2006 Guidance, Navigation, and Control Proceedings. American Institute of Aeronautics and Astronautics, Keystone, Colorado, August 2006.

[Pettersson and Lennartson, 1997] Pettersson, S. and Lennartson, B. (1997). *Controller Design of Hybrid Systems*. Springer. Lecture notes in Computer Science 1201, pp. 240-254.

[Philips et al., 2003] Philips, P. P. H. H., Heemels, W. P. M. H., Preisig, H. A., and van den Bosch, P. P. J. (2003). *Control of Quantised Systems Based on Discrete Events*. Taylor and Francis. International Journal of Control, Vol. 76, No. 3, pp. 277-294, 2003.

[Quine, 2006] Quine, B. M. (2006). *A Derivative-free Implementation of the Extended Kalman Filter*. Elsevier. Automatica volume 42, p. 1927-1934, September 2006.

[Rodrigues and How, 2003] Rodrigues, L. and How, J. P. (2003). *Observer-based Control of Piecewise-affine Systems*. IEEE. Internation Jounal of Control 2003, VOL. 76, NO. 5.

[Sandee and Heemels, 2006] Sandee, J. H. and Heemels, W. P. M. H. (2006). *Practical Stability of Pertubed Event-Driven Controlled Linear Systems*. IEEE. In proceedings of the 2006 American Control Conference, Minnesota, USA,

June 24-16, 2006.

[Sandee et al., 2005] Sandee, J. H., Heemels, W. P. M. H., and Bosch, P. P. J. (2005). *Event-Driven Control as an Opportunity in the Multidisciplinary Development of Embedded Controllers*. AAC. In proceedings of the 2005 American Control Conference, June 8-10, Portland, USA.

[Sandee et al., 2006] Sandee, J. H., Visser, P. M., and Heemels, W. P. M. H. (2006). *Analysis and Experimental Validation of Processor Load for Event-Driven Controllers*. IEEE. In proceedings of the 2006 IEEE International Conference on Control Applications, Munich, Germany, October 4-6, 2006.

[Sarjoughian and Cellier, 2001] Sarjoughian, H. S. and Cellier, F. E. (2001). *Discrete Event Modeling and Simulation Technologies: A Tapestry of Systems and AI-Based Theories and Methodologies*. Springer Verlag.

[Schei, 1997] Schei, T. S. (1997). *A Finite-Difference Method for Linearization in Nonlinear Estimation Algorithms*. Pergamon. Automatica volume 33, No. 11 p. 2053-2058, 1997.

[Seibel, 2005] Seibel, P. (2005). *Practical Common Lisp*. apress.

[Stern and Spencer, 2003] Stern, A. and Spencer, J. (2003). *New Horizons: The First Reconnaissance Mission to Bodies in the Kuiper Belt*. Springer. Journal of Earth, Moon, and Planets, Volume 99, no. 1-4, P. 477-482.

[Sun_Microsystems, 2007a] Sun_Microsystems (2007a). *Enterprise Java Beans Product Description*. Sun Microsystems. `http://java.sun.com/products/ejb/`.

[Sun_Microsystems, 2007b] Sun_Microsystems (2007b). *Enterprise Java Beans Product Description*. Sun Microsystems. `http://java.sun.com/products/javabeans/`.

[Taylor and Kebede, 1997] Taylor, J. H. and Kebede, D. (1997). *A Rigorous Hybrid Systems Simulation of an Electro-mechanical Pointing System with Discrete-time Control*. AAAC. Proceedings of the American Control Conference, Alberquerque, June 1997, page: 2786-2789.

[Utkin, 1977] Utkin, V. I. (1977). *Variable Structure Systems with Sliding Modes*. IEEE. IEEE Transactions on Automatic Control, Vol. AC-22, No. 2, April 1977.

[Vakakis and Burdick, 1990] Vakakis, A. F. and Burdick, J. W. (1990). *Chaotic Motion in The Dynamics of a Hopping Robot*. IEEE. In proceedings of the

1990 IEEE conference on Robotics and Control, p. 1464-1469, Cincinnati, OH, USA.

[Wan and Merwe, 2000] Wan, E. and Merwe, R. V. D. (2000). *The Unscented Kalman Filter for Nonlinear Estimation*. IEEE. Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000, 1-4 Oct. 2000.

[Wertz, 1978] Wertz, J. R. (1978). *Spacecraft Attitude Determination and Control*. Kluwer Academic Publishers.

[Wiesniewski, 1998] Wiesniewski, R. (1998). *Sliding Mode Attitude Control for Magnetic Actuated Satellite*. IFAC. In Proceedings of the 14th IFAC Symposium on Automatic Control in Aerospace.

[Williams et al., 2003] Williams, B. C., Ingham, M. D., Chung, S. H., and Elliot, P. H. (2003). Model-based programming of intelligent embedded system and robotic space explorers. In *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software, vol. 9, no. 1, pp. 212-237.*

[Williams and Nayak, 1999] Williams, B. C. and Nayak, P. P. (1999). A model-based approach to reactive self-configuring systems. In Minker, J., editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*, College Park, Maryland. Computer Science Department, University of Maryland.

[Zarchan and Musoff, 2000] Zarchan, P. and Musoff, H. (2000). *Fundementals of Kalman Filtering - A Practical Approach*. American Institute of Aeronautics and Astronautics.

[Zeigler, 1976] Zeigler, B. P. (1976). *Theory of Modelling and Simulation*. John Wiley and Sons. 1st edition.

[Zeigler et al., 2000] Zeigler, B. P., Praehofer, H., and Kim, T. G. (2000). *Theory of Modelling and Simulation*. John Wiley and Sons. 2nd edition.

[Årzén, 1999] Årzén, K.-E. (1999). *A Simple Event-Based PID Controller*. 1999. In Proceedings of the IFAC World Congress 1999.