Stellingen behorende bij het proefschrift:

Interfacing Control and Software Engineering: a formal approach

24 juni 1997

1. De hybride automaat specificatie die ontstaat uit de vertaling van een Hybrid-Astral specificatie is non-zeno.

2. Het is erg belangrijk om zich bij de keuze voor een hybride specificatietaal voor software requirements af te vragen of de voordelen die volgen uit de toegenomen expressiviteit wel opwegen tegen de naden len die veroorzaakt worden door de hogere complexiteit van de resulterende specificaties.

3. Een zinvolle uitbreiding van Ada95 programmeertaal betreft een gewijzigde vorm van het entry-call select-statement. Zo'n select-statement zou calls naar meerdere entries moeten toelaten zoals wordt geïllustreerd in het volgende voorbeeld:

```plaintext
select
  <call entry1>
or
  <call entry2>
or...
or
  <call entryN>
end select
```

4. De schedule- en environment-eigenschappen die in de Astral specificatie van de packetmaker [PKM94] worden gespecificeerd leggen zodanige beperkingen op aan het gemodelleerde gedrag dat niet meer van de specificatie van een realistisch systeem gesproken kan worden.


5. In veel gevallen zal de grootte van een Hybrid-Astral specificatie afnemen wanneer de taal mogelijkheden biedt voor de definitie van een generiek process type. Zo zou een generiek sensor process dan worden gedefinieerd als volgt:

```plaintext
GLOBAL SPECIFICATION Example

PROCESSSES
  c1 : SomeProcessType;
  c2 : SomeProcessType;
  sensor_for_c1 : GenericSensor(c1,x,y1);
  sensor_for_c2 : GenericSensor(c2,x,y2);

END GLOBAL SPECIFICATION

GENERIC
  monitoredprocess : SomeProcessType;
  monitoredvar : REAL;
  sensoroutputvar : REAL;

SPECIFICATION GenericSensor

END GenericSensor;
```

Uitbreidingen zoals deze zijn toegestaan zolang het aantal process instanties statisch bepaald blijft.
6. De overwinning van schaakcomputer 'Deep Blue' op Kasparov maakt het schaakspel een stuk minder uitdagend.

7. Het frequent gebruik van mobiele telefoons in openbare gelegenheden zou er op kunnen wijzen dat er in Nederland nog steeds sprake is van een cellentekort.

8. De hoeveelheid vuil die dagelijks door het winkelend publiek op straat wordt achterge- laten zou belangrijk verminderen wanneer het principe 'de vervuiler betaalt direct' toegepast zou worden.

9. Het optimisme over de invoering van de Euro en de Europese Monetaire Unie kan het best worden gekarakteriseerd als grenzeloos.

10. Met de ontwikkeling van de klapschaats heeft de wetenschap in de schaatssport toegeslagen.

11. De slogan van de nieuwste reclamecampagne van de Nederlandse Spoorwegen (NS): *Ieder z’n trein* geeft aan dat de NS slechts in staat is tot het vervoer van reizigers buiten de spits.

12. Wie het Pieterpad in één dag wil lopen moet heel wat in zijn mars hebben.
Interfacing Control and Software Engineering: a formal approach
Interfacing Control and Software Engineering: a formal approach

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Delft, op gezag van de Rector Magnificus Prof.dr.ir. J. Blaauwendaal, in het openbaar te verdedigen ten overstaan van een commissie, door het College van Dekanen aangewezen, op dinsdag 24 juni 1997 te 16.00 uur

door

Klaas BRINK

doctorandus in de informatica en H.B.O.-ingenieur in de technische informatica geboren te Hardenberg
Dit proefschrift is goedgekeurd door de promotor:  
Prof.dr.ir. J. van Katwijk

Toegevoegd promotor:  
Dr.ir. W.J. Toetenel

Samenstelling promotiecommissie:  
Rector Magnificus, voorzitter  
Prof.dr.ir. J. van Katwijk, Technische Universiteit Delft, promotor  
Dr.ir. W.J. Toetenel, Technische Universiteit Delft, toegevoegd promotor  
Prof.drs. M. Boasson, Universiteit van Amsterdam  
Prof.dr.ir. H.J. Sips, Technische Universiteit Delft  
Prof.dr. A.W. Strohmeier, Swiss Federal Institute of Technology  
Prof.ir. H.B. Verbruggen, Technische Universiteit Delft  
Prof.dr. J. Zalewski, University of Central Florida

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Brink, K.

Interfacing Control and Software Engineering: a formal approach / K. Brink -  
Delft: Delft University Press. - Illustrations.  
Thesis Delft University of Technology. - With ref. - With summary in Dutch.  
ISBN 90-407-1456-8  
NUGI 851  
Subject Headings: software development / real-time software /  
control engineering / formal specification

Copyright ©1997 by K. Brink

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without permission from the publisher: Delft University Press, Mekelweg 4, 2628 CD Delft, The Netherlands.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preface</strong></td>
<td>v</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>vii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 General Discussion</td>
<td>1</td>
</tr>
<tr>
<td>1.2 The Development Process</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 Embedded Control Systems</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2 Requirements Specification of Real-Time Software</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Research Topic</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Contents of this Thesis</td>
<td>7</td>
</tr>
<tr>
<td><strong>2 Control Engineering: An Introduction</strong></td>
<td>9</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Block Diagram Models</td>
<td>10</td>
</tr>
<tr>
<td>2.2.1 Global Structure</td>
<td>10</td>
</tr>
<tr>
<td>2.2.2 An Example: The Inverted Pendulum</td>
<td>12</td>
</tr>
<tr>
<td>2.2.3 Linear, Continuous-Time Systems</td>
<td>13</td>
</tr>
<tr>
<td>2.2.4 Linear, Discrete-Time Systems</td>
<td>15</td>
</tr>
<tr>
<td>2.2.5 Linear, Discrete-Continuous Time Systems</td>
<td>16</td>
</tr>
<tr>
<td>2.2.6 Non-Linear Systems</td>
<td>17</td>
</tr>
<tr>
<td>2.2.7 System Simulation Model</td>
<td>18</td>
</tr>
<tr>
<td>2.2.8 Control Software Model</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Controller Development Process</td>
<td>20</td>
</tr>
<tr>
<td>2.3.1 Phase 1: Construction of System Model</td>
<td>21</td>
</tr>
<tr>
<td>2.3.2 Transition to Phase 2</td>
<td>22</td>
</tr>
<tr>
<td>2.4 Evaluation</td>
<td>23</td>
</tr>
<tr>
<td><strong>3 Requirements Specification in Astral</strong></td>
<td>27</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>27</td>
</tr>
<tr>
<td>3.2 A Model for Real-Time Software Development</td>
<td>28</td>
</tr>
</tbody>
</table>
3.3 Formal Methods .............................................. 31
3.4 The Astral Specification Language ......................... 34
  3.4.1 Introduction ............................................. 34
  3.4.2 Language Overview ..................................... 34
  3.4.3 Example: Generalized Railroad Crossing (GRC) .... 37
3.5 Interfacing Controller and Software Development ........... 40
3.6 Evaluation .................................................. 41

4 The H-Astral Specification Language .......................... 45
  4.1 Introduction ................................................ 45
  4.2 A Computational Model for Hybrid Systems ............... 47
  4.3 Hybrid Automata ............................................ 50
  4.4 The Specification Language Hybrid-Astral ................. 57
    4.4.1 Motivation and Design Goals ......................... 57
    4.4.2 Hybrid-Astral ........................................ 58
    4.4.3 Semantics of Hybrid-Astral ......................... 60
  4.5 Verification with HyTech .................................. 67

5 Case Study: A Robot Control System .......................... 71
  5.1 Introduction ................................................ 71
  5.2 The Robot Control System ................................ 72
    5.2.1 Introduction .......................................... 72
    5.2.2 The Robot ............................................. 72
    5.2.3 Control Algorithms ................................... 76
    5.2.4 Block Diagram Representation ........................ 79
  5.3 Reformulation in H-ASTRAL ................................ 80
  5.4 The H-Astral Specification ................................. 83
    5.4.1 Architecture of the Specification ..................... 83
    5.4.2 Level 0 Processes .................................... 85
    5.4.3 Level 1 Processes .................................... 86
    5.4.4 Level 2 Processes .................................... 86
    5.4.5 Task Management ...................................... 90
  5.5 Simulation ................................................ 90
  5.6 Evaluation ............................................... 92

6 Prototyping Astral Specifications ............................ 95
  6.1 Introduction ............................................... 95
  6.2 The Ada95 Programming Language ......................... 96
  6.3 Mapping Astral Constructs onto Ada95 ...................... 98
    6.3.1 The Astral to Ada95 Mapping ......................... 98
    6.3.2 Prototyping Experiences ............................... 103
CONTENTS

6.3.3 Discussion .......................................................... 106
6.4 Timing Analysis ......................................................... 107
  6.4.1 Introduction ....................................................... 107
  6.4.2 Construction of the Implementation Automaton ......... 108
  6.4.3 Construction of a Scheduling Automaton ............... 111
  6.4.4 Verification Experiences .................................... 113
6.5 Evaluation ............................................................ 114

7 Summary and Conclusion .............................................. 115
  7.1 Summary ............................................................ 115
  7.2 Conclusion ........................................................ 118

A Syntax Definition of Hybrid-ASTRAL ............................. 121

B Transfer Function Manipulations .................................. 131
  B.1 Introduction ....................................................... 131
  B.2 Laplace Transform To State-Space Description ......... 132
  B.3 Defining a Hybrid Automaton .................................. 134

C Robot Control System Specification .............................. 135

D Specification Support Environment ............................... 163
  D.1 Overview ........................................................ 163
  D.2 The H-Astral Front-End ....................................... 165
  D.3 The Target Specific Back-Ends ............................... 166
  D.4 H-Astral to Hybrid Automata .................................. 166
  D.5 H-Astral to HyTech ............................................. 167
  D.6 H-Astral Simulator ............................................... 167
  D.7 Astral to Extended Timed Graphs ............................ 168
  D.8 Astral to Uppaal ................................................ 168
  D.9 Astral Prototype ................................................ 169

Samenvatting ............................................................ 171

Curriculum Vitae ....................................................... 173

Index ............................................................................ 175

Bibliography ................................................................... 179
Preface

This thesis is the result of four years of research performed within the group Software Engineering, Programming, Programming Languages and Compilers (SEPPC). Our research concerned the role of a real-time software specification language in the development of embedded control applications.

The application of such a language in actual development requires not only a well-defined specification language. Tool support for the use of such a language is also important. In this thesis we consider automated analysis, simulation and prototyping of specifications.

I would like to thank my promotor, professor Jan van Katwijk, for offering me the opportunity to work in this challenging and inspiring environment. More than once he demonstrated his capability to quickly and precisely point out what the difficulties in my research are. To my regret, more than once I had to conclude that indeed I was wrong and he had been right from the beginning.

Hans Toetenel has also played a major role in my research. His suggestions, criticism and eye for detail kept the research going and he continuously tried to keep the focus on the ‘real’ subject of research. I acknowledge Philips ASA Laboratories, Eindhoven, for the opportunity given to me to finish this thesis.

Numerous of other people contributed to my research either through in depth discussions about interesting research topics or by supporting me in other ways. Naming them all here would definitely lead to forgetting a number of them. Therefore, I would like to thank you collectively for your support. But a special word of thanks goes to Jonelleke for her continuous and everlasting support during the past four years. She is the only one who really knows how difficult it was to finish this work.

Eindhoven, May 6th 1997,

Klaas Brink (kbrink@worldaccess.nl)
Summary

Interfacing Control and Software Engineering: a formal approach

The thesis focuses on the use of an (existing) formal software requirements specification language, called Astral, in the development of embedded real-time control applications. Astral, developed by Ghezzi and Kemmerer, is a language for the specification of requirements of real-time software.

As a starting point, development in the application domain, control engineering, is considered. Our focus is on the use of different kinds of block diagrams (graphical representations of a model of the control system) for modeling dynamic system behavior of both physical process(es) and the controller(s).

Then, a software engineering viewpoint is taken. Software development in general is discussed and the language Astral is described. Although the block diagram notation and Astral are different with respect to semantics as well as their expressiveness, we suggest to improve current development by developing a means to reformulate block diagrams (resulting from controller development) in an initial software requirements specification in Astral. This initial requirements specification, resulting from reformulation of a block diagram specification, is then to be completed by adding (typical) software requirements concerning (temporal) behavior in exceptional situations or synchronization between (parallel) operating controllers.

To facilitate reformulation and to improve requirements elicitation we develop an extended Astral language called Hybrid-Astral (or H-Astral). H-Astral is suitable for modeling systems consisting of discrete as well as continuous components. The theory of hybrid automata is discussed and used to define the semantics of H-Astral. A translational semantics for H-Astral is defined by mapping H-Astral specifications on hybrid automata. We also report our experiences on using a verification tool for hybrid automaton specifications, showing that only simulation seems currently applicable for realistically sized system specifications.
Then experiences on the use of H-Astral in the specification of a robot control system are reported. Also the use of simulation to analyze H-Astral specifications is discussed.

In addition to simulation, we discuss prototyping of Astral specifications using Ada95 as an implementation language. Practical experiences with prototyping reveal that timing analysis of the Ada95 prototype is difficult. Therefore, we discuss the use of hybrid automata for timing analysis of Ada95 prototypes and report upon our experiences on the use of such an approach.

We conclude by summarizing our experiences and performing an evaluation of negative and positive experiences regarding the use of H-Astral. Then our conclusions are presented concerning the applicability of the specification language H-Astral in the development of embedded real-time control applications.
Chapter 1

Introduction

1.1 General Discussion

Computers are found inside a large number of daily used systems in our society. Usually, the correct and safe operation of these systems depends on the correct operation of the software found inside them. Systems of which the computer is an integral part [7], are also called embedded systems. Such embedded systems are found in a variety of application areas like control, communication and diagnostics.

It is possible to consider computing systems from a different perspective in which, for example, possible effects of software failures are emphasized. Systems in which the consequences of software failures might result in high economic loss or loss of lives are called safety-critical systems. Typical examples of systems that are both embedded and safety-critical are control systems used in nuclear reactors and airplanes.

To minimize the number of errors in software for safety-critical embedded systems, adequate development methods are needed (in which techniques are used) that assure to a large extent the development of correct and reliable software [12]. Therefore, a major issue in research concerning software development for safety-critical systems aims at the development of methods and techniques that help in reducing the number of errors in the final software product. Errors can be introduced throughout the software development process, but it has been shown that most errors are introduced in early development phases [46].

A literature study has revealed three important sources of errors:

1. The first source of software errors is due to deficiencies in the software requirements specification [37]. Such errors result from incompletenesses of, ambiguities in or erroneous requirements in the software re-
CHAPTER 1. INTRODUCTION

requirements specification. For the development of high-quality software complete and unambiguous descriptions of the software requirements are regarded as essential [47]. This observation stimulates research aimed at methods and techniques that can be employed to capture software requirements completely and unambiguously.

2. A second source of software errors follows from the limitations of current software modeling techniques. These techniques only allow correct software behavior to be modeled and analyzed leaving analysis of the possible consequences of (erroneous) software behavior impossible. The effects of software behavior upon the environment cannot be studied and analyzed. For safety-critical systems, analysis of this kind is important, or as Leveson calls it [53]: "...computers are relatively safe devices: They rarely explode, catch on fire, or cause physical harm. However, computers can contribute substantially to accidents when they operate as a subsystem within a potentially dangerous system".

3. A third source of errors is due to the way development is carried out. Currently different components of an embedded system are developed independently and good communication between developers of the implications of design decisions within one discipline to other domains is lacking. This often results in systems that do not satisfy their requirements [11].

Thus it is necessary to consider the relation of software to the application domain. With respect to software development for process control systems Leveson remarks [52]: "Writing software requirements in isolation from the system-engineering process is bound to lead to problems". In case software requirements specification is considered in the development of embedded systems, two issues seem to be of major importance in this respect.

Firstly, the development of embedded systems should be regarded from a system perspective [49]. Software development is only a small part of the whole systems development process in which other disciplines are also involved. However, different development activities cannot be carried out independently; they are likely to interfere.

Secondly, once established, the software requirements specification is of major importance in subsequent software development. The specification is meant to be the source document for design and is used as a reference to assess the correctness of design decisions. Only then the development is likely to result in a software product that will operate as required, in a correct and reliable way. Applicability of a software requirements specification language is not solely determined by the ability to express software requirements. Its usability in subsequent software development must also be considered.
1.2. THE DEVELOPMENT PROCESS

The development of software requirements specifications for embedded systems, therefore needs to be considered from at least two viewpoints:

1. The application domain (the application-oriented view);
2. The software engineering point of view.

Negligence of the first viewpoint complicates the process of establishing software requirements (requirements elicitation) while negligence of the second will make the development of an application that conforms to the requirements specification more difficult.

In our opinion, the current generation software development methods puts too much emphasis on software requirements specification from the second viewpoint. Negligence of the first viewpoint carries a high risk of software development being regarded as a mono-disciplinary activity instead of a multi-disciplinary activity which is part of a system development process.

In the next section, software development for embedded control systems is discussed. It is shown that the relation between the application domain, control engineering, and software engineering domain is insufficiently addressed in the currently used software development methods. Recent developments in the area of software specification languages that could be employed to solve these problems are considered. Our subject of research is discussed in section 1.3. An overview of the contents of this thesis is given in section 1.4.

1.2 The Development Process

This section presents an overview of the development of embedded control systems. Section 1.2.1 addresses the development of embedded control systems and outlines the problems when the current development methods are considered from a software engineering point of view. In section 1.2.2 we consider requirements specification for embedded control software. This section focuses on the characteristics of embedded control software and its implications for software development.

1.2.1 Embedded Control Systems

The term 'embedded system' is used in various contexts and with widely different meanings. The following definition of an embedded system is therefore adopted here:

An embedded system consists of a computing system and a technical process (often called environment). The computing system is built into this technical process and its behavior depends upon and is determined by the behavior of the technical process.
Embedded systems, integrated into some larger system, are sometimes characterized as being environment-driven and reactive. The notion environment-driven indicates that the behavior of the computing system largely depends upon the behavior of the environment. The notion reactive indicates that the computing system reacts upon the behavior of the technical process, i.e., there is an ongoing interaction with the environment [54].

Since future embedded systems will become more and more complex [63] than today's, it is reasonable to expect that interfacing (or integration) of different development activities for these systems will become even more important. A suitable, multi-disciplinary development process with provisions for integrated development is needed [11].

A particular class of embedded systems are embedded control systems whose definition given here is based on a definition given by Kundig in [50]: An embedded control system consists of a computing system and a technical process. The computing system is embedded within a given plant or external technical process with the aim of influencing this process in a way that certain overall functional and performance requirements are met. It is common [28] to restrict this definition to: Those systems whose major function is to dynamically or actively command, direct or regulate.

The use of computers in the realization of today's control systems offers potential possibilities to build more complex (safety-critical) systems. In order to build this kind of systems right, development activities have to be carried out with care and precision. One possibility to achieve this is to use techniques to model and analyze the operation (especially the dynamics) of the control system [1].

An overview of important issues concerning the use of computers in control systems is given in the article "Contemporary Computers Considered Inappropriate for Real-Time Control" [35]. The article identifies software temporal behavior as being crucial for the correct operation of a process control system.

In many cases, software found in today's embedded control systems is designed and implemented by control engineers themselves. Often structured methods are employed that lack the possibility to formally verify properties of the software model [67]. Our experiences indicate that software development is in most cases carried out independently from the already developed control system models [48]. In some (limited) cases software is automatically generated based on control system models [55].

Our research hypothesis, which is addressed in section 1.3, was formulated based on experiences in a small case study. These experiences showed that a number of software requirements are not formulated explicitly during control system development [13]. This revealed one major problem: in order to
1.2. THE DEVELOPMENT PROCESS

develop the required software an active participation of a control engineer in all phases of the software development process is required. The fact that a number of important software requirements are not explicitly formulated is crucial. This bears a high risk of requirements being easily overlooked, resulting in the development of erroneous software.

The current paradigm of the development for embedded control systems is presented in Figure 1.1. It shows the discontinuity between control system models developed in the control engineering domain and the software models used for software development. The figure indicates the unclear relationship between development of control software and results from earlier controller development activities.

![Diagram](image)

Figure 1.1: Overview of current development process

1.2.2 Requirements Specification of Real-Time Software

From the observed discontinuity in the development process (see Figure 1.1) makes construction of a requirements specification for real-time control software difficult. In our opinion difficulties are caused by:

1. Different background and skills of engineers involved
2. Different notations in the disciplines involved.

In case of embedded control systems, software requirements are established by consulting control engineers or they are derived from already developed control system designs.

Various experiences on the use of software requirements specification notations for modeling real-time embedded control systems are reported in [8]. They show that the resulting software requirements specifications are not well-suited for use in control engineering. The approaches followed in the
control and software engineering domains have been shown to be incompatible in various ways [67].

However, research during recent years has led to the development of a number of formal software specification languages, notably languages for hybrid systems specification (hybrid systems are systems consisting of both continuous and discrete components). The resulting specification notations seem expressive enough to be used for development in both domains [6, 34]. Basically, the use of formal specification prevents ambiguity in a specification and enables formal verification. Practical experience in the use of formal specification languages in the development of realistically sized embedded control applications is still lacking however.

1.3 Research Topic

Our claim is that the main reason of the discontinuity, indicated with a question mark in Figure 1.1, is due to the differences between techniques applied in controller and software development. This is the cause of what might be called an ‘understandability problem’ [13] complicating communication between the control and software engineer. Because of this ‘understandability problem’, control system software is in most cases developed by the control engineers themselves.

Theoretical research concerning formal software specification has led to the development of expressive and relatively user-friendly software specification languages. These specification languages might be used to bridge the gap, indicated by the dashed line in Figure 1.1, between both domains of interest.

In this thesis we develop a formal software specification language as a vehicle to bridge the gap between the control engineering and software engineering domain. Our main focus can be formulated as follows:

To investigate the use of a formal software requirements specification language in and to develop technology to support the development process of embedded control applications.

In order to address the subject of research, the following topics need to be considered:

1. Analyze the development process in the application domain and select a usable software specification language; an inventory of development in the application domain should provide insight in the context in which a formal software specification language
is to be applied. The use of a formal specification language must be motivated and its application in the specification of real-time control software needs to be addressed;

2. Gain practical experiences;
   obtain experience in the use of the language through a non-trivial case study in modeling software requirements;

3. Support the process of constructing requirement specifications and define the semantics of the specification language,
   discuss what can be done to facilitate the construction and to improve the quality of these requirement specifications. This requires a mathematical formalism which can be used to define the language semantics and enables analysis of requirement specifications.

Our former experience in using a formal specification language called Astral [32] has led to prefer Astral’s use. However, motivations for both its use and for its applicability in the specification of software requirements for real-time control systems is addressed. The desired situation is schematically depicted in Figure 1.2.

1.4 Contents of this Thesis

Chapter 2 discusses the development within the application domain, control engineering. The overview presented, takes a software engineering point of view and puts main emphasis on those elements of control system development that are important in the development of the control software. This
Chapter 1. Introduction

Chapter concludes with an overview of software engineering related difficulties with current development practice.

Chapter 3 discusses the software engineering domain. A formal specification language called Astral [32] is proposed and its use in the specification of control systems software is evaluated.

From the evaluation in chapter 3 it becomes clear that in order to be applicable some extensions to Astral are needed. An extension of Astral called Hybrid-Astral (H-Astral) is therefore developed in chapter 4. In this chapter the semantics of the H-Astral language is defined by a transformation of H-Astral specifications into sets of machine descriptions, called hybrid automata. In this way, the theory underlying hybrid automata is used in defining the semantics of H-Astral.

Chapter 5 evaluates the applicability of the proposed approach with respect to the development in the control engineering domain. It evaluates the usability of the specification language by performing a case study concerning the specification of a robot control system.

In chapter 6, we elaborate on automated support which is regarded as important in the construction of H-Astral specifications. In particular, it is discussed how H-Astral specifications can be prototyped using Ada95 as an implementation language.

Finally, chapter 7 contains conclusions that can be drawn from this work and suggestions for future research are made. This section evaluates the use of a formal software specification language with respect to the following criteria:

- Advantages and disadvantages of using a formal software specification notation in the controller development process;
- Suitability of the H-Astral language;
- Available support for the construction of requirement specifications;
- Extent to which application of a formal software specification has succeeded in removing the observed discontinuity in the development process;
- Applicability in practice.
Chapter 2

Control Engineering: An Introduction

2.1 Introduction

In this chapter, the development of embedded control systems with main emphasis on development in the control engineering domain is addressed. The main purpose of this overview given is to show in what ways development in the control engineering domain and the software engineering domain are interrelated. Problems with respect to software development that result from the current approach are summarized. They show the need for another approach to the development of software for embedded control systems.

This chapter focuses on control system development in which a graphical representation of the control system is used (called block diagram). Different kinds of block diagram models are discussed in section 2.2 and their use is illustrated with examples. The discussion concerning the block diagram notation aims at: (i) providing the necessary background to understand the meaning of block diagrams, and (ii) pointing out the significance of the block diagram notation for development in the control engineering domain.

The current development process, based on the use of a block diagram model of the control system, is addressed in subsection 2.3. The discussion is structured by dividing the development process in two phases. In the first phase, a block diagram model of the control system is developed. The second phase addresses the design and implementation of the control software.

An evaluation of control system development follows in subsection 2.4. It is performed by taking a computer science point of view and aims at answering the following two questions:

1. What are the implications of the use of a block diagram model for
software development?

2. What are the problems in current practice of software development based on block diagram models of control systems?

The answers to these questions motivate research for an alternative approach to resolve the above mentioned difficulties in embedded software development.

### 2.2 Block Diagram Models

Block diagrams are a widely used graphical representation of control system models in the control engineering domain. A number of different interpretations for block diagrams are discussed in this section. An example is used to illustrate how and why these block diagrams are used in the development of control systems.

For a more detailed discussion and a more extensive introduction to the subject the reader is referred to the control engineering literature [1, 19, 22, 28, 29].

#### 2.2.1 Global Structure

![Block Diagram](image-url)

**Figure 2.1: A general block diagram**

In general, a block diagram is a graphical representation of a system model. Two basic elements of a block diagram are: (i) blocks and (ii) interconnections between blocks. Consider the general representation of a single block like the one depicted on the right of Figure 2.1. Such a block has one incoming arrow called an input (denoted I), one outgoing arrow called an output (denoted O) and a label, $S$, called a transform. A transform defines the relationship between the input and output of a block.

Generally, a block diagram consists of a number of interconnected blocks. An example is shown on the left of Figure 2.1. Interconnections can be made...
2.2. **BLOCK DIAGRAM MODELS**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Transform</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>continuous</td>
<td>continuous</td>
<td>linear, time-invariant differential equation</td>
<td>2.2.3</td>
</tr>
<tr>
<td>discrete</td>
<td>discrete</td>
<td>linear difference equations</td>
<td>2.2.4</td>
</tr>
<tr>
<td>discrete or continuous</td>
<td>discrete or continuous</td>
<td>linear difference and/or linear time-invariant differential equation</td>
<td>2.2.5</td>
</tr>
<tr>
<td>discrete or continuous</td>
<td>discrete or continuous</td>
<td>non-linear equations</td>
<td>2.2.6</td>
</tr>
<tr>
<td>discrete or continuous</td>
<td>discrete or continuous</td>
<td>function, linear difference and/or linear time-invariant differential equation</td>
<td>2.2.7</td>
</tr>
<tr>
<td>data</td>
<td>data</td>
<td>program procedure</td>
<td>2.2.8</td>
</tr>
</tbody>
</table>

Table 2.1: Classification of block diagrams

between an output of one block and an input of another block. The block labels $H1$, $H2$ and $G$ represent the transforms of each of the blocks in the diagram.

In a block diagram, an interconnection can be split or a number of interconnections can be joined. These elements of a block diagram are called a *take-off point* and a *summing junction* respectively (see also Figure 2.1). The edges leaving a take-off point all represent the signal value present on the incoming edge. The signal leaving a summing junction equals the sum of the signals entering it. An entering signal labeled with a '−' sign is negated before being added to the other incoming signals.

In control engineering, block diagrams are a widely used modeling notation. However, the precise interpretation of a block diagram can be diverse, depending the situation in which it is used. A block diagram can be used to represent different kinds of system models. Block diagrams have therefore been classified based on the precise nature of input and output signals and the specification of the transform of the blocks. An overview of the different classes that will be considered in the following sections is given in Table 2.1.

Before turning over to a discussion of each of these classes a simple example is introduced in the following section. The example is used to illustrate the use of a block diagram in each case considered.
2.2.2 An Example: The Inverted Pendulum

Throughout the discussion of the block diagram notation the inverted pendulum [29] as depicted in Figure 2.2 serves as a running example.

The inverted pendulum system consists of a cart with a stick mounted on top. The stick (or pendulum) rotates around a mounting point. A force, \( u(t) \), can be applied to the cart. The main goal is to control the force, \( u(t) \), in such a way that the pendulum is kept in its upright position.

![Figure 2.2: The inverted pendulum](image)

This system can be modeled using the following variables:
- Angle of rotation \( \theta \);
- Mass of the cart \( M \);
- The mass of the stick \( m \);
- Length of the pendulum \( l \).

For notational convenience the following shorthand notation for the first and second derivative of a variable \( x \) with respect to time is introduced: \( \dot{x} = \frac{dx}{dt} \) and \( \ddot{x} = \frac{d^2x}{dt^2} \).

In case the system is in equilibrium both the sum of the torques (\( T \)) and forces (\( F \)) equal zero. This observation is used to derive the following equations:

\[
\sum T = ml \cos(\theta) \ddot{y} + ml^2 \ddot{\theta} - mgl \sin(\theta) = 0 \quad (2.1)
\]

\[
\sum F = M \ddot{y} + ml \cos(\theta) \ddot{\theta} - u(t) = 0 \quad (2.2)
\]
2.2. BLOCK DIAGRAM MODELS

2.2.3 Linear, Continuous-Time Systems

In control engineering, the physical process behavior can often be modeled through differential equations. In these equations, the relation between input and output signals of the process is captured.

A certain class of differential equations used to model (continuous time) behavior of time-invariant linear systems are so-called Ordinary Differential Equations (ODE) [28] of the following kind:

\[
\begin{align*}
& a_n \frac{d^ny}{dt^n} + a_{n-1} \frac{d^{n-1}y}{dt^{n-1}} + \cdots + a_1 \frac{dy}{dt} + a_0 y = \\
& b_m \frac{d^mu}{dt^m} + b_{m-1} \frac{d^{m-1}u}{dt^{m-1}} + \cdots + b_1 \frac{du}{dt} + b_0 u
\end{align*}
\]  

(2.3)

Models of this kind can be analyzed, e.g., for stability (that is, whether or not the system returns to a steady state when a certain input signal is applied) and performance (that is, how much time elapses between the time instant at which the system input is changed and the time instant at which the system is in steady state again).

Equations 2.1 and 2.2, modeling the behavior of the inverted pendulum, can be transformed in a linear differential equation using the following assumptions: \( M \gg m, \cos(\phi) \approx 1, \sin(\phi) \approx 0 \) (when \( \phi \approx 0 \)). Under these assumptions equations 2.1 and 2.2 reduce to:

\[ M g \theta - M l \ddot{\theta} = u(t) \]  

(2.4)

The resulting differential equation is a linear, time-invariant differential equation. Important in the analysis of these equations is the Laplace transformation [19]. The Laplace transformation allows analysis and manipulation of this kind of differential equations. Therefore the Laplace transform of the differential equation governing the behavior of a linear time-invariant system is often used instead of the differential equation itself.

Application of the Laplace transformation to the equation 2.4 (under the assumption that the values of all variables and their derivatives initially equal zero) results in the following Laplace transform specifying the relation between input (force applied to the cart) and output (rotation angle) of the inverted pendulum (note that the simplifying assumptions made earlier do not become immediately apparent from the transform):

\[
\frac{\Theta(s)}{U(s)} = \frac{-1}{M l s^2 - M g}
\]
Using known techniques, associated with the Laplace transformation, and analogous to those applied in modeling the inverted pendulum, a control algorithm can be developed. A popular and simple control strategy, called Proportional Integral Derivative (PID) control, can be applied to control the force applied to the cart. Basically, the output of a PID controller is dependent on current and past values of its input signal (denoted \( e(t) \)).

In our example the output of the PID controller equals the force applied to the cart (\( u(t) \)) and is defined by the following equation:

\[
K_p e(t) + K_i \int e(t) dt + K_d \dot{e}(t) = u(t)
\]  

(2.5)

The current value of the control signal \( u(t) \) depends upon on the current value of \( e(t) \) (with weight \( K_p \)), the summation of past values (with weight \( K_i \)) and the first derivative (with weight \( K_d \)).

The model of this PID controller (represented by its Laplace transform) is:

\[
\frac{U(s)}{E(s)} = K_p + \frac{K_i}{s} + K_d s
\]  

(2.6)

The control system model which consists of the model of the inverted-pendulum system and the controller is graphically represented by the block diagram depicted in Figure 2.3. In this case, the value of the input signal \( e(t) \) to the block representing the PID controller is equal to the difference between the disturbance, \( d(t) \), and the current angle of the stick, \( \theta(t) \).

The block diagram representation depicted in Figure 2.3 is a representation of the linear, time-invariant model of the inverted pendulum control system. The interconnections between blocks represent the continuous time signal values of the system. Transforms are defined by Laplace transforms of linear, time-invariant differential equations.

![Figure 2.3: Linear, time-invariant system](image)

The main task is to determine the values of \( K_p, K_d \) and \( K_i \) such that the pendulum is kept in the upright position even in the presence of (small)
disturbances. It must be analyzed to assure that the system as a whole is stable, i.e., the rotation angle will return to zero after occurrence of a (small) disturbance with a finite duration.

The model of the inverted pendulum control system as discussed here is linear and time-invariant (see Figure 2.3). In this case, optimal values of $K_p, K_d$ and $K_i$ (given certain constraints concerning response time and stability) can be obtained through mathematical analysis.

### 2.2.4 Linear, Discrete-Time Systems

Behavior of discrete-time systems is usually modeled through difference equations instead of differential equations. It is assumed that a continuous time signal is observed at equidistant points $kT$ in time.

There exists a well-developed theory for a class of systems that can be described by the following kind of a difference equation:

$$
\sum_{i=0}^{n} a_i y(k + i) = \sum_{i=0}^{m} b_i u(k + 1)
$$

Discretization (using first-order approximation) of the continuous time model of the inverted pendulum (see equation 2.4) results in the following linear difference equation:

$$
Mg\theta(kT) - Ml \frac{(\theta(kT) - 2\theta((k - 1)T) + \theta((k - 2)T))}{T^2} = u(kT) \quad (2.7)
$$

Discretization of the differential equation modeling the controller (see equation 2.5) results in:

$$
u(kT) = u((k - 1)T) + K_p(e(kT) - e((k - 1)T)) + \frac{K_i T}{2} (e(kT) + e((k - 1)T)) + \frac{K_d}{T} (e(kT) - 2e((k - 1)T) + e((k - 2)T)) \quad (2.8)
$$

Analogous to the Laplace-transformation as used for modeling linear, time-invariant systems another transformation called the ‘z-transformation’ has been defined for this kind of discrete systems. The z-transformation facilitates the analysis and manipulation of these difference equations.

Applying the z-transformation to equation 2.7 results in:

$$
\frac{\Theta(z)}{U(z)} = \frac{T^2 z^2}{(MgT^2 - Ml)z^2 + 2Mlz - Ml} \quad (2.9)
$$
and the $z$-transform of equation 2.8 equals:

$$\frac{U(z)}{E(z)} = \frac{(2TK_p + KiT^2 + 2K_d)z^2 + (K_iT^2 - 2TK_p - 4K_d)z + 2K_d}{2Tz(z - 1)}$$ (2.10)

$$a_2 = 2TK_p + KiT^2 + 2K_d$$
$$a_1 = K_iT^2 - 2TK_p - 4K_d$$
$$a_0 = 2K_d$$

Figure 2.4: Linear, discrete-time system

A block diagram can be used to represent the discrete control system model. In this case, all transforms of the blocks are defined by $z$-transforms and interconnections represent discrete signal values (at time instants $kT$). An example of a block diagram representing the discrete model of the inverted pendulum control system is shown in Figure 2.4.

In this case, the block diagram is equivalent to a (discrete) mathematical model of the system expressed by a difference equation or its $z$-transform.

### 2.2.5 Linear, Discrete-Continuous Time Systems

In practice, block diagrams consisting of both discrete and continuous blocks are also encountered. For some blocks in these diagrams, the relation between input and output signal is discrete, i.e., defined by a $z$-transform, and for other blocks it is continuous, i.e., defined by a Laplace transform.

In the case of the pendulum control system such a block diagram can be constructed. It results from the block diagram depicted in Figure 2.3, in which the Laplace-transform of the block modeling the controller (see equation 2.6) is replaced by its discrete counterpart, the $z$-transform as defined by equation 2.10.

Underlying these block diagrams consisting of both discrete and continuous blocks are the following assumptions:

1. Signal values can only be observed at discrete (sampling) instants, i.e., despite the suggestion that parts of the system are continuous, sig-
2.2. BLOCK DIAGRAM MODELS

...n... values are only known and can be analyzed only at discrete time instants.

2. In case an output of a continuous is directly connected to the input of a discrete block, a sampling device is assumed to be present (often not explicitly drawn in the block diagram representation).

3. In case an output of a discrete block is directly connected to an input of a continuous block, a D/A converter called a zero-order-hold block (or ZOH-block) is assumed to be present (often not explicitly drawn in the block diagram representation).

2.2.6 Non-Linear Systems

The three different classes of block diagram models discussed until now have in common that they represent linear systems. Linear systems obey the linearity principle defined in [28] as follows:

Definition 2.1 If all initial conditions in the system are zero, that is, if the system is completely at rest, then the system is a linear system if it has the following property:

1. If an input \( u_1(t) \) produces an output \( y_1(t) \), and

2. an input \( u_2(t) \) produces an output \( y_2(t) \),

3. then input \( c_1 u_1(t) + c_2 u_2(t) \) produces an output \( c_1 y_1(t) + c_2 y_2(t) \) for all pairs of inputs \( u_1(t) \) and \( u_2(t) \) and all pairs of constants \( c_1 \) and \( c_2 \).

A system is called non-linear when it does not obey this definition of linearity. Mathematical analysis of non-linear system models is difficult, though in this area advances are being made [1]. Restricted cases of non-linear models can be analyzed. In general, simulation of these models is the only practical means to study system behavior.

An example of a non-linear model is the model of the inverted pendulum defined by non-simplified equations 2.1 and 2.2. From a comparison of these equations and the general differential equation modeling a linear system (see equation 2.3) it follows that non-linearities are due to the presence of \( \cos(\theta) \) and \( \sin(\theta) \) in the coefficients of the differential equation.

![Non-linear system](image)

Figure 2.5: Non-linear system
A non-linear block diagram is depicted in Figure 2.5. It contains a block which is called non-linear because its transform is defined by a function which is dependent on the value of its input. A block diagram is called non-linear in case it contains at least one non-linear block. Analogous to the previous cases considered, a non-linear block diagram is a graphical representation of a set of equations. The main difference with all previous cases considered is in the specification of the transforms. In addition to being defined through Laplace or z-transforms the model can also be defined by non-linear functions that explicitly depend on the value of the input.

2.2.7 System Simulation Model

Simulation software packages such as [44, 57] for example, allow one to define a (numerical) simulation model through the use of a block diagram. In many cases, these block diagrams closely resemble the block diagrams that have been discussed in sections 2.2.3 to 2.2.6. Block diagrams used for simulation purposes are considered separately here to emphasize their use for simulation of system behavior. All signal values are calculated using numerical (discrete) techniques.

Because the main goal of simulation is numerical, not analytical, computations, transforms can be defined in different ways and are not restricted to either z- or Laplace transforms. Restrictions that are put on the structure of block diagrams and the definition of transforms originate from employing numerical techniques for simulation.

In order to perform simulation, it is necessary to incorporate special kinds of blocks in the simulation block diagram. Such blocks are, for example clocks needed for the specification of sampling frequency (in case the block diagram contains discrete blocks). Furthermore, the block diagrams used in simulation are annotated with attributes that: (i) define numerical integration routines to be used in the calculation of signal values, and (ii) define the time interval for which system behavior must be simulated.

An example of the block diagram for simulation of the inverted pendulum control system (using SCICOS [44]) is shown in Figure 2.6. Each block represents an equation governing the relation between its input and output signal or it is a special kind of block like a clock or an oscilloscope. In the former case the kind of relation which can be specified is restricted only by limitations of current numerical techniques employed in simulation.

The kind of systems that can be simulated are both linear and non-linear systems in which discrete as well as continuous components can be mixed. Again, it is implicitly assumed that, in cases where discrete and continuous components are directly connected, ideal sampling devices and D/A devices
(ZOH-blocks) are present [57]. A simulation of the system model defined by the block diagram in Figure 2.6 over the time interval \([0, 20]\) is depicted in Figure 2.7. This figure depicts how the system responds when a constant disturbance is applied at \(t = 0\) and all initial conditions are equal to zero at \(t = 0\).

### 2.2.8 Control Software Model

Finally the use of block diagrams as a model for (automatic) code generation. Most notable in this respect is the SIMULINK C code generator [55]. It is important to realize that there exists a difference between block diagrams models used for simulation and those used for control software implementation. When used as an implementation model, the blocks in a block diagram represent the program code that implements the operation performed by the block on its input signal(s) to calculate the value of the output signal.

When using the block diagram as a representation of a program, additional blocks are available that represent fragments of device driver software. These device driver blocks are connected to the inputs or outputs of the block diagram that represents the controller that is to be realized in software. The resulting block diagram is in many ways comparable to a data-flow diagram [8]. Based on the block diagram the software is generated.

In general, no possibilities exist for the verification of the timing behavior of the resulting software. Furthermore the restricted expressiveness offered
by the block diagram has implications for the kind of software that can be generated.

2.3 Controller Development Process

The control system development process, based on using the block diagram modeling notation, is hardly addressed in control engineering literature. Instead, attention merely focuses on the techniques that can be applied to solve specific control problems, such as development of the control algorithm. This observation is based upon a study of available literature concerning development of a robot control system [48], control systems for active suspension [65] and has been partly based upon the description of a framework for development of embedded control systems discussed in [67].

Based upon these studies control system development, when considered from a software engineering viewpoint, can be regarded as a two-phase pro-
cess. This is depicted in Figure 2.8. In the first phase attention is focused on the development of a system model (represented by a block diagram). This phase is addressed in subsection 2.3.1.

The second phase of control system development aims at the realization in software of the controller model that resulted from the first phase. This phase, focusing on the transition from the first to the second phase, is discussed in subsection 2.3.2.

![Figure 2.8: Control system development as a two-phase process](image)

### 2.3.1 Phase 1: Construction of System Model

The first phase starts with the construction of a mathematical model of the physical process that is to be controlled. In the cases that were studied [48, 65], the physical process was first decomposed in a number of more or less independent components in order to reduce the complexity of model construction. Development then continued with modeling these components of the system separately.

In the cases considered, the construction of a physical model is not preceded by an activity in which its required characteristics are explicitly formulated. Although there are specific requirements which must be met, they are usually formulated and verified during the development process.

These physical models can be based upon or derived from standard models and well-known techniques are used during their development. An overview of standard models and well-known techniques is presented in [29]. Afterwards the different parts are integrated into a complete model of the physical process. Throughout this phase, verifying the correctness of the constructed model, is often done through comparison of results from model analysis (including simulation) with the actual process behavior.

When the physical process model is sufficiently accurate, development continues with the creation of a controller model. Development decisions,
such as the kind of control strategy to be employed are taken, based on experiences in previous, comparable cases or application of available (sometimes well-known, sometimes experimental) techniques in controller synthesis [29].

The control system model, consisting of the controller model and the physical process model (see Figure 2.9), is then constructed and subsequently analyzed. The controller model is modified whenever results from analysis do not satisfy requirements. This iterative process, consisting of modeling and analysis is performed repeatedly until the results are satisfactory.

![Control System Model](image)

Figure 2.9: Model resulting from first phase

It is obvious that in the development of a controller, modeling and analysis are of main importance. This first phase of control system development can be characterized as a highly iterative process in which modeling and analysis activities are repeatedly performed. The block diagram notation introduced earlier is one technique used in the development of a control system model.

Many Computer Aided Control System Design (CACSD) tools offer support mainly in the first phase focusing on modeling and analysis. One example are the simulation tools, such as Scilab [44], to perform (numerical) simulation of system behavior (as discussed in section 2.2.7).

### 2.3.2 Transition to Phase 2

In phase 2 development is based on the controller model as established in phase 1. Usually this transition from phase 1 to phase 2 is informal, in some cases the controller model is reformulated in another notation (for example, a modeling notation used in structured software development methods [67]) as described in [8]. In other cases, the block diagram itself is used but its
interpretation is changed (for example, used as a model of control software, as discussed in subsection 2.2.8). In phase 2 attention is focused on realization of the controller model in software and interfaces to the physical process as depicted in Figure 2.10.

![Figure 2.10: Phase 2: design and implementation of controller](image)

The techniques used in modeling control systems in phase 1 are aimed at (mathematical) modeling and analysis of dynamic system behavior. These techniques lack expressiveness with respect to a number of important software characteristics of control systems and thus cannot be used to model a control system completely.

This necessitates either the reformulation of the controller model in another notation or changing the interpretation (semantics) of the block diagram model of the controller. Reformulation in another notation is usually performed when typical characteristics of control system software must be expressed and analyzed, such as parallelism, composition of different controllers, synchronization and the definition of controller behavior in exceptional situations, etc.

## 2.4 Evaluation

In this section, a number of difficulties concerning control system development are summarized. It is important to note that this evaluation is restricted to the control system development process in which a block diagram model of the physical process and controller is developed. Other development approaches in control engineering exist in which the physical process is modeled in a different way or sometimes not at all. Our attention will be restricted to the development based on the use of block diagrams.

Block diagrams are widely used in control engineering for the representation of control system models. The structure of a block diagram often bears a clear correspondence with the physical system being modeled, which might
be one of the main motivations to use them. However, using block diagrams for the representation of different classes of models makes an unambiguous interpretation of block diagrams by software engineers difficult. The precise interpretation of a block diagram is dependent on the situation in which it is used.

Block diagrams contain elements that model the dynamic behavior of software components that may have to be implemented in software. From a software engineering viewpoint, block diagram models are not appropriate for defining requirements concerning parallelism, synchronization, error detection, error handling/recovery and timing behavior of control software.

On the other hand, one possible interpretation used the block diagram is as a software model based upon which control software is automatically generated. This provides evidence for the fact that block diagrams are in some way related to the control software and can thus not be neglected in control software development.

However, in the process of modeling, abstractions or simplifying assumptions have been made which do not become directly clear from the block diagram model itself. In our opinion, these abstractions and assumptions should be made explicit.

The transfer from phase 1 to phase 2 in development causes an 'understandability problem' [13]. In our view the reasons for these problems in communication between developers are: (i) currently used notations to reformulate controller models lack a sound, mathematical basis, which causes ambiguities and complicates analysis, (ii) a framework for a systematic transfer of controller models from phase 1 to phase 2 is lacking, and (iii) the physical process model is completely discarded implying that the effect of certain design decisions cannot be studied with respect to their consequences for the behavior of the physical process.

Important software requirements, especially timing requirements, are implicit when block diagrams are used. In that case the transition of a controller model into a software model is difficult and a discontinuity is clearly present as has been expressed in Figure 1.1.

Important software requirements should, from a software engineering viewpoint, be made explicit. Only in that case possibilities are created to verify during software development whether the software being built satisfies its specification. Especially for control systems that are safety-critical this seems important.

Main arguments for the development of an alternative approach are based upon the use of block diagrams, a common and well-known formalism for the specification of control systems. We have however tried to develop an alternative approach which is more general and whose application does not depend
2.4. EVALUATION

on the use of block diagrams. In subsequent chapters such an alternative approach is presented and its application is studied in the context of using the block diagram modeling technique.
Chapter 3

Requirements Specification in Astral

3.1 Introduction

Software for real-time control systems is subject to both functional and timing requirements. In case the software fails to meet these constraints the consequences can be severe. Development techniques should aim at minimizing risks of errors. The main goal of this chapter is to introduce the reader to a requirements specification language, called Astral [32], which is suited for requirements specification of real-time software and, in our opinion, seems applicable in the development of real-time, embedded control applications.

The focus is on software requirements specification, but it is important to elaborate on the software development process first, of which software requirements specification is only a single phase. Although software requirements specification is regarded as a major source of software errors, the risk of errors is present in all phases of the development. To minimize this risk, it is required that all development activities are carefully planned, managed and carried out with great precision.

A model of the software development process is used that embodies how, in our opinion, real-time control software should be developed. In such a model it is common to structure the software development process by: (i) dividing it into a number of phases, (ii) impose some order upon them, and (iii) clearly stating the input and output of each development phase. This model is presented in section 3.2.

This process model is rather abstract. It does not prescribe how development in each phase should be carried out to achieve the required result. How to carry out development activities and which techniques can be applied is
prescribed by what is called the method. A method defines how development activities are to be carried out, i.e., which techniques to use and giving guidelines on how they are to be applied in order to produce the required output. Formal methods (methods based on a mathematical formalism) and their usability in software requirements specification is of particular concern.

Section 3.3 introduces formal methods, motivates the application of formal methods in the development of real-time control software and suggests using an existing formal specification language, called Astral [32], for requirements specification. This will explain our preference to use a formal method to specify requirements of real-time control software.

Then attention is restricted to the software requirements specification phase. Because development decisions are based on the resulting software requirements specification, a major interest (from a software engineering point of view) is in a complete and realizable (consistent) requirements specification. A consistent and complete requirements specification is regarded as a prerequisite for development of the right control software.

Section 3.4 discusses the specification language Astral. The use of the language is illustrated by discussing an example specification concerning a specification of the so-called Generalized Railroad Crossing (GRC) [37].

The relation between the controller model which results from development in the CE domain and the software requirements specification in Astral, is then addressed in section 3.5. In the final section (section 3.6) the applicability of Astral is evaluated. A number of issues related to the application of Astral are then formulated which are resolved in subsequent chapters.

3.2 A Model for Real-Time Software Development

In this section, a model for software development is introduced and some attention is given to software development methods in general.

When describing how development of some system is carried out, a distinction is made between a process and a method [49]:

**Definition 3.1** A process is an activity which takes place over time and which has a precise aim regarding the result to be achieved

**Definition 3.2** A method defines a way to conduct a process

Informally, a method defines how a certain goal can be accomplished while a process refers to actual development activities in the realization of some product and period in which these development activities are performed.
3.2. A MODEL FOR REAL-TIME SOFTWARE DEVELOPMENT

Conceptually, it is useful to think of the software development process as consisting of a number of phases. Structuring the software development process by partitioning it into a number of distinct phases and by stating their goal (task to be accomplished) and their output (the result to be produced) is common in managing the complexity of the development process [66].

Each phase represents a specific development activity which takes place in the course of building the final product. The output of a certain phase is either an (intermediate) product which in its turn serves as input to other development phases or it is the required end-product to be developed.

Development within one phase is iterative and is aimed at the construction of the required output of this phase of development. Each phase comprises building some kind of representation of the system under construction. This representation is analyzed, validated (checking whether the required product is being developed), and adapted on the basis of analysis of the results. Within one phase, these activities are iteratively performed until a satisfactory result is achieved.

The division in a number of phases is also present in the waterfall model of software development [64] and the precise input-output description of each phase is clearly apparent in the contractual model of software development [24].

As experience grew, the iterative nature of software development was acknowledged and has been emphasized in later software development models. This, for example, led to Boehm's spiral model of the software development process [64]. Re-iteration over previous phases of software development is important and has even been shown to be inherent to the development of real-time software as exemplified by the HRT-HOOD development model [21].

A model of the software development process should clearly represent these characteristics. Such a model in which this iterative development process within each phase as well as in between different development phases is apparent, has been depicted in Figure 3.1.

The development process presented in Figure 3.1, consists of four different phases:

1. **Software Requirements Specification**, to which the (informal) user requirements are input and the requirements specification is an output;
2. **Design**, in which a design proposal is constructed based on the requirements document;
3. **Coding**, where the actual construction of a software implementation occurs, which conforms to the design proposal;
4. **Code Timing / Testing**, in which the software is subject to timing
measurements and tests in order to verify that the product conforms to the specification.

The software development model is a convenient means to structure the development process. It does not describe, however, how each development activity is or should be performed. This aspect of development is described by the method.

Central to a method are the capabilities offered to describe and analyze an abstraction of a real-world entity. Such an abstract representation is sometimes named a model defined in [49] as follows:

**Definition 3.3** A model is a representation of some thing actual or contemplated, with relevant characteristics being the same as the real-world entity being modeled. Which features are relevant depends on the circumstances and intended use of the model.

In accordance with [49], the following elements of a method are distinguished here:

1. An underlying model, which determines the kind of real-world entities that can be represented;
2. A language or notation, providing the concrete means to define entities and interactions between entities;

3. Steps and their ordering, that is, a set of recommendations concerning the way the language can be applied and used in specification and analysis.

The scope and nature of the various methods applied in software development varies. They differ in two respects: (i) the phases of the software development process in which they can be used and (ii) their level of formality [37]. Some methods are only used in one particular phase of development while other methods cover all phases of the software development process. A method can be based on some mathematical formalism in which each entity that is modeled is given a precise, mathematical interpretation, while other methods lack such a precise interpretation.

More information about and comparisons between different methods is presented in [12, 37, 49]. In most cases, the choice of a specific method is based on criteria like: (i) the level of preciseness (and analysis capabilities) offered, (ii) expressiveness, and (iii) simplicity of the specification notation.

3.3 Formal Methods

In this section the use of formal methods (as opposed to more informal methods) in software development is discussed. It also presents our motivations for proposing the application of formal methods in the development of real-time control software. We then focus on the software requirements specification phase of development and select a formal software specification language which is suitable for the requirements specification of real-time control software.

Whether or not a method can be characterized as 'formal' is a rather subjective matter [62]. Our definition is based on a definition given in [37] in which the term 'formal' applies to the model, underlying the method:

**Definition 3.4** A formal method is a method which is based on a formal model.

To define the model underlying a formal method, a mathematical formalism is used. Based on this formalism a specification language is then developed. From a user's (application) point of view, the specification language of a method is of main importance.

We will shortly touch upon the nature of a formal model by using mathematically well defined constructs like sets, functions and sequences. We can use these constructs in modeling a train passing some railroad crossing in the
following way: The train is represented by a set consisting of one element which represents the train. This set, called \( \text{Thetrains} \), can be defined as follows:

\[
\text{Thetrains} = \{ T1 \}
\]

The only positions of interest of the train are simply whether it is far from, approaching, or on the crossing. The possible train positions are defined by the following set, named \( \text{Trainpos} \):

\[
\text{Trainpos} = \{ \text{Far, Near, In} \}
\]

The current position of the train is given by a function:

\[
\text{TP} : \text{Thetrains} \rightarrow \text{Trainpos}
\]

In much the same way we model the gate and gate-position of the crossing:

\[
\begin{align*}
\text{Thegates} & = \{ G1 \} & \text{(gate)} \\
\text{Gatepos} & = \{ \text{Open, Closed} \} & \text{(possible gate positions)} \\
\text{GP} & : \text{Thegates} \rightarrow \text{Gatepos} & \text{(current gate position)}
\end{align*}
\]

The state of the system, consisting of train \( T1 \) and gate \( G1 \) can, at a discrete time instant; be represented by taking the value of the function \( \text{TP} \) and \( \text{GP} \) at this time instant, denoted as follows: \( (\text{TP}(T1), \text{GP}(G1)) \).

The system can then be modeled by a fictitious observer which at certain time-instants registers the state of the system. The fictitious observer thus takes snapshots and the resulting state sequence represents a possible observation sequence of the system:

\[
(\text{Far, Open}) \rightarrow (\text{Near, Open}) \rightarrow (\text{Near, Closed}) \rightarrow \\
(\text{In, Closed}) \rightarrow (\text{Far, Closed}) \rightarrow (\text{Far, Open}) \rightarrow \cdots
\]

This system can, for example, be modeled by the set of allowed (possible) observation sequences. In this case the set is simply the set of all possible observation sequences except all sequences in which \( (\text{In, Open}) \) occurs. The state \( (\text{In, Open}) \) is of course, possible, although undesirable.

One (major) motivation to apply a formal method is given in [37]:

"Formal methods may use formal specification languages and formal modeling techniques to describe required system behavior and formal analysis techniques to demonstrate that system behavior satisfies critical properties."

In the context of control engineering, there is another reason to prefer a formal method for requirements specification. In our case, formality is required to prevent misunderstandings which are easily introduced because
3.3. FORMAL METHODS

engineers from different disciplines are involved. This will help in minimizing the number of errors in the requirements specification and is expected to result in more reliable software. A more extensive discussion concerning the use and (dis)advantages of formal methods in software development for real-time systems can be found in [12, 37, 61].

For requirements specification of real-time control software a requirement specification language is needed which:

1. allows definition of required behavior (operational specification);
2. is usable to define required properties (and behavioral constraints);
3. offers sufficient expressiveness to define both functional and (quantitative) timing requirements;
4. is simple so it can be used by both control- and software engineers and its application does not require too many (domain) specific skills in order to be applied.

We prefer behavioral specifications, in which a system is modeled as an abstract machine. This is, for example, the case in a transition system which is based on a precise definition of state and in which state changes are defined by transitions. A behavioral specification is much more intuitively clear than, for example, an axiomatic specification, in which only the (required) properties of the system are defined. Thus a behavioral style of specification, using an easy-to-learn (simple) language with a small number of primitives is expected to facilitate use by control and software engineers.

Being able to explicitly define constraints on or required properties of the system’s behavior, is also regarded as important. This ability serves two different goals:

(i) to put emphasis on those system properties that are crucial for correct functioning of the system (properties that should follow from the specification), and

(ii) to formalize additional constraints on system behavior (properties that are required to hold for the implementation).

Obviously, the aim is to produce a specification of real-time control software. It is therefore required that both functional and timing behavior is defined in requirement specifications.

The specification language Astral [32] meets the requirements stated above, but in that respect it is not fundamentally different from other specification languages (like for example Statecharts [36] or Modechart [37]).

It was selected for use for two reasons: first, Astral offers a small number of primitives that are easy to learn and it allows an operational style of specification which facilitates its use by both control engineers and software
engineers. Secondly, the semantics of the language are formally defined [33], preventing errors due to misinterpretation.

Our choice for Astral was also based on pragmatic reasons. We already had experiences with its application (which also demonstrated its simplicity, important with respect to the fourth requirement listed above) and promising results concerning automatic verification of Astral specifications were reported in [33].

3.4 The Astral Specification Language

3.4.1 Introduction

This section introduces the specification language Astral, developed by Ghezzi and Kemmerer [32]. In subsection 3.4.2 an overview of the language is presented. We then discuss the use of Astral by describing a specification of the so-called Generalized Railroad Crossing system in section 3.4.3.

3.4.2 Language Overview

The goal of this section is to provide an overview of the Astral language which is needed to understand the example specification discussed next. An overview of the language is given and then the underlying computational model is discussed.

A major element of an Astral specification is a set of process type specifications. In Astral, a system is modeled as a collection of processes each of these being an instance of a certain process type. Each process type specification contains:

1. State variable declarations;
2. Transitions (to define state changes);
3. Some (default) process attributes;
4. Specification of the potential communication between different processes;
5. Specification of process properties.

Processes are state-based, that is, their state is defined by the evaluation of its state variables. State variables are declared in a part of the process type specification called the VARIABLE-clause. A variable can be assigned values only from a specific set of values, i.e., variables are typed.

The type of a variable can be a standard or a user-defined type. Standard types are the IDType (the set of process identifiers), Time (any real value
greater than or equal to zero denoting absolute time instants), Boolean, Integer and Real. User-defined types can be enumerated types, sub-types, structured (record) types and list-types.

Constraints on the initial values of the state variables of a process are specified in the INITIAL-clause of the process type specification. This clause, being a predicate over state variables, defines the constraints on the values of the state variables, on start up time of the system.

Possible state changes of a process are defined by a set of TRANSITION-clauses. Each transition clause consists of a unique transition name, a precondition, a postcondition and a fixed duration.

The precondition defines when the transition can be fired: 'In every state in which the precondition evaluates to true the transition is executable'. The postcondition defines the effect of the execution of the transition. Its meaning is defined as: 'When the transition is taken the postcondition holds in the resulting state'. For each transition the execution time (duration) must be specified. A transition duration is required to be greater than zero. Further rules of the execution of transitions will be discussed later.

By default, the process instances in the specification have the following attributes:

1. A state variable, named 'self' (of type IDType), containing a unique process identifier. In preconditions and postconditions it is possible to refer to a specific process instance through the use of the variable 'self'.

2. In each Astral specification, a single global variable called 'now' (of type Time) exists, visible to all processes in that specification. 'now' contains a representation of the current time. Time is dense, i.e., the variable 'now' can have any real value greater than zero. The value of 'now' initially equals 0 and increases monotonically.

3. The following pre-defined functions exist:
   - \texttt{Past(<var>,t)}, returns the value the variable \texttt{<var>} had at time instant \texttt{t} (\texttt{t < now}).
   - \texttt{Start(<transition>,k)}, returns the time instant in the past at which the \texttt{k}th previous \texttt{<transition>} execution was started. Because \texttt{k} refers to the \texttt{k}th previous start of a transition execution, each start of a transition execution implies that the value of \texttt{Start(<transition>,k)} changes.
   - \texttt{End(<transition>,k)}, returns the time instant in the past at which the \texttt{k}th previous \texttt{<transition>} execution was ended. Because \texttt{k} refers to the \texttt{k}th previous end of a transition execution, each time instant at which a transition execution ends implies that the value of \texttt{End(<transition>,k)} changes.
CHAPTER 3. REQUIREMENTS SPECIFICATION IN ASTRAL

In addition to the process type specifications an Astral specification also contains one part named the GLOBAL SPECIFICATION. A global specification consists of:

1. Definition of global constants;
2. Data type definitions;
3. Specification of global properties;
4. Declaration of process instances.

The only means of inter-process communication in Astral is by exported and imported state variables or attributes:

- A process that allows other processes to read values or attributes of its state variables and transitions explicitly lists the names of these entities in the EXPORT-clause of its process specification.
- In process specifications where the values of exported variables are read these variables are listed in the IMPORT-clause.
- The global variable ‘now’ is not declared explicitly in the variable-declaration of any process and is by default imported by each process in the specification.

In addition to these constructs, used in the specification of process behavior, Astral offers constructs for the specification of constraints on or properties of process behavior:

- Properties that should hold in every reachable state are expressed in the INARIANT-clause of a process type specification;
- Restrictions upon consecutive states are expressed in the CONSTRAINT-clause of the process specification;
- Timing properties or constraints on transition executions can be expressed in a SCHEDULE-clause. The SCHEDULE-requirements imposed on a process its transition executions must be feasible, i.e., it must be demonstrated that a possible execution sequence of TRANSITIONS exists for which this property holds.

Properties specified locally, i.e., within a process type specification, only apply to the process specification containing the property specification, and relate to instances of that process type.

Non-local constraints or properties that should hold for the total collection of Astral-processes are expressed in the INARIANT-, CONSTRAINT and SCHEDULE-clauses of the global specification.

The number of processes in an Astral specification is static and all process instances are defined in the PROCESS-clause of the global specification.

An Astral specification can be regarded as a program that runs on a
3.4. THE ASTRAL SPECIFICATION LANGUAGE

virtual machine. This virtual machine is characterized by the following attributes:

- **Maximal Parallelism.** It is assumed that every process runs on its own, dedicated processor;
- **Maximal Progress.** No processor is idle when a transition is able to execute, i.e., idling is not allowed when at least one pre-condition of the transitions of a process evaluates to TRUE. Transitions within a single process exclude each other, i.e., within a process at most one transition is executing at any time. In case more than one transition of a single process is enabled one is selected for execution nondeterministically;
- **Multicast Communication.** A change in value of exported variables is broadcast instantaneously at the time transition execution ends to every process that imports them. The same applies to start and end times of exported transitions.

Constraint and invariant properties should be provable from the Astral specification (i.e., they are already implied by the specification). However, a SCHEDULE-property can also impose additional constraints over transition executions.

3.4.3 Example: Generalized Railroad Crossing (GRC)

The use of the Astral specification language is illustrated by discussing fragments of the 'Generalized Railroad Crossing (GRC)'. The example is widely used in the literature to illustrate the use and compare the expressiveness of real-time specification languages [37]. The problem formulation from [37] is given in Figure 3.2. The goal is to construct a system specification in Astral.

First consider the global specification, it contains the declaration of process instantiations and constant- and type definitions:

GLOBAL SPECIFICATION grc

PROCESSES
  Tr : ARRAY[1..NRTR] OF Train;
  Gt : Gate;
  Gtc : Gatecontroller;

TYPE
  Nat1 : TYPEDEF i:INTEGER (i>=1);
  Trainid : TYPEDEF i:Nat1 (i<=NRTR);
  Trainpos = { Far, Near, In };
  Gatepos = { Open, Closed };
  Gcommand = { Open, Gclose };

CONSTANT
  NRTR = 2;

END GLOBAL SPECIFICATION
The system to be developed operates a gate at a railroad crossing. The railroad crossing $I$ lies in a region of interest $R$, i.e., $I \subseteq R$. A set of trains travel through $R$ on multiple tracks in both directions. A sensor system determines when each train enters and exits region $R$. To describe the system formally we define a gate function $g(t) \in [0, 90]$ where $g(t) = 0$ means the gate is down and $g(t) = 90$ means the gate is up. We define a set of occupancy intervals, where each occupancy interval is a time interval during which one or more trains are in $I$. The $i^{th}$ occupancy interval is represented as $\lambda_i = [\tau_i, \nu_i]$, where $\tau_i$ is the $i^{th}$ entry of a train into the crossing when no other train is in the crossing and $\nu_i$ is the first time since $\tau_i$ that no train is in the crossing (i.e., the train that entered at $\tau_i$ has exited as have any trains that entered the crossing after $\tau_i$).

Given two constants $\xi_1$ and $\xi_2$, $\xi_1 > 0, \xi_2 > 0$, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:

**Safety property:** $t \in \cup_i \lambda_i \Rightarrow g(t) = 0$ (The gate is down during all occupancy intervals.)

**Liveness Property:** $t \notin \cup_i [\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$ (The gate is up when no train is in or near the crossing)

Figure 3.2: GRC Problem Statement

The process Train has only one state variable which is used to represent the train position relative to the crossing. The following is an excerpt from the specification of process-type Train:

```plaintext
SPECIFICATION Train
EXPORT tpos;
CONSTANT
 T_en = 1; T_ei = 1; T_ev = 1;
 ROUND_TRIP = 10;
 MIN_DELAY = 3;
 MIN_OCC_DUR = 1;

VARIABLE
tpos : Trainpos;
```

The state of processes of type 'Train' at startup time is defined by the following INITIAL-clause:

```plaintext
INITIAL
tpos = Far;
```

The behavior of the process Train can be defined by three transitions. Execution of these transitions results in a train entering the region 'Near',
'In' or 'Far' respectively. The following transitions within the process Train can be executed:

\[
\text{TRANSITION EnterN} \quad T_{\text{en}}
\]
\[
\text{PRE} \quad \text{tpos} = \text{Far} \& \text{now} - \text{End(Leave,1)} \geq \text{ROUND\_TRIP}\text{T}
\]
\[
\text{POST} \quad \text{tpos} = \text{Near}
\]

\[
\text{TRANSITION EnterI} \quad T_{\text{ei}}
\]
\[
\text{PRE} \quad \text{tpos} = \text{Near} \& \text{now} - \text{Start(EnterN,1)} \geq \text{MIN\_DELAY}
\]
\[
\text{POST} \quad \text{tpos} = \text{In}
\]

\[
\text{TRANSITION Leave} \quad T_{\text{lv}}
\]
\[
\text{PRE} \quad \text{tpos} = \text{In} \& \text{now} - \text{Start(EnterI,1)} \geq \text{MIN\_OCC\_DUR}
\]
\[
\text{POST} \quad \text{tpos} = \text{Far}
\]

END Train;

The pre-conditions of the transitions, 'EnterN', 'EnterI' and 'Leave', express the conditions under which a train is allowed to enter another region. Consider the pre-condition of transition 'EnterN'. Its first conjunct defines that in order to execute this transition the value of the 'tpos' variable must be 'Far' and the second conjunct defines that at least ROUND\_TRIP time units must have been passed since the train left the crossing for the last time. The second conjunct guarantees a minimum separation time between two subsequent passings of the crossing by the same train. The duration of each of the transition is defined to be equal to T_{en}, T_{ei}, T_{lv} respectively.

Communication in Astral takes place through shared variables. By exporting the variable named 'tpos', other processes are offered the possibility to read the value of this variable. Processes in which the value of this variable is needed list the name of the variable in the IMPORT-clause of their process specification. For example, the process type named 'Gatecontroller' in the specification contains an import clause in which the variable 'tpos' is listed. The process 'Gatecontroller' contains a transition modeling the command to close the gate which is enabled as soon as the value of 'tpos' of one of the train process instances 'Tr[i]' equals the value 'Near'.

Critical for the correct behavior of the GRC system is that there is no train in the crossing while the gate is still open. In Astral such a constraint can be conveniently specified in the INVARIANT-clause of the process specification as follows:

\[
\text{INVARIANT}
\]
\[
\text{FORALL \ } \text{tid:Trainid}; (\text{Tr[tid].tpos=In } \Rightarrow \text{Gt.gpos=Closed})
\]
CHAPTER 3. REQUIREMENTS SPECIFICATION IN ASTRAL

As this safety property concerns two different processes from the Astral specification it is specified in the INVARIANT-clause of the global specification.

The liveness property is formulated in Astral as follows:

\[
\text{INVARIANT}
\begin{align*}
& \text{FORALL tid:Trainid; (Tr[tid].tpos <> In)} \\
& \quad \Rightarrow \\
& \quad \text{exists t:Time; tid:Trainid; j:Integer; (t=Tr[tid].End(Leave,j) \\
& \quad \quad \quad \quad \quad \quad \quad \text{now} <= t + xi_1)} \\
& \quad \quad \text{exists t:Time; tid:Trainid; j:Integer; (t=Tr[tid].End(EnterN,j) \&} \\
& \quad \quad \quad \quad \quad \quad \quad t >= \text{now} - xi_2)
\end{align*}
\]

More examples of Astral-specifications can be found in [20, 23, 32, 33].

3.5 Interfacing Controller and Software Development

This section presents the relation of Astral with the development of a control strategy in the CE domain. It discusses how specifications in Astral might be related to the block diagram modeling notation discussed in the previous chapter and how Astral could be applied in the development of embedded, real-time control software.

Once a controller has been developed we cannot expect to have it directly available in an Astral specification. In general, some kind of control system model, for example a block diagram, defining controller behavior will be available.

Therefore our initial goal is to construct a requirements specification in Astral given that the control algorithm(s) to be realized have already been defined by a set of block diagrams. As should be clear from previous discussions, the models underlying these two modeling notations are different and sometimes they even make conflicting assumptions.

It is thus not straightforward to derive an Astral specification from a block diagram model. This is inherent to situations in which different modeling notations are used together. In general, this problem is known as method integration [49], i.e., the combination of domain specific methods into an overall development method.

In our opinion, a systematic translation of block diagram specification(s) to an Astral specification is regarded as infeasible. We therefore propose the reformulation in Astral of the components of the block diagram specification
that are to be realized in software. Reformulation requires manual intervention because: (i) block diagrams only specify behavior of the control software in specific modes of operation, and (ii) the model underlying the block diagram notation assumes that operations are carried out simultaneously and instantaneously which conflicts with Astral where one is required to specify the duration of each transition and constraints are imposed on transition executions.

The usability of Astral will increase if possibilities for the analysis of specifications would exist. Analysis can help in the process of detection and correction of errors in the specification which in turn will have a positive effect on the quality of the resulting software. Current experience in the analysis of Astral specifications is limited. Furthermore, prototype tools for automated analysis of formal specifications have only recently become available. As analysis is important in the construction of a specification we have to consider analysis of Astral specifications also.

Although it might seem that extra development efforts are needed when Astral is used, this is not completely true. Experiences concerning the use of formal methods [43] show that in many cases the extra development efforts that have been put in the requirements specification work out positively because the efficiency of subsequent development phases is increased.

Our aim is to construct an Astral specification from a control system model defined in a block diagram. In the next section, we will discuss what is needed to perform the reformulation of block diagrams in Astral. Furthermore, we discuss what can be done in the software engineering domain, to facilitate the use of Astral as a requirements specification language for real-time control software.

3.6 Evaluation

The proposed approach based on the use of Astral for control software specification has been discussed in section 3.5. We think Astral offers sufficient capabilities that make it suitable for requirements specification of real-time software. Our major concern is how to enable an easy reformulation of block diagram models. An important issue in this respect is how to deal with the closed-world model represented by block diagram models developed in the control engineering domain.

Within the context of controller development and based on the contents of the previous two sections, the following issues concerning the applicability of Astral need to be considered:

1. A block diagram model represents a closed-system model in which both
controller behavior and the (often continuous) physical process being
controlled is modeled. How to deal with these (continuous) system-
components?

2. A block diagram model abstracts from certain aspects of the controller
behavior. In essence, this model represents an idealized real-world be-
havior like calculations taking zero time to execute and zero delay.
These abstractions are needed to keep the block diagram model (mathe-
ematically) analyzable. How should we deal with these abstractions
(and simplifying assumptions)?

3. A block diagram model is limited with respect to its expressiveness.
There are aspects of system behavior which are not modeled in a block
diagram. This could be, for example, behavior in exceptional situations
and application of different control strategies (each represented by a
single block diagram) in different situations. What is the best way to
elicitate these additional requirements from the control engineer?

The overall conclusion is that software requirements can only be partially
derived from a block diagram model. Furthermore, the abstractions (simpli-
fying assumptions) which have been made sometimes conflict with what is
actually realizable in software.

We therefore propose to reformulate the controller model, represented
by a block diagram, in Astral. The resulting Astral specification is then to
be completed (adding requirements concerning previously neglected aspects
of controller behavior) by the control engineer. This software requirements
specification phase is finished as soon as a complete software requirements
specification has been established.

However, reformulation is a difficult task and does require quite some
involvement of a control engineer. The question is thus what support can
be given to facilitate the process of constructing a software requirements
specification in Astral. In the following chapters, the following issues that
are important in this respect will be addressed:

1. Extending Astral such that it also becomes suited for the specification
   of continuous system components. This would allow one to study soft-
ware behavior in relation to the behavior of the environment which is
common practice in the CE domain. In that case, it implies that the
discrete model underlying Astral must be left and replaced by another
one;

2. Assessing applicability in practice through a case study concerning the
   specification of a non-trivial control system;
3. Investigating the extent to which software requirements specifications can be (automatically) analyzed. The analysis of the resulting Astral specifications is expected to facilitate detection and correction of errors in the requirements specification. Furthermore, analysis results demonstrate the correctness of a specification with respect to required (and safety-critical) properties of the specification;

4. Using tools to support the construction of a requirements specification of a control system.
Chapter 4

The H-Astral Specification Language

4.1 Introduction

In the previous chapter we chose to derive an (initial) Astral specification from an existing control system model (represented by a block diagram). This causes some difficulties. The major difficulty is the capability to model continuous as well as discrete phenomena with block diagrams. This puts additional requirements on the expressiveness of the Astral language. It implies that Astral must be extended to become a hybrid system specification language, i.e., capable of modeling systems which consist of continuous as well as discrete components.

The choice for a hybrid systems model offers possibilities for improving the current development process. The use of a hybrid system model enables the specification of closed systems, i.e., both the control software and the environmental (often continuous) processes are modeled. The behavior of the software can thus be studied in relation with the behavior of the environment [14]. In control engineering, the correctness of development decisions are in most cases justified by analyzing controller behavior in relation with its environment (the controlled physical process). Enabling the specification of both (discrete) control software and (continuous) environmental processes corresponds to current practice in the control engineering domain. This ability, resembling current practice in controller development, is therefore expected to improve the process of software requirements specification.

One major interest, from a software engineering point of view is to facilitate the development of a complete requirements specification. After a requirements specification has been constructed, another software engineer-
ing issue is the realization of the discrete components. In extended Astral, the sole means of communication between discrete and continuous system parts is by explicitly reading or writing shared variables. This is expected to facilitate the realization of the discrete control components in software which can in that case be developed independently from the continuous system components.

As will be discussed in this chapter, a suitable theory for hybrid system specification already exists. This offers a formal model for hybrid systems on which the semantics of extended Astral can be based. First the theory is introduced and then language constructs which have been added to Astral to enable the specification of continuous system components are discussed.

A useful semantic model has become available because in the past continuous extensions to well-known semantic models used for software specification have been developed. In such models, software behavior is modeled by sets of (infinite) state sequences. Extensions to this semantic model have already been developed that enable the specification of time-dependent and continuous system behavior. These extensions to the basic (discrete) model have resulted in modeling this kind of systems by sets of so called hybrid behaviors. Our goal is to demonstrate that this model can be used to define the semantics of extended Astral.

This chapter is structured as follows: A model (and theory) for the specification of hybrid systems is discussed first which can be used to define the semantics of the extended Astral specification language. In section 4.2, the semantic model adopted is discussed. Using this model, the entire system is modeled by a set of so called hybrid behaviors.

In section 4.3, an abstract machine for the specification of real-time control systems, the Hybrid Automaton (HA), is discussed.

In section 4.4, we develop a language for the specification of hybrid systems, Hybrid-Astral (H-Astral). H-Astral is based on Astral [32] augmented with language constructs for the specification of continuous time components. The semantics of the specification language are defined using the hybrid automaton introduced earlier.

Finally, we consider the use of tools for automatic analysis of hybrid system specifications. Experiences in using a tool called HyTech [40] for the verification of hybrid systems is discussed.

Throughout this chapter, we use the example of the inverted pendulum system which was introduced in section 2.2.2 to clarify the approach. A hybrid automaton specification of the inverted pendulum system is discussed in section 4.3. An H-Astral specification of the inverted pendulum then follows in section 4.4.

Experiences using HyTech in analysing the hybrid automaton specifica-
tion of the inverted pendulum system is then discussed in section 4.5. Such verification experiences provide insight in the capabilities and limitations of a (prototype) tool like HyTech in the automatic analysis of hybrid automaton specifications.

4.2 A Computational Model for Hybrid Systems

In this section, we define a computational model for hybrid systems. A hybrid system can be modeled through hybrid behaviors [4]. The set of all possible behaviors is also called a concrete semantic model of a (real-world) system. In the past few years, several concrete semantic models for hybrid systems have been developed. A comprehensive introduction can be found in [30, 34].

Informally, a hybrid behavior is a sequence whose elements can be either states or state trajectories. States appearing in the sequence represent observations (snapshots) of a system state at a particular time instant. The state trajectory defines the state of the system in a closed interval of $\mathbb{R}_0^+$ between two adjacent observations in the sequence. A number of definitions are given first that subsequently are used in defining hybrid behaviors more precisely.

**Definition 4.1** The state variable set, $V$, is defined as the union of two disjoint, finite sets: $V = V_d \cup V_a$ in which:

- $V_d$, called the set of discrete variables
- $V_a$, called the set of continuous (analog) variables

**Definition 4.2** A state, $\sigma$, is a function: $V \rightarrow \mathbb{R}$ that assigns a value to every state variable $v \in V$. The set of all possible states (valuations) is denoted by $\Sigma$.

For convenience, the value of a state variable, $v$ in state $\sigma$ is written as: $\sigma(v)$. Type-consistency is assumed, i.e., the valuation function $\sigma$ assigns to each variable a value of the set of allowed values, $T_v$. The restriction of $\sigma$ to the elements of a set of variables $V_r \subseteq V$ is denoted with $\sigma_{|V_r}$. To define discrete state changes, a relation $R$ over states (with restrictions) is used.

**Definition 4.3** Discrete state changes are defined by a relation $R$ over states: $R \subseteq \Sigma \times \Sigma$. Such that for each $(\sigma_1, \sigma_2) \in R$ the following holds:

$$\forall \sigma \in \Sigma : (\sigma, \sigma) \in R$$

When considering a pair $(\sigma_1, \sigma_2) \in R$, $\sigma_1$ is named the source state and $\sigma_2$ is named the target state.
The apostrophe when appended to a variable name indicates that it refers to the value of the variable in the source state, i.e., \( v' \) denotes the value of \( v \) in the source state of \( (\sigma_1, \sigma_2) \in R \). The apostrophe when operating on a set of state variables, \( V_s \), is defined as follows: \( V' = \{ v' \mid v \in V_s \} \).

State trajectories, i.e., definitions of system states during a certain time interval will be defined using piecewise continuous functions.

**Definition 4.4** A piecewise continuous function, \( f \), is a differentiable function of the time interval \([0, \delta]\) \((0, \delta \in \mathbb{R}_0^+)\) to \( \mathbb{R} \):

\[
f : [0, \delta] \rightarrow \mathbb{R}
\]

We assume \( f \in \mathcal{F} \), in which \( \mathcal{F} \) denotes the set of all differentiable functions over time intervals \([0, \delta]\).

In particular:

- \( f'(0) \) is defined to be equal to the right derivative with respect to time of \( f(0) \)
- \( f'((\delta)) \) is defined to be equal to the left derivative with respect to time of \( f(\delta) \).

The variable \( \dot{v} \), called a dotted variable, defines the rate of change of some variable \( v \in V \) with respect to time, i.e., \( \dot{v} = \frac{dv}{dt} \). The first derivatives of all discrete variables are assumed equal zero, \( \forall v_d \in V_d : \dot{v}_d = 0 \). In this chapter, often an implicit definition of piecewise continuous functions is given using dotted variables. The dot-operator when applied to a set of state variables, \( V_s \), is defined as follows: \( V' = \{ \dot{v} \mid v \in V_s \} \).

Time is modeled by a single global clock. A dense time model is adopted in which clock values are modeled by the set of non-negative reals \( \mathbb{R}_0^+ \).

The definitions given, allows to define hybrid state sequences (used to model real-world system behavior). Instantaneous state changes are defined by \( (\sigma_j, \sigma_k) \in R \) and state trajectories are defined by piecewise continuous functions \( f_i \) defined on time intervals \([0, \delta_i]\).

**Definition 4.5** A hybrid state sequence, \( HSS \), is a sequence:

\[
\sigma_0 \xrightarrow{S_0} \sigma_1 \xrightarrow{S_1} \sigma_2 \xrightarrow{S_2} \ldots
\]

in which \( S_i \) is either:

- An instantaneous state change, thus \( (\sigma_i, \sigma_{i+1}) \in R \), or
- A pair, \( (F_{V,i}, \delta_i) \), consisting of a set \( F_{V,i} \) of continuous functions, \( f_{v,i} \in \mathcal{F} \) and a positive real \( \delta_i \) (called delay). Each function \( f_{v,i} \) defines the value of the system variable \( v \) in the (closed) interval \([0, \delta_i]\).
The function \( f_{v,i} \) agrees with the value of \( v \) in start and end state of the interval: \( f_{v,i}(0) = \sigma_i(v) \) and \( f_{v,i}(\delta_i) = \sigma_{i+1}(v) \).

In case a HSS is finite and ends with \( s_m \), its length (denoted \( |HSS| \)) equals \( m \). In all other cases its length equals \( \infty \) (unbounded).

The time-instant associated with the observation \( \sigma_i \) is the sum of the delays until \( \sigma_i \), i.e.,

\[
\sum_{k=0}^{k=i-1} \delta_k
\]

(in which \( \delta_k = 0 \) when \( S_k \in R \)).

Values of state variables defined by \( \sigma_i \), at each time instant \( d \in [0, \delta_i] \) during the delay \( \delta_i \), is denoted \( \sigma_i(v, d) \).

Hybrid state sequences are used to model the behavior of a real-time control system. However, additional constraints will be put on these hybrid state sequences in order to rule out sequences that cannot represent valid behaviors of any (physical) real-time system.

**Definition 4.6** A real-time system CS is modeled by a set of hybrid behaviors \([CS]\). Every hybrid behavior \((H \in [CS])\) is a hybrid state sequence satisfying the following conditions:

- divergence property: the sum \( \sum_{i=0}^{H} \delta_i \) is unbounded (non-zenoess)
- finite variability: in any closed time interval \([t_1, t_2]\) there are only finitely many (discrete) state changes \((\sigma_j, \sigma_k) \in R\)

**Example: Inverted Pendulum**

![Figure 4.1: Model for the inverted pendulum control system](image)

As an example, consider the inverted pendulum introduced in section 2.2.2. The block diagram model, consisting of a discrete controller and a continuous system, is depicted in Figure 4.1. It will be shown how such a system can be modeled by hybrid behaviors.
First, the continuous transfer function is transformed in so-called state-space form. The transformation can be carried out using standard available techniques [29]. One such technique is described in appendix B. This results in the following equations modeling the movement of the pendulum:

\[
\begin{pmatrix}
\dot{x}_0 \\
\dot{x}_1
\end{pmatrix} =
\begin{pmatrix}
0 & 1 \\
g/l & 0
\end{pmatrix}
\begin{pmatrix}
x_0 \\
x_1
\end{pmatrix} +
\begin{pmatrix}
0 \\
1
\end{pmatrix} u
\]

\[
\Theta = \frac{-1}{Ml} x_0
\]

(4.1)

The (discrete) control function is defined in equation 2.8. From this equation, it follows that the control signal \(u_k\) (at time instant \(kT\)) can be calculated using the previously calculated value \((u_{k-1})\), and past input values to the controller \(e_k, e_{k-1}\) and \(e_{k-2}\) respectively. The state of this inverted pendulum control system can be modeled by the following set of state variables:

\[
V = V_a \cup V_d = \{x_0, x_1, \Theta\} \cup \{u_k, u_{k-1}, e_k, e_{k-1}, e_{k-2}\}.
\]

Furthermore, the presence of a constant disturbance \(D(t) = C_d\) and a sampling period equal to \(T_s\) is assumed. In this case the initial state of the system, \(\sigma_0\), can be defined as follows:

\[
\sigma_0(x_0, x_1, \Theta, u_k, u_{k-1}, e_k, e_{k-1}, e_{k-2})^T = (0, 0, 0, K_p * C_d, 0, C_d, 0, 0)^T
\]

A fragment of a possible hybrid behavior of the inverted pendulum control system is shown in Figure 4.2. The fragment shows two different states \((\sigma_0\) and \(\sigma_1\)) by also shows the value of the state variables in \(\sigma_0\) and \(\sigma_1\). State changes are indicated by arrows and the definition of the state change, either an instantaneous change or a trajectory, is listed above the arrow.

The system is modeled as a strictly alternating sequence of state trajectories with duration \(T_s\) and (instantaneous) changes of the control value \(u_k\) at time instants \(kT_s\). Even in this relatively simple example, in which only different but constant values for the input value \(D(t) = C_d\) were assumed, the set of hybrid behaviors describing all possible system behaviors is infinite.

A more abstract means to define such (possibly infinite) sets is needed. Therefore, the next subsection addresses the specification of a set of behaviors by an abstract machine, the hybrid automaton.

### 4.3 Hybrid Automata

Hybrid automata are generalized finite state machines for the specification of hybrid systems proposed by Alur and Henzinger [3].
4.3. HYBRID AUTOMATA

\[
\begin{align*}
\sigma_0(x_0) &= 0 \\
\sigma_0(x_1) &= 0 \\
\sigma_0(\Theta) &= 0 \\
\sigma_0(u_k) &= K_p \ast C_d \\
\sigma_0(u_{k-1}) &= 0 \\
\sigma_0(e_k) &= C_d \\
\sigma_0(e_{k-1}) &= 0 \\
\sigma_0(e_{k-2}) &= 0 \\
\end{align*}
\]

\[
\begin{pmatrix}
\dot{x}_0 = x_1 \\
\dot{x}_1 = \frac{9}{4}x_0 + \sigma_0(u_k)
\end{pmatrix}, T_s
\]

\[
\begin{align*}
\sigma_1(x_0) &= \sigma_0(x_0, T_s) \\
\sigma_1(x_1) &= \sigma_0(x_1, T_s) \\
\sigma_1(\Theta) &= \frac{\Theta}{M} \sigma_0(x_0, T_s) \\
\sigma_1(u_k) &= \sigma_0(u_k) \\
\sigma_1(u_{k-1}) &= \sigma_0(u_{k-1}) \\
\sigma_1(e_k) &= \sigma_0(e_k) \\
\sigma_1(e_{k-1}) &= \sigma_0(e_{k-1}) \\
\sigma_1(e_{k-2}) &= \sigma_0(e_{k-2}) \\
\end{align*}
\]

\[
\begin{pmatrix}
x_0 \\
x_1 \\
\Theta \\
u_k \\
u_{k-1} \\
e_k \\
e_{k-1} \\
e_{k-2}
\end{pmatrix},
\begin{pmatrix}
x_0 \\
x_1 \\
\Theta \\
u_k' \\
u_{k-1}' \\
e_k' \\
e_{k-1}' \\
e_{k-2}'
\end{pmatrix}
\]

\[
\begin{align*}
K_p \ast (C_d - \Theta)' - K_p \ast e_{k-1}' + \ldots u_k' - \Theta f
\end{align*}
\]

Figure 4.2: A hybrid behavior fragment of the inverted pendulum

In this section, we briefly discuss hybrid automata based on the definition as presented in [39], but with a different timing semantics. It is shown that hybrid automata are a suitable means for the specification of embedded control systems. The hybrid automaton, the composition of hybrid automata and the semantics of hybrid automata are defined. To illustrate the use of hybrid automata in specification, this section concludes with a simple example in which a specification of the inverted pendulum control system based on hybrid automata is given.

**Definition 4.7** An atomic predicate is defined by the following grammar:

- **atomic Predicate** ::= term relop term | true | false
- **relop** ::= \'<', '<=', '==', '>=' | '>'
- **term** ::= c | v | term arop term
- **arop** ::= '+' | '-' | '*'

with c a rational constant and
v a variable name (v ∈ V)
Definition 4.8 A predicate is a boolean combination of atomic predicates. A predicate is defined by the following grammar:

\[
\text{predicate ::= atomic\_predicate} \mid \\
\neg \text{atomic\_predicate} \mid \\
\text{predicate `\&` predicate} \mid \\
\text{predicate `\vee` predicate}
\]

When evaluating predicates the usual priorities of operators are assumed. Predicates define sets of states or regions. The state set defined by the predicate \( P \) (denoted \([P]\)) is defined as the set of all states, \( \sigma \in \Sigma \), in which the predicate \( P \) holds:

\[
[P] = \{ \sigma \in \Sigma \mid P[V := \sigma] \}
\]

\( P[V := \sigma] \) denotes the evaluation of the predicate \( P \) in which variables have been assigned values in accordance with a given state \( \sigma \). A subscripted predicate name \( P_{V^r} \) represents the predicate \( P \) which only reference variables out of the set of variables \( V^r \). The fact that a predicate, \( P \), holds in a particular state \( \sigma \) is denoted: \( \sigma \in [P] \).

Predicates defined over source and target states, like \( P_{V^\cup V'} \), defines regions which consist of tuples:

\[
[P_{V^\cup V'}] = \{ (\sigma_1, \sigma_2) \in \Sigma \times \Sigma \mid P[V := \sigma_1, V' := \sigma_2] \}
\]

A predicate \( P_{V^\cup V'} \) defines the region:

\[
[P_{V^\cup V'}] = \{ (\sigma_1, \sigma_2) \in \Sigma \times \Sigma \mid P[V := \sigma, V' := \sigma_2] \}
\]

Definition 4.9 A hybrid automaton is a tuple:

\[
HA = (\text{Var, Loc, Edge, Inv, Jump, Lab, Ev, Init})
\]

in which:

- \text{Var}, the set of state variables
- \text{Loc}, a finite set of locations \( \text{Loc} = \{l_0, \ldots, l_n\} \)
- \text{Edge}, a set of edges \( (l_i, l_j) \), \( \text{Edge} \subseteq \text{Loc} \times \text{Loc} \)
- \text{Inv}, a mapping of locations onto predicates \( \text{Inv} : \text{Loc} \rightarrow P_{V^\cup V^\prime} \)
  \( \text{Inv}(l_i) \) is called the invariant of location \( l_i \)
- \text{Jump}, a mapping of edges onto predicates \( \text{Jump} : \text{Edge} \rightarrow P_{V^\cup V^\prime} \)
- \text{Lab}, a set of event labels
  \( \)Every set of event labels has a null-element (\( e \in \text{Lab} \))
- \text{Ev}, a mapping of edges onto event labels \( \text{Ev} : \text{Edge} \rightarrow \text{Lab} \)
4.3. HYBRID AUTOMATA

- Init, a mapping of locations onto predicates: \( \text{Init} : \text{Loc} \rightarrow \mathcal{P}_{\text{Var}} \)
  \( \text{Init}(l_i) \) is used to define the admissible initial state(s) of the automaton

A hybrid automaton has a finite number of control locations. Conceptually, it is convenient to think of the state of the automaton having one additional variable whose value defines the current location of the automaton. At any time instant, control can reside in only one of the locations of the automaton. Location invariants are used for the specification of the rate of change of analog variables and of constraints on the value of analog variables.

The jump condition is a predicate over variables in source and target states. It defines the allowed changes in control locations of the automaton and defines the new value of discrete and analog state variables in the target state (state transition).

Associated with each location, \( l_i \in \text{Loc} \), is a predicate, called the invariant of \( l_i \). The invariant is used to define constraint(s) on the value of a variable while control of an automaton resides at \( l_i \).

The set of all possible execution sequences of the hybrid automaton (called hybrid traces, denoted \([HA]\) ) are all sequences:

\[
(l_0, \sigma_0) \rightarrow (l_1, \sigma_1) \rightarrow (l_2, \sigma_2) \rightarrow \ldots
\]

Control of the automaton starts at location, \( l_0 \) and state \( \sigma_0 \), such that \( \sigma_0 \in \llbracket \text{init}(l_0) \rrbracket \). Every step \( (l_i, \sigma_i) \rightarrow (l_{i+1}, \sigma_{i+1}) \) can be a time step or a transition step. During a time step control stays at the same location, i.e., \( l_i = l_{i+1} \) and the invariant associated with this location is required to hold during the delay of the time step. In case of a transition step, \( l_i \neq l_{i+1} \), this transition step \( (l_i, \sigma_i) \rightarrow (l_{i+1}, \sigma_{i+1}) \) denotes a change of control location in which the values of the state variables in source and target state are governed by the jump condition associated with the state change.

**Definition 4.10** The composition of hybrid automata

\[
HA_1 = (\text{Var}_1, \text{Loc}_1, \text{Edge}_1, \text{Inv}_1, \text{Jump}_1, \text{Lab}_1, \text{Ev}_1, \text{Init}_1)
\]

and

\[
HA_2 = (\text{Var}_2, \text{Loc}_2, \text{Edge}_2, \text{Inv}_2, \text{Jump}_2, \text{Lab}_2, \text{Ev}_2, \text{Init}_2)
\]

denoted \( HA_1 \parallel HA_2 \), is the hybrid automaton

\[
HA_c = (\text{Var}_c, \text{Loc}_c, \text{Edge}_c, \text{Inv}_c, \text{Jump}_c, \text{Lab}_c, \text{Ev}_c, \text{Init}_c)
\]

in which:

- \( \text{Var}_c = \text{Var}_1 \cup \text{Var}_2 \)
- \( \text{Loc}_c = \text{Loc}_1 \times \text{Loc}_2 \)
• \( \text{Edge}_c \subseteq \text{Loc}_c \times \text{Loc}_c \) such that \(((l_1^1, l_1^2), (l_2^1, l_2^2)) \in \text{Edge}_c\) if one of the following conditions holds:

(i) \((l_1^1, l_1^2) \in \text{Edge}_1 \land (Ev_1(l_1^1, l_1^2) = \epsilon \lor Ev_1(l_1^1, l_1^2) \not\in \text{Lab}_2) \land l_1^2 = l_2^2\)

(ii) \((l_1^1, l_1^2) \in \text{Edge}_2 \land (Ev_2(l_1^2, l_2^2) = \epsilon \lor Ev_2(l_1^2, l_2^2) \not\in \text{Lab}_1) \land l_1^1 = l_2^1\)

(iii) \((l_1^1, l_1^2) \in \text{Edge}_1 \land (l_1^1, l_1^2) \in \text{Edge}_2 \land Ev_1(l_1^1, l_1^2) = Ev_2(l_1^2, l_2^2) \land Ev_1(l_1^1, l_1^2) \neq \epsilon \land Ev_2(l_1^2, l_2^2) \neq \epsilon\)

• \(\text{Inv}_c : \text{Loc}_c \rightarrow P_{\text{Var}_c \cup \, \ast \, \text{Var}_c}\) defined as follows:

\[\text{Inv}_c(l_1, l_2) = \text{Inv}_1(l_1) \land \text{Inv}_2(l_2)\]

• \(\text{Jump}_c : \text{Edge}_c \rightarrow P_{\text{Var}_c \cup \, \ast \, \text{Var}_c}\) defined as follows:

(i) if \(l_1^1 \neq l_2^1 \land l_1^2 = l_2^2\) then \(\text{Jump}_c((l_1^1, l_1^2), (l_2^1, l_2^2)) = \text{Jump}_1(l_1, l_2)\)

(ii) if \(l_1^1 = l_2^1 \land l_1^2 \neq l_2^2\) then \(\text{Jump}_c((l_1^1, l_1^2), (l_2^1, l_2^2)) = \text{Jump}_2(l_1^1, l_2^2)\)

(iii) if \(l_1^1 \neq l_2^1 \land l_1^2 \neq l_2^2\) then \(\text{Jump}_c((l_1^1, l_1^2), (l_2^1, l_2^2)) = \text{Jump}_1(l_1, l_2) \land \text{Jump}_2(l_1^1, l_2^2)\)

• \(\text{Lab}_c = \text{Lab}_1 \cup \text{Lab}_2\)

• \(\text{Ev}_c : \text{Edge}_c \rightarrow \text{Lab}_c\) defined for each edge, \(((l_1^1, l_1^2), (l_2^1, l_2^2)) \in \text{Edge}_c\), as follows:

(i) if \((l_1^1, l_1^2) \in \text{Edge}_1 \land (Ev_1(l_1^1, l_1^2) = \epsilon \lor Ev_1(l_1^1, l_1^2) \not\in \text{Lab}_2) \land l_1^2 = l_2^2\)
then \(Ev_c((l_1^1, l_1^2), (l_2^1, l_2^2)) = Ev_1(l_1^1, l_1^2)\)

(ii) if \((l_1^2, l_2^2) \in \text{Edge}_2 \land (Ev_2(l_1^2, l_2^2) = \epsilon \lor Ev_2(l_1^2, l_2^2) \not\in \text{Lab}_1) \land l_1^1 = l_2^1\)
then \(Ev_c((l_1^1, l_1^2), (l_2^1, l_2^2)) = Ev_2(l_1^2, l_2^2)\)

(iii) if \((l_1^1, l_1^2) \in \text{Edge}_1 \land (l_1^1, l_1^2) \in \text{Edge}_2 \land Ev_1(l_1^1, l_1^2) = Ev_2(l_1^2, l_2^2) \land Ev_1(l_1^1, l_1^2) \neq \epsilon \land Ev_2(l_1^2, l_2^2) \neq \epsilon\)
then \(Ev_c((l_1^1, l_1^2), (l_2^1, l_2^2)) = Ev_1(l_1^1, l_1^2)\)

• \(\text{Init}_c : \text{Loc}_c \rightarrow P_{\text{Var}_c}\) defined as follows:

\[\text{Init}_c(l_1, l_2) = \text{Init}_1(l_1) \land \text{Init}_2(l_2)\]

The composition of two hybrid automata results in a hybrid automaton in which the execution of transitions of the two components is interleaved unless transitions have been assigned the same event label (except for the null-element, \(\epsilon\)). In that case, their execution is synchronized. The invariant of every location \((l_1, l_2)\) is the conjunction of the invariant of \(l_1\) of the first automaton and the invariant of \(l_2\) of the second automaton.

The semantics of hybrid automata can be defined using a labeled tran-
sition system in which every step is either a time-step or a discrete-step [2].
A labeled transition system \( LTS = (\Sigma, I, L, \xrightarrow{\delta}) \) is associated with the hybrid automaton, \( HA = (\text{Var}, \text{Loc}, \text{Edge}, \text{Inv}, \text{Jump}, \text{Lab}, \text{Ev}, \text{Init}) \) in which:

- \( \Sigma \) is the set of all possible valuations, \( \sigma \) of the variables \( v \in \text{Var} \) of the hybrid automaton
- \( I \) is the set of initial states: \( I = \{ \sigma \in \Sigma \mid \exists l \in \text{Loc} \ (\text{Init}(l)[V := \sigma]) \} \)
- \( L = \text{Lab} \cup \mathbb{R}_0^+ \), the union of the set of event labels of the hybrid automaton and the set of non-negative reals
- \( \xrightarrow{\delta} \), a transition step relation whose definition is given in Figure 4.3.

Define the predicate \( \text{Discrete}(l_1, \sigma_1, l_2, \sigma_2, \text{lab}) \) as:

\[
(l_1, l_2) \in \text{Edge}, \ \text{Ev}(l_1, l_2) = \text{lab}, \ (\sigma_1, \sigma_2) \in [\text{Jump}(l_1, l_2)], \ \exists \mu_1, \mu_2 : ((\sigma_1, \mu_1) \in [\text{Inv}(l_1)], \ (\sigma_2, \mu_2) \in [\text{Inv}(l_2)])
\]

Define the predicate \( \text{Time}(l, \sigma_1, \sigma_2, \delta) \) as:

\[
\forall v \in \text{Var}, \exists f_v \in \mathcal{F} : (\sigma_1(v) = f_v(0), \ \sigma_2(v) = f_v(\delta), \ \forall 0 \leq t \leq \delta, \ \exists (\sigma, \mu) \in [\text{Inv}(l)] : (\sigma(v) = f_v(t))
\]

\[
\frac{\text{Discrete}(l_1, \sigma_1, l_2, \sigma_2, \text{lab})}{(l_1, \sigma_1) \xrightarrow{\text{lab}} (l_2, \sigma_2)}
\]

\text{discrete step}

\[
\text{Time}(l, \sigma_1, \sigma_2, \delta) \land \\
\forall 0 \leq t < \delta, \ \exists l_4, \sigma_3, \sigma_4, \text{lab} : \\
(\text{Time}(l, \sigma_1, \sigma_3, t) \land \text{Discrete}(l, \sigma_3, l_4, \sigma_4, \text{lab}))
\]

\[
\frac{(l, \sigma_1) \xrightarrow{\delta} (l, \sigma_2)}
\]

\text{time step}

Figure 4.3: Labeled transition system for the hybrid automaton

The definition given excludes the passage of time in a location whenever one of the transitions out of this location is enabled (urgency semantics).
Scheduling transitions this way has been defined as a synchronous scheduling policy in [58].

Hybrid automata are executable [4]. Execution sequences of a hybrid automaton can be generated by taking time-steps and performing discrete transitions. This implies that simulation of specifications of this kind is possible.

Various decidability results have been established for various classes of hybrid automata [41]. They show that only in relatively simple cases verification (especially model checking) is feasible. This is illustrated in section 4.5 in which experiences in automated verification of an example hybrid automaton specification is reported. Given the current state of the theory, simulation seems the only technique to analyze realistically sized systems specified through hybrid automata.

**Example: Inverted Pendulum**

In the graphical representation of a hybrid automaton, locations are depicted as circles. The location name and location invariant are given inside the circle. Arrows are both used to represent the automaton edges and initial conditions of the automaton.

Edges are represented by arrows starting at the source location and pointing to the target location. The corresponding event label and jump conditions are beside the arrow. Arrows which do not start from a source location and only point to a location represent the initial conditions of the automaton. Arrows are labelled with the corresponding init condition of the location. If the init condition of a location equals false the corresponding arrow is omitted in the graphical representation.

As an example, consider the hybrid automaton specification in Figure 4.4 of the inverted pendulum control system. The specification, corresponding to the block diagram depicted in Figure 4.1, consists of two automata. One automaton models the inverted pendulum, consisting of a single location (called NORMAL) with invariant condition $\dot{x}_0 = x_1$ and $\dot{x}_1 = \frac{g}{L} x_0 + u_k$. The other automaton models the controller action that consist of periodically calculating a new control value $u_k$ based on the observed pendulum angle $\Theta = \frac{1}{M_1} x_0$. Furthermore the presence of a constant disturbance $D(t) = C_d$ is assumed.
4.4. THE SPECIFICATION LANGUAGE HYBRID-ASTRAL

\[ x_0 = 0 \land x_1 = 0 \]

**NORMAL**

\[ x_0 = x_1 \land \]

\[ ẋ_1 = \frac{1}{T} x_0 + u_k \]

Automaton Inverted Pendulum

\[ u_k = C_d \land u_{k-1} = 0 \land \cdots \]

**RUN**

\[ T = kT_s \land \]

\[ ẋ = \cdots \land \]

\[ u_k = \cdots \land \]

\[ u_{k-1} = \cdots \]

**sample**

\[ T = kT_s \land \]

\[ ẋ = \cdots \land \]

\[ u_k = \cdots \land \]

\[ u_{k-1} = \cdots \]

**CALC**

**ready**

\[ T = kT_s \]

Automaton Controller

Figure 4.4: Hybrid automaton for the inverted pendulum control system

### 4.4 The Specification Language Hybrid-Astral

#### 4.4.1 Motivation and Design Goals

Although hybrid automata are means for formal specification of a system, they lack a number characteristics to make them useful in the specification of large, complex systems. For example, consider the following characteristics not present in a formalism like hybrid automata:

- **Scope**, all variables of the hybrid automaton are global variables. The formalism itself offers no facilities to restrict the scope of a variable.

- **Constraint Specification**, hybrid automata can be used to define system behavior. In some cases, it is preferred to define additional restrictions over system behavior by constraints. A constraint puts additional restrictions on the execution sequences of a given hybrid automaton.

- **Structure**, a hybrid automaton defines the behavior of a system component. The interactions taking place between hybrid automata, i.e., the communication structure does not become directly apparent from a hybrid automaton specification.
• *Typing*, large systems often consist of a number of the same components. Specifications of this kind of systems clearly benefit from the possibility to define instances of a specific 'component type' in the specification.

In general, the attractiveness of a hybrid automaton is its simplicity. However, using hybrid automata directly for the specification of large complex systems is difficult. As indicated above, in the specification of hybrid automata higher-level constructs cannot be used which makes the specification of large, complex systems by humans difficult. Therefore a specification language at a more abstract level, offering higher-level constructs is desired.

In the following section such a specification language for hybrid systems (called Hybrid-Astral or shortly H-Astral) is defined. H-Astral is an extension of the real-time specification language Astral [32] discussed in section 3.4. Constructs have been added to the language Astral to allow specification of continuous systems.

The next subsection discusses the specification language H-Astral. In section 4.4.3 the semantics of H-Astral are defined by a translation of H-Astral specifications into hybrid automata.

### 4.4.2 Hybrid-Astral

Astral itself is not suited for the specification of continuous systems. To enable the specification of continuous components, additional language constructs must be defined. These *continuous extensions* are based on the transforms found in block diagrams. Astral, discussed in section 3.4, extended with the language constructs for the specification of continuous phenomena defined in this section, is called Hybrid-Astral(H-Astral). Continuous phenomena are defined in H-Astral by continuous processes.

The state of a continuous process is made up of a set of continuous state variables which are declared in the VARIABLE-clause of the continuous process specification. Like in discrete process specifications, constraints on initial values of state variables are specified in the INITIAL-clause.

Communication between a continuous process and other Astral processes is based upon the use of shared data. In the EXPORT-clause of continuous processes only names of continuous state variables can occur, meaning that their value can be read by other processes.

Other (discrete) processes can refer to such a variable, for example, in pre-conditions of transition specifications. Transition executions can thus be triggered by predicates becoming TRUE over the state of a continuous process.
4.4. THE SPECIFICATION LANGUAGE HYBRID-ASTRAL

The IMPORT clause of a continuous process specification lists the state variables and constants whose values can be read by the continuous process.

The values of the state variables of a continuous process are neither defined nor constrained by transitions. Instead their value is defined in a special part of a continuous process specification called the RELATION-part. The RELATION-part consists of a number of (continuous) time equations or transfer functions optionally guarded with (mutually exclusive) conditions. Each condition defines the situation(s) in which the equations associated with this condition define the values of state variables over time.

A basic assumption is that the variables imported by a continuous process are state variables from discrete processes and thus change only finitely often in every time interval.

Communication between continuous processes is disallowed. The reason for this is simplicity, since from this restriction it follows that interactions between continuous processes are disallowed. We think that such a restriction is reasonable as long as synthesis (i.e., actual development) of continuous time models is not a primary issue.

The specification of the syntax of the Hybrid-Astral language is provided in appendix A. Below, the specification of the inverted pendulum control system in H-Astral is discussed.

Example: Inverted Pendulum

As an example consider the following specification of the inverted pendulum control system (see block diagram of Figure 4.1) in Hybrid-Astral:

GLOBAL SPECIFICATION Inverted_Pendulum_Control_System
PROCESSES
   invpend : Inverted_Pendulum;
   ipcontrol : IPController;
TYPE
   Control_Val = REAL;
CONSTANT
   C_p = 5;
END GLOBAL SPECIFICATION

SPECIFICATION IPController
IMPORT Control_Val, theta_p, C_p;
EXPORT u_k;
CONSTANT
   T_c, T_s : Time;
   K_p, K_i, K_d : REAL;
VARIABLE
   u_k : Control_Val;
   u_k1 : Control_Val;
   e_k : Control_Val;
   e_k1 : Control_Val;
   e_k2 : Control_Val;
   prev : Time;
INITIAL
   u_k = K_p*C_p & u_k1 = 0 & e_k = C_p & e_k1 = 0 & e_k2 = 0;
TRANSITION Sample \[ T_c \]
PRE
\[ \text{now} = \text{prev} + T_s \]
POST
\[ \text{prev} = \text{now} \& \ u_k1 = u_k' \& \]
\[ u_k = u_k1' + K_p*(C_p - \theta_p - e_k1') + \]
\[ K_i*(\ldots) + K_d*(\ldots) \]
\[ e_k = C_p - \theta_p \& e_k1 = e_k' \& \]
\[ e_k2 = e_k1' \]
END IPController

CONTINUOUS SPECIFICATION Inverted_Pendulum
IMPORT u_k;
EXPORT theta_p;
VARIABLE
\[ \theta_p : \text{REAL}; \]
INITIAL
\[ \theta_p = 0.0; \]
RELATION
\[ \text{CASE TRUE} : \theta_p/u_k = -1/(Mls^2-Mg); \]
END RELATION;
END Inverted_Pendulum;

Several alternatives can be defined in the RELATION-part of the continuous process Inverted_Pendulum. This facility can be used to specify continuous process behavior under varying conditions. Conditions specified in one of the case-alternatives can only contain references to imported state variables or to state variables of the continuous process.

4.4.3 Semantics of Hybrid-Astral

In this section, a so called translational semantics [59] for H-Astral is defined. Instead of defining a new semantics for H-Astral, the hybrid automaton model is used and a mapping of H-Astral specifications on hybrid automata is defined.

In defining the mapping the following (simplifying) assumptions about H-Astral specifications are made: (i) No existential or universal quantification is used in PRE- and POST-conditions of transitions, (ii) references to variable values in the past (using the function past()) are not made, (iii) PRE- and POST-conditions only reference latest start and end time of a transition execution, and (iv) all state variables are of primitive type Real, Integer or Time.

The translation scheme for the translation of H-Astral specifications into hybrid automata consists of the following steps:

1. Mapping discrete processes of the specification on a set of hybrid automata;
2. Mapping continuous processes of the specification on a set of hybrid automata;
3. Mapping of schedule constraints on a set of hybrid automata;
4. Taking the composition of the automata which result from performing each of the previous steps.

To facilitate the definition of the mappings, an abstract representation of an H-Astral specification (by sets and functions) is introduced first. Then each of the steps in the order listed above will be discussed.

In the following, an H-Astral specification is represented by a number of sets. The elements of each set denote the components of a specification like for example processes, state variables and transitions. To represent an H-Astral specification the following sets are used:

\[
\begin{align*}
Dproc &= \{P_1, \ldots, P_n\} \\
Cproc &= \{CP_1, \ldots, CP_p\} \\
Proc &= Dproc \cup Cproc \\
Var &= \{v_1, \ldots, v_q\} \\
Trans &= \{T_1, \ldots, T_m\} \\
Mode &= \{M_1, \ldots, M_k\} \\
Pred &= \text{a set of predicates}
\end{align*}
\]

The set $Dproc$ represents discrete process instances from the H-Astral specification. Each process instance is assigned a unique identifier, $P_i \in Dproc$. Continuous process instances are represented by the set $Cproc$. Each process instance is assigned a unique identifier, $CP_j \in Cproc$. State variables are contained in a set called $Var$. State changes for discrete processes are described by the set of transitions, $Trans$. Behavior of continuous processes is based on a distinction in different situations which are called $Modes$. The predicates used in a specification (following definition 4.8) are contained in the set $Pred$.

The structure of a given H-Astral specification is defined by the following functions:

\[
\begin{align*}
\text{Svar} : & \quad \text{Proc} \rightarrow \mathcal{P}(\text{Var}) \\
\text{Trs} : & \quad Dproc \rightarrow \text{Trans-set} \\
\text{Init} : & \quad \text{Proc} \rightarrow \text{Pred}_{\text{Var}} \\
\text{Pre} : & \quad \text{Trans} \rightarrow \text{Pred}_{\text{Var}} \\
\text{Post} : & \quad \text{Trans} \rightarrow \text{Pred}_{\text{Var} \cup \text{Var'}} \\
\text{Dur} : & \quad \text{Trans} \rightarrow \mathbb{Q}^+ \\
\text{Sit} : & \quad Cproc \rightarrow \text{Mode-set} \\
\text{Cond} : & \quad \text{Mode} \rightarrow \text{Pred}_{\text{Var}} \\
\text{Flow} : & \quad \text{Mode} \rightarrow \text{Pred}_{\text{Var} \cup \text{Var}}
\end{align*}
\]
The set of state variables of a process instantiation, \( P_j \in Proc \) is given by \( Svar(P_j) \). The initial predicate of a process instance equals \( Init(P_j) \). The set of transitions of a discrete process \( P_i \in Dproc \) is defined by, \( Trs(P_i) = \{ T_{1,i}, \ldots , T_{n,i} \} \) in which \( n_i \) equals the cardinality of \( Trs(P_i) \). For a continuous process, \( CP_j \in Cproc \), different modes of operation are distinguished and defined by \( Sit(CP_j) = \{ M_{1,j}, \ldots , M_{p,j} \} \) (\( p_j \) equals the cardinality of \( Sit(CP_j) \)).

Preconditions and postconditions of transitions, \( T_{j,i} \in Trs(P_i) \) are denoted \( Pre(T_{j,i}) \) and \( Post(T_{j,i}) \) respectively. The predicate which is used to define a mode of the continuous process, \( M_{j,i} \in Sit(CP_j) \), is defined by \( Cond(M_{j,i}) \) and the predicate defining the behavior of the continuous variables is given by \( Flow(M_{j,i}) \).

The meaning of process identifiers, \( P_i \in Proc \), transition identifiers, \( T_{j,i} \in Trans \) and modes \( M_{j,i} \in Modes \) depends on the context in which they are being used. \( P_i \) is sometimes used as a process identifier as well as to denote the (unique) name of the process instance. In much the same way, \( T_{j,i} \) is used both as a transition identifier but also as a shorthand for the (unique) name of the transition. A mode, \( M_{j,i} \), is used as a mode identifier as well as a denotation of the unique name of a mode.

**Translation of Discrete Processes**

Each instance of a discrete process, \( P_i \in Dproc \), is mapped on the hybrid automaton:

\[
AHA_i = (Var_i, Loc_i, Edge_i, Inv_i, Jump_i, Lab_i, Ev_i, Init_i)
\]

The variables of the automaton are the state variables of the H-Astral specification of the discrete process (including imported state variables). This set of state variables of the automaton is then augmented with the following state variables:

- \( c.P_i \), an analog variable, which is used to register the time instant at which the execution of the transition currently executed by \( P_i \) has to end;
- for every transition \( T_j \in Trs(P_i) \), the state variables \( start.T_j \) and \( end.T_j \);
- an analog state variable 'now'

\[
Var_i = Svar(P_i) \cup \{ c.P_i \} \cup \cup_{T_j \in Trs(P_i)} \{ start.T_j, end.T_j \} \cup \{ now \}
\]
4.4. THE SPECIFICATION LANGUAGE HYBRID-ASTRAL

The set of locations of the hybrid automaton, \( AHA_i \), consists of a single location named \( \text{idle}.P_i \) and one location called \( \text{exec}.T_j \) expressing the execution for each transition \( T_j \in Trs(P_i) \).

\[
\text{Loc}_i = \bigcup_{T_j \in Trs(P_i)} \{ \text{exec}.T_j \} \cup \{ \text{idle}.P_i \}
\]

The set of edges of the hybrid automaton consists of an edge \( (\text{idle}.P_i, \text{exec}.T_j) \) (start of an Astral transition execution) and one edge \( (\text{exec}.T_j, \text{idle}.P_i) \) (end of transition execution) for each transition \( T_j \in Trs(P_i) \):

\[
\text{Edge}_i = \bigcup_{T_j \in Trs(P_i)} \{ (\text{idle}.P_i, \text{exec}.T_j) \} \cup \\
\bigcup_{T_j \in Trs(P_i)} \{ (\text{exec}.T_j, \text{idle}.P_i) \}
\]

The global clock, whose value is kept in the variable ‘now’ progresses monotonically in all locations of the automaton. The invariant associated with the \( \text{idle}.P_i \) location equals \( \text{now} = 1 \). Each \( \text{exec}.T_j \)-location of the automaton should be left once control has stayed in this location for the transition duration. For this purpose the variable \( c.P_i \) is used and the invariant of the location is defined as follows: \( \text{Inv}(\text{exec}.T_j) = (c.P_i = 1 \land \text{now} = 1) \)

\[
\text{Inv}_i = \{ \text{idle}.P_i \mapsto \text{now} = 1 \} \cup \\
\bigcup_{T_j \in Trs(P_i)} \{ \text{exec}.T_j \mapsto \text{now} = 1 \land c.P_i = 1 \}
\]

The following jump conditions are defined for every Edge(\( \text{idle}.P_i, \text{exec}.T_j \)) of the automaton:

\[[\text{Pre}(T_j)]' \land \text{start}.T_j = \text{now} \land c.P_i = 0 \]

and for every Edge(\( \text{exec}.T_j, \text{idle}.P_i \)):

\[\text{Post}(T_j) \land \text{end}.T_j = \text{now} \land c.P_i' = \text{Dur}(T_j) \]

\[
\text{Jump}_i = \bigcup_{T_j \in Trs(P_i)} \{ (\text{idle}.P_i, \text{exec}.T_j) \mapsto [\text{Pre}(T_j)]' \land \\
(\text{exec}.T_j, \text{idle}.P_i) \mapsto \text{Post}(T_j) \land \\
c.P_i = \text{Dur}(T_j) \land \text{end}.T_j = \text{now} \}
\]

Each edge of the automaton is labelled with a unique event label. The label \( \text{St}.T_j \) is assigned to edge \( (\text{idle}.P_i, \text{exec}.T_j) \), denoting start of transition \( T_j \)’s execution. The edge \( (\text{exec}.T_j, \text{idle}.P_i) \), denoting the end of a transition execution, is labelled with \( \text{E}.T_j \). The label-set of the automaton is defined as follows:
Figure 4.5: Automaton resulting from discrete processes translation

\[
Lab_i = \bigcup_{T_j \in \text{Tr}(P_i)} \{ St.T_j, E.T_j \}
\]

Each edge is either assigned the label \(St.T_j\) or \(E.T_j\):

\[
Ev_i = \bigcup_{T_j \in \text{Tr}(P_i)} \{ (idle.P_i, exec.T_j) \mapsto St.T_j, (exec.T_j, idle.P_i) \mapsto E.T_j \}
\]

\[
Init_i = \{ idle.P_i \mapsto Init(P_i) \land now = 0 \}
\cup \bigcup_{T_j \in \text{Tr}(P_i)} \{ exec.T_j \mapsto False \}
\]

An overview of the resulting hybrid automaton for process \(P_i\) is depicted in Figure 4.5.

**Translation of Continuous Processes**

A continuous process instance is mapped on a hybrid automaton. As with discrete processes, every continuous process instance of an H-Astral specification is uniquely identified as \(CP_j \in C\text{proc}\)

\[
AHA_j = (Var_j, Loc_j, Edge_j, Inv_j, Jump_j, Lab_j, Ev_j, Init_j)
\]

\[
Var_j = S\text{var}(CP_j)
\]

The number of locations of this automaton is equal to the the number of modes (case alternatives) in the RELATION-part of the continuous process specification

\[
Loc_j = \bigcup_{M_{k,j} \in S\text{u}(CP_j)} \{ M_{k,j} \}
\]
4.4. THE SPECIFICATION LANGUAGE HYBRID-ASTRAL

Essentially the \( n \) modes of process \( CP_j \) specify \( n \) different situations and each situation corresponds to a location of the automaton. A jump condition is specified from location \( M_{k,j} \) to every other location \( M_{l,j} \) with \( l \neq k \). The jump condition associated with the transition \((M_{k,j}, M_{l,j})\) equals condition \( Cond(M_{l,j}) \):

\[
\text{\text{Edge}}_j = \bigcup_{M_{k,j}, M_{l,j} \in \text{Modes}(CP_j) \land k \neq l} \{(M_{k,j}, M_{l,j})\} \\
\text{\text{Jump}}_j = \bigcup_{M_{k,j}, M_{l,j} \in \text{Modes}(CP_j) \land k \neq l} \{(M_{k,j}, M_{l,j}) \rightarrow [Cond(M_{l,j})]\} \\
\text{\text{Lab}}_j = \{\epsilon\} \\
\text{\text{Ev}}_j = \bigcup_{e \in \text{Edge}_j} \{e \mapsto \epsilon\} \\
\text{\text{Init}}_j = \bigcup_{M_{k,j} \in \text{Modes}(CP_j)} \{M_{k,j} \mapsto \text{Init}(P_j) \land Cond(M_{k,j})\}
\]

An important element of the proposed translation concerns the way equations together with the associated conditions are obtained from the reformulation of continuous parts (i.e., transfer functions) of block diagram models in H-Astral. This is addressed extensively in control engineering and the translation of transfer functions into time-dependent equations is discussed in appendix B. In the sequel, it is assumed that, in each mode from the Laplace-transform defining system behavior in this mode, equations of the following kind can be derived:

\[
\dot{x} = Ax + Bu \\
y = Cx
\]

The invariant of each location, denoted \( M_{k,j} \), equals the flow condition, \((Flow(M_{k,j}))\). \( Flow(M_{k,j}) \) then equals the conjunction of equation 4.2 and equation 4.3. The variables \( u, y \in \text{Var}_j \) are also defined in the block diagram but the transformation requires additional variables to be introduced. These additional variables represent the components of the vector \( x \) and must be added to the set of variables of the automaton. The hybrid automaton resulting from a translation of a continuous process \( C_p \) is depicted in Figure 4.6.

**Translation of SCHEDULE-constraints**

The previous sections showed how the behavioral part (i.e., defined by state, initial variable values and transitions) of an H-Astral specification has been mapped on hybrid automata. Another important issue, addressed in this section, are schedule constraints which are part of the SCHEDULE clause of a discrete process specification. A schedule constraint is defined as: the part of a SCHEDULE clause which is not implied by the behavioral part of the H-Astral specification.
Contrary to schedule properties, schedule constraints are additional requirements on timing behavior which must be satisfied by the implementation. In our opinion, a major task during design and implementation is to make sure that schedule constraints become schedule properties, i.e., properties satisfied by the resulting implementation of the system.

In H-Astral, a schedule constraint puts additional requirements on scheduling of transition executions by defining relations over start and end times of transition executions. The problem of finding a feasible schedule, i.e., one that satisfies the set of schedule constraints, is NP-complete [31]. This means that the behavioral specification part must be adapted manually and that subsequently it must be verified that the original schedule constraint has become a property of the adapted specification.

In this section we discuss one possible solution to realizing SCHEDULE-constraints based on the use of hybrid automata: Each SCHEDULE-constraint is transformed into a ‘separate’ SCHEDULE-automaton which is capable of synchronizing with the ‘start’ of a transition execution only when the SCHEDULE-constraint is satisfied.

Three possible SCHEDULE-constraints will be considered here, based on a classification of timing constraints proposed by Dasarathy [27]:

- **Maximum Timing Constraints,**
  i.e., no more than MAX time units may elapse between the start of two transition executions, which is defined in a SCHEDULE-constraint as follows:
4.5 Verification with HyTech

The previous sections have focused on the use of hybrid automata to model real-time embedded control systems. Recently a tool for the analysis of hybrid automaton specifications, called HyTech [40] has become publicly available. Although only a restricted class of hybrid automaton specifications can be verified by HyTech, it is currently the only verification tool for hybrid automata which is available.

The goal of this section is to provide insight in the usability of HyTech in automated verification of properties of hybrid automaton specifications. It discusses to what extent HyTech has shown useful to verify a hybrid au-
tomaton specification of the inverted pendulum control system. A number
of experiences concerning its use by others are reported in [26, 38].

Initially the H-Astral specification of the inverted pendulum system has
been transformed into a hybrid automaton specification following the trans-
formation discussed in the previous section. The goal was to evaluate the use
of HyTech in the verification of the resulting hybrid automaton specification.
It soon became apparent that in this case the resulting automaton specifica-
tion became too complex. This was mainly due to the following: (i) the
calculation of the control value was too complex to be handled by HyTech,
and (ii) HyTech can only perform verification on systems of which the first
time derivative of analog variables is constant or bounded by a fixed lower
and upper bounds.

To solve the first problem, several (alternative and simpler) controller
models have been considered. Such a model has been found in [29] and is
described by the following equations:

\[
\begin{pmatrix}
\dot{x}_0 \\
\dot{x}_1
\end{pmatrix} = \begin{pmatrix}
0 & 1 \\
\frac{g}{l} & 0
\end{pmatrix} \begin{pmatrix}
x_0 \\
x_1
\end{pmatrix} + \begin{pmatrix}
0 \\
-h_1 \frac{x_0} {l} - h_2 \frac{x_1} {l}
\end{pmatrix}
\] (4.4)

The angle (\(\Theta\)) of the pendulum is proportional to the value of \(x_0\). From the
equations, it becomes obvious that the control value (\(u = -\frac{h_1}{l} x_0 - \frac{h_2}{l} x_1\))
is proportional to the current angle (\(x_0\)) and the angular velocity \(\dot{x}_1\). The
values of \(h_1\) and \(h_2\) are constants and how their values can be calculated to
arrive at a stable control system is shown in [29].

Figure 4.7: Automaton resulting from translation of timing constraints
The second problem, the first derivatives not having constant bounds, is caused by the dependency of $\dot{x}_0$ and $\dot{x}_1$ on $x_0$ and $x_1$. This implies that $\dot{x}_0$ and $\dot{x}_1$ are not being bounded by some constant interval. Analysis of systems of this kind is not directly possible in HyTech. This problem can however be overcome by constructing an approximation of actual system behavior. In [42], a technique is described to construct an approximation which is also applicable in our case. This approximation of the system specification is conservative. Properties that hold for the approximation also hold for the original system specification.

Such an approximation is constructed using information which becomes apparent when the time system response is depicted in the $x_0x_1$-plane. Four different possible system responses of the inverted pendulum control system have been drawn in Figure 4.8.

![Figure 4.8: Responses of the inverted pendulum system](image)

Based on the information obtained from the system response, the $x_0x_1$-plane is partitioned. Figure 4.9 shows how such an approximation of system behavior is constructed. Using the minimum and maximum values of $x_0$ and $x_1$ in each partition and through substitution of these values in equation 4.4, constant upper and lower bounds on the value of $\dot{x}_0$ and $\dot{x}_1$ can be calculated.

Based on this partitioning, a hybrid automaton specification is constructed in which every location corresponds to a part of the $x_0x_1$ plane shown in Figure 4.9. The upper and lower bounds on the value of $x_0$ and $x_1$ are defined
Figure 4.9: Approximation of control system behavior

through an invariant of the location. Appropriate jump conditions are defined for each location corresponding to the system passing a partition boundary.

The resulting system specification, i.e., a conservative approximation of the behavior of the control system specified in equation 4.4, is useful in verifying a property with HyTech: \textit{when the system initially starts at } $0 \leq x_0 \leq 1 \land 0 \leq x_1 < 1$, \textit{then the system response stays within} $-3 \leq x_0 \leq 3 \land -1 \leq x_1 \leq 1$.

Using HyTech, it follows that the previous property holds for the hybrid automaton specification of the system.

This verification experiment indicates that possibilities exist for automated verification of simple hybrid automaton specifications. They also support the claim that simulation is currently the only feasible means to study behavior of more complex hybrid automaton specifications.
Chapter 5

Case Study: A Robot Control System

5.1 Introduction

In the previous chapters we considered the use of a formal specification language for requirements specification of real-time control software. This has resulted in the development of a requirements specification language called H-Astral. This chapter reports on the experiences from a case study aimed at the evaluation of the usability of H-Astral in the control engineering domain.

The subject of this case study is the robot control system described in [48]. That document provides a detailed description of a set of block diagram models. These block diagrams either model behavior of a different part of the robot or they model behavior of the same part but under the application of a different kind of control algorithm.

Based on this document, a specification of the system in H-Astral has been developed. The development of such a specification essentially consists of two phases. In the first phase, the already developed block diagram models are reformulated in H-Astral. The result of this step is an H-Astral specification of the control system which is incomplete because the required software behavior is only partially defined. In the second phase, all discrete components of the H-Astral specification are reconsidered and additional requirements are formulated to arrive at a complete specification of the software for the robot control system.

The block diagram model defining dynamic behavior of the robot and the control algorithms are described in section 5.2. Reformulation of the block diagram models for the robot control system in H-Astral is addressed in section 5.3. The second phase, in which additional requirements are added
to the specification, is discussed in section 5.4. Results from simulation experiments are then reported in section 5.5. In the final section, based on these case study experiences, the usability of H-Astral is evaluated.

Inherent in the semantic differences between H-Astral and the block diagram notation there exists a methodological problem concerning reformulation of block diagrams in H-Astral. In our case study, we have chosen to perform the reformulation of block diagrams in H-Astral by hand, because currently there is not sufficient knowledge or experience available to develop a generic transformation scheme.

5.2 The Robot Control System

5.2.1 Introduction

In this section, we discuss the robot model on which our case is based. Relations used to describe the robot movements are discussed in subsection 5.2.2. Algorithms to control robot movements are discussed in section 5.2.3. The graphical representation of this control system model through block diagrams is discussed in subsection 5.2.4. The contents of this section are based upon literature concerning the robot control system [5, 48, 60].

5.2.2 The Robot

The robot being modeled is a commercially available robot (Bosch TurboSCARA) with an arm composed of four separate parts (also called links), shown in Figure 5.1. The different links have been assigned a unique number as indicated in the figure. The front view of the robot (see Figure 5.1) shows a gripper connected to link number four. The position of the gripper, or more generally end-effector is called the end-effector location.

Positions are specified in a coordinate frame, the Cartesian Base Frame, shown in Figure 5.1. Since robot movements refer to the location of the endpoint of link 4 (i.e., the place where the gripper is located), this location is important. Rotational movement of links 1 and/or 2 results in a movement in the XY-plane. The orientation angle (also named joint angle) of link 1 \( (q_1) \) and 2 \( (q_2) \) can be measured through a position sensor.

Translational movement of link 3 results in a movement of the end-effector in the Z-direction. A position sensor is available to measure the position of link 3 \( (l_3) \) relative to a coordinate frame with its origin at the joint between link 2 and link 3. The fixed distance in Z-direction between this origin and the origin of the base coordinate frame equals \( d_1 \). The Z-coordinate of the
end-effector location equals \(d4\) relative to the end point of link 3. The end-effector location relative to the Cartesian base frame is then equal to \(d1 - l3\).

Link 4 (rotational) only causes a change in orientation (around Z-axis). A position sensor is connected to this link and used to measure the current orientation \(q4\).

### Kinematic Relationships

The relation between position and speed of the end-effector and the positions and speeds of the individual links is given in a number of rules, particular to the robot used. A systematic derivation of these rules, applicable to the Bosch TurboSCARA, is discussed in [5].

Given the joint angles of link 1, 2 and 4 (respectively \(q1, q2, q4\)) and the position of link 3 \(l3\) the following holds for the location of the end-effector:

\[
\begin{bmatrix}
  x \\
  y \\
  z \\
  \theta
\end{bmatrix} = \begin{bmatrix}
  l2 \cos(q1 + q2) + l1 \cos(q1) \\
  -l2 \sin(q1 + q2) - l1 \sin(q1) \\
  -d_1 + d_4 - l_3 \\
  -q_4
\end{bmatrix}
\]

(5.1)

The values of \(d1, d4, l1\) and \(l2\) are constants particular for the Bosch TurboSCARA robot, their values are given in table D.1 of [60].

In case the \((x, y)\)-coordinates of the end-effector are given, the corresponding joint-angles are defined by the following rules.

\[
\begin{aligned}
q_{1a} &= -\arccos\left(\frac{x^2 + y^2 - l_1^2 - l_2^2}{2dl_2}\right) \quad \vee \quad q_{1b} = -q_{1a} \\
q_{2a} &= -\beta + \gamma \quad \vee \quad q_{2b} = -q_{2a}
\end{aligned}
\]

(5.2)

In these rules, \(\beta\) and \(\gamma\) are defined as:
\[ \beta = \arctan 2 \left( \frac{y}{x} \right) \quad \wedge \quad \gamma = \arccos \left( \frac{l_1^2 + x^2 + y^2 - l_2^2}{2l_1 \sqrt{x^2 + y^2}} \right) \]

The use of these relations is subject to the following condition (the first two conjuncts reflect the mechanical limits imposed upon the first two links):

\[
\cos(-2(\text{rad})) \leq \frac{x^2 + y^2 - l_1^2 - l_2^2}{2l_1 l_2} \leq \cos(2(\text{rad})) \\
\wedge \\
-1.3(\text{rad}) \leq \beta - \gamma \leq 1.3(\text{rad}) \\
\wedge \\
-1 \leq \frac{x^2 + y^2 + l_1^2 - l_2^2}{2l_1 \sqrt{x^2 + y^2}} \leq 1
\]

The relation between angular joint velocities and Cartesian velocities of the end-effector location is given by the rules:

\[
\left( \begin{array}{c} \dot{x} \\ \dot{y} \end{array} \right) = J \left( \begin{array}{c} \dot{q}_1 \\ \dot{q}_2 \end{array} \right) \tag{5.3}
\]

where \( J \), the Jacobian for the Bosch robot, is given by:

\[
J = \left( \begin{array}{cccc} -l_2 \sin(q_1 + q_2) - l_1 \sin(q_1) & -l_2 \sin(q_1 + q_2) \\ -l_2 \cos(q_1 + q_2) - l_1 \cos(q_1) & -l_2 \cos(q_1 + q_2) \end{array} \right)
\]

For the calculation of joint velocities, when given the Cartesian velocities, the inverse relation is also needed:

\[
\left( \begin{array}{c} \dot{q}_1 \\ \dot{q}_2 \end{array} \right) = J^{-1} \left( \begin{array}{c} \dot{x} \\ \dot{y} \end{array} \right) \tag{5.4}
\]

\[
J^{-1} = \frac{1}{\det J} \left( \begin{array}{cccc} -l_2 \cos(q_1 + q_2) & l_2 \sin(q_1 + q_2) \\ l_2 \cos(q_1 + q_2) + l_1 \cos(q_1) & -l_2 \sin(q_1 + q_2) - l_1 \sin(q_1) \end{array} \right)
\]

The use of this inverse relation is subject to the condition that the determinant exists (i.e., \( q_2 \neq 0 \)).

The relations defined so far can be used to calculate the end-effector position and speed at a specific time instant for which the position and speed of the individual links is known.
5.2. THE ROBOT CONTROL SYSTEM

Dynamic Behavior

A model for the dynamic behavior of a system expresses the relation between the input signal that is applied and the resulting response, i.e., output signal of the system as a function of time. The robot model is made up of four independent parts, each expressing the dynamic behavior of a single link.

Input signals to the Bosch robot are voltages \( U \) applied to the motors driving a link. Output signals can be: (i) the position \( Z \) or orientation angle \( q \) of a link, or (ii) the velocity of a link \( V \) or \( \omega \), respectively. The relation between input and output signal is expressed by means of a differential equation. These differential equations can be transformed into transfer functions by applying Laplace transformation \([19]\).

Under the assumption that cross-coupling effects, i.e., the influence of one link on other links, can be neglected, dynamic behavior of link 1 and 2 can be described by the following transfer functions \([60]\):

\[
\frac{\omega_{m1}}{U_{m1}} = \frac{K_{m1}}{K_{m1}} \frac{(J_{L1}s^2 + (F_{eL1} + D_{h1})s + K_{h1})}{\xi_3 * s^3 + \xi_2 * s^2 + \xi_1 * s + \xi_0}
\] (5.5)

\[
\frac{\omega_{m2}}{U_{m2}} = \frac{K_{m2}N_2^2}{R_{m2}J_2s + R_{m2}F_{v2} + K_{m2}^2N_2^2}
\] (5.6)

These transfer functions express the relation between the input voltage, \( U \) (in volts) applied to the motor driving link 1 or 2 and the resulting angular velocity, \( \omega_i \) (in rad/s) of the link.

For the third link, two different models had to be developed. One model governs the dynamic behavior during free motion, i.e., when there is no contact between robot and environmental objects, the other one expresses the dynamic model of link 3 in contact situations:

\[
\frac{V}{U_{\text{in}}} = \frac{k_{DC} * (1 + s\tau_i)}{1 + \frac{K_m^2 + K_m k_p k_{ta} + B R + k_e L a^2}{a} s^2 + \frac{J R + B L}{a} s + \frac{J L}{a} \tau_i s^3}
\] (5.7)

\[
a = K_m k_p k_{ta} + k_e n^2 R \tau_i
\]

\[
k_{DC\text{vcontact}} = \frac{n K_m k_p}{a}
\]

\[
k_{DC\text{vnocontact}} = \frac{n}{k_{ta}}
\]
The model of the fourth link resembles the kind of model developed for link 3 [48] to the extent that there is no need to discuss the model for this link here. Appropriate values of the parameters for each individual link of the Bosch TurboSCARA robot can be found in tables 4.1 and 4.3 of [48] and table 2.9 of [60].

5.2.3 Control Algorithms

For different kinds of constraints, different algorithms have to be developed for the control of the movements of the robot links. E.g., movements of a painting robot require constant speed. A robot used for welding is required to perform movements at constant speed in the XY-plane while the contact force in Z-direction is kept constant. For the BOSCH robot, used to carry out assembly tasks (i.e., putting parts together), movements in the XY-plane are often to be performed with constant speed, while the contact force in the Z-direction must be kept constant.

The control algorithms developed are different with respect to their inputs, i.e., the controlled variable. When the controlled variable is the desired angle (for rotational moving axes), the algorithm is characterized by the term Joint-Position (JP). When the controlled variable is the position of a robot link, the algorithm is referred to as Cartesian Position (CP). Besides positions, speed (CV or JV) and force (CF) can also be used as controlled variables.

We refer to the term 'control mode' in case a specific control algorithm is applied. For the robot, different kinds of control algorithms have been developed, characterized by:

1. Cartesian Position (CP)
   The controlled variable is the position ($x$) in the base-coordinate frame.

2. Cartesian Velocity (CV)
   The controlled variable is the speed ($v$) relative to the base coordinate frame.

3. Joint Position (JP)
   The controlled variable equals the angle ($q_i$) of a link with reference to a fixed coordinate frame relative to link $i$.

4. Joint Velocity (JV)
   The controlled variable is the angular velocity ($\omega$) of a link.
5.2. THE ROBOT CONTROL SYSTEM

5. Force (CF)
   The controlled variable is the force \( f \) between the robot and environment with respect to the base coordinate frame.

6. Stopped (S)
   The link is supposed to stay at its current position.

It is, however, not the case that for each link six control algorithms have to be developed. E.g., no JV control algorithm has to be developed for link 1, while for link 3 no JP control algorithm has to be developed.

The control algorithm to be used is determined by the control mode that is specified in the description of the movement to be carried out (called a robot task). Additionally, there is a dependency defined between the control algorithm used in control of link 1 and the control algorithm used in control of link 2. When the control algorithm to be used for link 1 is denoted by \( CM_1 \) and the control algorithm to be used for link 2 is denoted by \( CM_2 \), the allowed combination of control algorithms for these two links is defined to be:

\[
(CM_1, CM_2) \in \{(CP, CP), (JV, JV), (CV, CV), (CV, CF), (CF, CV), (CF, CF)\}
\]

In the control engineering domain, first a distinct number of control modes has been distinguished. For each control mode, a single model of each link, its dynamic behavior has been established, i.e., influences of one link on another link are neglected. A number of control algorithms have been developed per link, one control algorithm for each control mode.

In order to understand the kind and nature of the control algorithms, the control algorithms developed for link 3 of the robot are discussed here in more detail. For the following control modes, a discrete control algorithm for link 3 has been developed:

- **CP**
  The (discrete) output signal of the Cartesian position control algorithm for link three \( u(n) \) is calculated periodically. Every \( t_{sp} \) time units the new value of the control signal is calculated. This value is proportional to the value of the error signal \( e(n) \), i.e., the difference between desired link position, \( z_{des}(n) \) and the actual (measured) position \( z_{act}(n) \). The simulation model of link 3 in CP-control mode is depicted in Figure 5.2.

The relation between input and output of the Cartesian position control algorithm of link three is described by the following equations:

\[
\begin{align*}
   e(n) & = z_{des}(n) - z_{act}(n) \\
   u(n) & = K_{pcp} \times e(n) \\
   K_{pcp} & = 200
\end{align*}
\]
Figure 5.2: Block diagram model (simulation) of link 3 in CP mode

- **CV**
The (discrete) input signal for the velocity control algorithm is the desired speed, \( v_d(n) \). The controller uses a cartesian position control algorithm and changes the desired position which is input to this algorithm periodically. Every \( t_{sp} \) time units the output signal is calculated. From the desired speed, the new desired position \( z_{des}(n) \) is derived. Based upon the value of this desired position and the measured actual position the value of the control signal is calculated in the same way as for the Cartesian position control algorithm.

The relation between input and output signal of the Cartesian velocity control algorithm of link 3 is defined by the following equations:

\[
\begin{align*}
  z_{des}(n) &= z_{des}(n-1) + v_{desired}(n) \cdot t_{sp} \\
  e(n) &= z_{des}(n) - z_{act}(n) \\
  u(n) &= K_{pcv} \cdot e(n) \\
  K_{pcv} &= 400
\end{align*}
\]  

- **CF**
The Force Control algorithm for link 3 has as its input signal the desired force in Z-direction \( f_{des}(n) \). The control signal is computed every \( t_{ef} \) time units. Through the use of a force sensor the force in Z-direction, \( f_{act}(n) \), can be measured.
5.2. THE ROBOTT CONTROL SYSTEM

The relation between input and output signal of the force control algorithm for link three is defined by the following equations:

\[
\begin{align*}
    e(n) &= f_{\text{des}}(n) - f_{\text{act}}(n) \\
    u(n) &= k_f \ast e(n) \\
    K_f &= 500
\end{align*}
\]  

(5.10)

The control algorithms for the other links are defined in a similar way. The dependency between link 1 and 2 cannot be neglected when certain movements in the XY-plane are to be carried out (see equations 5.1, 5.3). This requires careful synchronization between the separate controllers for link 1 and 2.

5.2.4 Block Diagram Representation

In the previous two subsections, models for the robot links (defined by equations 5.5, 5.6 and 5.7) and the developed control algorithms (or controllers) have been discussed. Usually the complete system model consists of both the controller and the system to be controlled. Such a model (in a specific control mode) is presented graphically as a block diagram. An example block diagram, which has been used for simulation, was already depicted in Figure 5.2.

The resulting system model for the robot control system consists of a collection of block diagrams. Each block diagram expresses the model of one link in a specific control mode.

As mentioned earlier (and apparent from equation 5.1), in some control modes, control of link 1 and link 2 needs to be synchronized in order to perform the required robot movement in the XY-plane. This is most apparent in CP control mode in which a movement of the robot relative to (Cartesian) XY-pane is specified. In that case, control of link 1 and 2 has to be synchronized despite the fact that both their dynamic behavior was modeled and a control algorithm was developed independently.

Therefore the two block diagrams modeling dynamic behavior of the first two links when in CP-control mode have been combined as shown in Figure 5.3. The block diagram depicted in Figure 5.3 models the behavior of link 1 and 2 in Cartesian Position control mode. This emphasizes the needed synchronization in control of these links in order to achieve the desired movement in the XY-plane. The switch in the figure denotes an A/D converter with sample period \( t_{sp} \).
The input signal of the control system as depicted in Figure 5.3, is a vector \((x_d, y_d)^T\) denoting the desired position of the end-effector in the XY-plane. Using equation 5.2 (abbreviated as \(L^{-1}(x)\)), the corresponding joint angles of link 1 and 2, \((q_{1,des}, q_{2,des})\), can be calculated.

Given the value of \((q_{1,des}, q_{2,des})\), the new values of the control signals, \((u_{m1}, u_{m2})^T\), are then calculated using the CP-control algorithm. The calculation of the output value is based on the difference between the desired joint positions \((q_{1,des}, q_{2,des})^T\) and the measured joint positions \((q_{1,act}, q_{2,act})^T\). Measuring joint positions, performing calculations and the output of the new control signals for both links must be performed synchronously in order to perform the required movement. The calculation is similar to the one defined in equation 5.8. Finally the control signal is applied to the motor driving link 1 and the motor driving link 2 of the robot whose behavior is defined by Laplace-transforms expressed in equation 5.5 and 5.6.

A model of the robot control system (consisting of a controller and the robot) is thus available as a set of block diagrams (see [48]). This set of block diagrams which result from carrying out development in the control engineering domain, constitute the model of the control system.

### 5.3 Reformulation in H-ASTRAL

In this section, we discuss the reformulation of the robot control system model in H-Astral and the resulting H-Astral specification. A detailed discussion of the different parts of the specification then follows in section 5.4.2 and further. The full H-Astral specification of the robot control system is given in appendix C.

For each link, every block diagram modeling its dynamic behavior in a particular control mode has been reformulated into H-Astral in the following way:

1. Define the processes
5.3. REFORMULATION IN H-ASTRAL

Basically, for each block in the block diagram, a separate H-Astral process specification is constructed. Each block labeled with a Laplace-transfer function is transformed into a CONTINUOUS PROCESS specification. All remaining blocks in the block diagram are transformed into a (discrete) PROCESS specification;

2. Define the interfaces
Each signal entering and leaving a block is labeled with a unique name. The process specifications contain an import clause listing the names of the signals entering the associated block in the block diagram. The name of each signal leaving the block is declared as a state-variable of the process and listed in the EXPORT clause of the process specification;

3. Specify the behavior of continuous processes
The transfer function (as expressed inside the block) of a CONTINUOUS PROCESS specifications is put in its RELATION-part;

4. Specify the behavior of discrete processes
To the discrete H-Astral processes a transition is added modeling the calculation of a new value of the output signal(s) based upon the current value of the input signal(s);

5. Specify required synchronization of transition executions
Execution of transitions in the processes that result from the transformation of the block diagram into H-Astral need to be synchronized. Therefore the PRE-conditions of these transitions must be adapted to reflect that execution of transitions is only enabled when the value of the input signals have been updated.

For each A/D-converter a single process results with a single transition. The PRE-condition of the transition constrains the periodic (equidistant) execution of the transition to once in each sampling period.

For each D/A-converter a single process results with a single transition. The PRE-condition of this transition assures that it is executed as soon as a new value of the control signal has been calculated.

As an example, the specification that results after the transformation of the switch (A/D conversion) and the ZOH-block (part of the block diagram in Figure 5.3) is given in Figure 5.6.

The calculation of the control signal (control_value) is done in the block labelled P (see Figure 5.3). Translation of this block results in a controller process called Ax1_Control, for robot link 1. Finally, the block labelled
SPECIFICATION Ax1_Sensor

IMPORT ax1_angle, T_sp;
EXPORT axis_angle, Read_Ax1angle;

CONSTANT
T_axis = 0.0001;

VARIABLE
axis_angle : REAL;

TRANSITION Read_Ax1angle T_axis
PRE
(Start(Read_Ax1angle,1)+T_sp=now)
| (now=0)
POST
axis_angle=ax1_angle
END Ax1_Sensor

SPECIFICATION Ax1_Actuator

IMPORT control_value;
EXPORT u_motor1;

CONSTANT
T_oax1 = 0.0001;

VARIABLE
u_motor1 : REAL;

INITIAL u_motor1=0;

TRANSITION Output T_oax1
PRE
u_motor1 <-> control_value
POST
u_motor1 = control_value
END Ax1_Actuator
5.4. THE H-ASTRAL SPECIFICATION

\( L^{-1}(x) \), representing the preprocessing of the control signal, is transformed into a process specification. The transformation of these two elements of the block diagram is illustrated in Figure 5.9.

Synchronization ensures that a new setpoint is not calculated before the new setpoint has been processed by the process Ax1_Control. This synchronization is achieved through the use of boolean state variables (e.g., ax1Setprequest).

Having transformed a block diagram for each link into an H-Astral specification, the block diagrams modeling the behavior in other control modes are incorporated as follows:

1. The discrete H-Astral-process modeling the controller is augmented with a transition modeling the calculation of the control signal in the control mode being incorporated. A state variable is introduced to keep track of the current control mode. The pre-conditions of all transitions are adapted to ensure that transitions are only enabled in the appropriate control mode and executed in the proper sequence.

2. Whenever a certain controller is based upon another type of controller, as was discussed in subsection 5.2.3 (see equation 5.9), a new process has been introduced. The calculation of the setpoint is modeled by a transition in the newly introduced process. The actual calculation of the control signal is performed by adding a transition to the H-Astral-process modeling the controller. Calculations like the one defined in equation 5.9 require careful synchronization, so boolean state variables have been added to achieve synchronization between the processes involved.

5.4 The H-Astral Specification

5.4.1 Architecture of the Specification

The resulting four level structure of the control system is given in Figure 5.10.

An interface process, the task-manager, has been added to the specification. The process models the interaction with the environment (depicted at level 3 in the figure). It receives commands to perform specific robot-movements (by a specification of the control mode and the set-point) from the environment. The task-manager communicates with the management processes at level 2 through exported state variables in order to signal control-mode changes to these lower level processes. Communicates of new values of the setpoints is performed using shared variables.
**SPECIFICATION** Ax1_Control

IMPORT Read_Ax1angle, ax1newsetpoint, ax1setpoint, axis_angle;
EXPORT control_value, ax1setprequest, ax1phi;

CONSTANT
T_r = 0.0002;
T_cp = 0.0002;

VARIABLE
ax1phi : REAL;
control_value : REAL;
ax1setprequest : BOOLEAN;

INITIAL
control_value = 0 &
NOT ax1setprequest

TRANSITION Read T_r
PRE
End(Read_Ax1angle,1)>Start(Read,1)
| now=0
POST
ax1phi=axis_angle &
ax1setprequest

TRANSITION Ctr_ValCartPosP T_cp
PRE
ax1newsetpoint
POST
control_value = P_Contr(K_pc, ax1setpoint, ax1phi,
control_value, error)
& NOT ax1setprequest
END Ax1_Control

**SPECIFICATION** Ax12_Manager

EXPORT ax1setpoint, ax1newsetpoint;
IMPORT x,y,ax1setprequest;

CONSTANT
T_cp = 0.0002;
T_c1 = 0.0002;

VARIABLE
ax1setpoint : REAL;
ax1newsetpoint : BOOLEAN;

TRANSITION New_Setp_CP T_cp
PRE
ax1setprequest
POST
ax1newsetpoint &
ax1setpoint = L_Inv(x,y)

TRANSITION Ctr_Setpax1 T_c1
PRE
NOT ax1setprequest &
ax1newsetpoint
POST
NOT ax1newsetpoint
END Ax12_Manager

Figure 5.7: Process Block

Figure 5.8: Inverse Kinematics

Figure 5.9: Reformulation of a control and an inverse kinematics block
5.4. THE H-ASTRAL SPECIFICATION

The processes at different levels of the H-Astral robot control specification are discussed in more detail in the following subsections.

5.4.2 Level 0 Processes

Level 0 processes in the H-Astral specification specify the behavior of the physical (continuous) processes that are controlled. The interfaces to these processes, i.e., processes modeling sensors and actuators, are also considered as belonging to this level of the specification.

Physical process specifications contain input-output relations that hold between imported and exported variables. The relations are expressed in terms of Laplace transforms, as already discussed in section 5.2.2.

The processes at this level result from the observation that a robot can be modeled by modeling the four links independently. In the H-Astral specification, the continuous behavior of each of the links is therefore specified in a separate process [48, 60]. This observation also holds for the sensors and actuators in the system that can be regarded as independent devices.

The process specification of an actuator contains a single transition that copies the controller value of the controller process (at level 1 in the figure) to the actuator variable. The pre-condition of this transition ensures that its execution is enabled whenever a new value of the actuator variable has been calculated in the controller process.

The specification of a sensor follows the same structure. It periodically
copies the new value of a (physical) process variable to the so-called sensor variable. This sensor variable is consulted by the (control) processes to monitor the state of the environment.

Notice the delay between the time instant, at which the (physical) process variable is observed, and the time instant at which it is assigned to the sensor variable (in the processes modeling a sensor). This is due to transition executions having non-null durations. The same holds for the value of the control signal. Some time elapses between the time instant at which a newly calculated control value becomes available and the time instant at which the actuator variable is assigned this value.

### 5.4.3 Level 1 Processes

Processes that belong to this level of the specification periodically calculate a new value of the control signal and issue this value to the actuators in the system. For each link of the robot, a separate controller process (at level 1 of the specification) models the control of this link.

Control of links 1 and 2 is sufficiently complex to allow further discussion. Therefore the specification of the process Ax1_Control (named A1_C in Figure 5.10) is discussed here (the specification of Ax2_Control is almost identical to Ax1_Control). The full specification of this process is incorporated in appendix C.

In Ax1_Control, the initial control mode is 'Stopped'. The imported variable ax1desmode has as its value the desired mode for the movement of this link. The imported variables ax1consmet and tmalarm indicate whether or not the state of the system allows a safe control mode switch. Execution of the transition Change_Ctrmode causes the new control mode to be assigned to the variable ax1mode.

The transitions of process Ax1_Control, in any of the possible control modes are shown in Figure 5.11. The boxes in the figure denote the state(s) in which a certain transition is enabled for execution and from which it becomes apparent that there is a specific ordering on transition executions. The preconditions of the transitions contain terms to ensure that control of link 1 is properly synchronized with the other robot links. This synchronization is the main task of the management processes that are discussed next.

### 5.4.4 Level 2 Processes

The processes at this level assure proper synchronization in control of individual links as modeled by the processes at level 1. The level 2 processes also
take care of establishing suitable setpoints used by the level 1 control processes in calculation of control values. Processes of this kind can be regarded as managing the actual controllers and are therefore called management processes.

Axis12_Manager

The required synchronization of the movements of link 1 and link 2 of the robot is performed by the process Axis12_Manager (named A12_man in figure 5.10). Main responsibilities of the process Axis12_Manager are the following:

(i) Commanding mode switches to the level 1 control processes for link 1 and 2.
(ii) Synchronization of the control of movements of link 1 and link 2.
(iii) Calculation of setpoints that are used in the calculation of control values by the level 1 processes.

A state variable manmode is declared in the process Axis12_Manager. The value of this state variable indicates the control mode of both control processes for link 1 and link 2.

The behavior of the process Ax12_Manager is discussed for each of the possible values of the state variable manmode.

1. manmode = Joint-Position: Setpoints for the level 1 control processes Ax1_Control and Ax2_Control are generated simultaneously. This results in a proper synchronization of movements of link 1 and 2.

The changes in setpoints for the lower level control processes are limited to a certain maximum value. Therefore the value of setpoints used by the processes Ax1_Control and Ax2_Control is changed by the process
Ax12_Manager in such a way that its value does not change with a rate that is higher than the allowed maximum.

2. manmode = Cartesian Position: In this case the desired position of the end-effector is given as coordinates \((x, y)\) relative to the Cartesian base frame. Before the actual movement to the desired position is started, it is required to verify whether or not the required movement is possible given the current robot position. Due to mechanical limitations it is sometimes necessary to change the position of the first two links relative to each other before the movement towards the desired \((x, y)\)-position can be made.

As an example consider the two different robot configurations shown in Figure 5.1. Although the relative position of the first two links is different, the resulting \((x, y)\)-position of the end-effector is the same in both cases. Changing from one robot configuration to another is called a 'flip'. A flip movement only changes the orientation of the two links but the location of the end-effector before and after a flip movement is the same. Before actual control of robot movement towards the desired \((x, y)\)-position is started, a flip movement is made when necessary.

During the actual movement towards the desired \((x, y)\)-position, setpoints are generated simultaneously for both control processes A1_C and A2_C, which achieves the required synchronization. The joint positions used as setpoints in level 1 control processes are derived from the desired \((x, y)\) position of the end-effector through the use of equation 5.2 (expressing the inverse kinematic relationship). In this case the inverse kinematics Cartesian position controller as presented in Figure 4.19 of [48], is used.

3. manmode = Cartesian Velocity: In this control mode the process Ax12_Manager calculates periodically the desired positions \((x_{des}, y_{des})\) corresponding to the desired speed \((v_x, v_y)\) in the Cartesian Base Frame. The controller in this mode, in which the inverse Jacobian is used, is also called the Inverse Jacobian based Cartesian position controller, as depicted in Figure 4.20 of [48].

4. manmode = Force: This mode of control can be divided in three different sub-modes of control:

- In the first sub-mode, both links are controlled in force mode.
- In the second sub-mode, movement in the direction of the x-axis direction is controlled in Force-mode while movements in y-axis direction are velocity controlled.
- In the last sub-mode, movement in the direction of the x-axis is velocity controlled and in y-direction it is force controlled.

A transition is incorporated in the process specification of \texttt{Ax12.Manager} that is always executed when a new robot movement has to be started. This transition, called \texttt{New.Task}, is enabled only when the previous movement of the robot has been accomplished. The behavior of the process \texttt{Ax12.Manager}, after the transition \texttt{New.Task} has been executed and the value of the state variable \texttt{manmode} equals \texttt{JointPosition}, is shown in Figure 5.12. The behavior in other modes except for mode \texttt{CartPos} is comparable to the mode \texttt{JointPos}. The main difference between these modes lies in the calculation of new setpoints.

![Figure 5.12: Behavior in JointPos mode](image)

The required system behavior in which, after the transition \texttt{New.Task} has been executed and a \texttt{flip} should be made is graphically depicted in Figure 5.13. After the flip has been made, the process behaves as depicted in Figure 5.12.

![Figure 5.13: Behavior in mode Flip](image)
Management of control of link 3 and 4

The processes, managing the control of the link 3 and 4, are similar to the specification of process Axis12.Manager that was discussed in the previous subsection. The specification differs with respect to the allowed control modes and the calculation of setpoints but is far less complicated than the process Ax12.Manager.

5.4.5 Task Management

The task management process is responsible for acceptance of so-called robot jobs from the environment. These jobs contain descriptions of the required robot movements and the associated control modes. A robot job also contains a predicate over the controlled variables (like position and speed). The time instant at which the predicate becomes true can be regarded as a signal indicating the completion of this job. At this time instant the task management process signals the end of task execution to the management processes at level 2 of the specification. Besides signaling the end of a robot job, these predicates also indicate whether or not mode switches are allowed.

When an Alarm-signal is received from the environment the (exported) variable tm_alarm is assigned the value TRUE. This results in stopping robot movements immediately.

5.5 Simulation

As indicated in section 4.3 hybrid automata are executable. This implies that simulation of system behavior defined by a H-Astral specification is possible. In the course of constructing the robot control system specification, several simulation runs have been carried out. A simple simulator for H-Astral specification has been built and used to simulate parts of the specification of the robot control system.

Simulation results can be used in demonstrating that some property holds for a (part of a) possible system behavior. Unfortunately, this does not imply that the property is valid for all possible system behaviors. However, simulation is attractive for two reasons: (i) it can be automated and is therefore a relatively cheap analysis technique, and (ii) it can lead to the detection of errors in the robot control system specification which would, without simulation, remain undiscovered.

To illustrate the possibilities (and limitations) of simulation of H-Astral specifications, one experiment will be discussed here. It concerns the part of the robot control system which is concerned with controlling behavior
of link 3. It concerns the processes A3_man, A3_Ctrl and Link 3 (see Figure 5.10).

Consider the following robot job which describes a movement for link 3 of the robot. Initially, control mode for link 3 is 'Stopped' and the link resides at the origin (position = 0). It is then to be moved to position −0.01 (m) and during this movement the control mode for link 3 is CP (Cartesian Position). As the link reaches this position, control mode should be switched from CP to CV (Cartesian velocity). Link 3 is then to be moved at constant speed (+0.1 m/s) to the position 0.0 (m). When link 3 is at its origin again, control mode has to be switched to 'Stopped'.

We incorporated a specification of this robot job in the process A3_man and performed a simulation of the resulting specification. The results of the simulation experiment are shown in Figure 5.14

![Figure 5.14: Response of link 3](image)

Other simulation results concerning the robot are reported in [48]. The simulation results reported there have been compared with our simulation results. The comparison revealed that our simulation results are the composition of the two separate simulation responses of link 3 described in [48], the first in CP control mode and the other in CV control mode.

An important difference is in the way they were obtained. Here they resulted from a simulation of a model of a discrete controller. The H-Astral language has been applied in this case study to show its applicability in defining software requirements. In the example given here, these requirements concern the way in which a robot job is to be carried out and how
control mode switches are to be carried out.

From a control engineering point of view, there is one peculiar aspect in the way the control mode switch is modeled. At the time of occurrence of such a mode switch, all the state variables of the system are set equal to zero. The position, at the moment the mode switch occurs, is then simply obtained by taking a superposition of this position and system response when started from its zero (origin) position. This response, being a composition of two separate system responses, follows from the definition of H-Astral semantics by the translation to hybrid automata which was defined in section 4.4.3.

5.6 Evaluation

In this chapter, we discussed the use of H-Astral in the specification of a robot-control system. Below, we evaluate:

(i) Reformulation of the block diagram specifications;
(ii) Simplifications made in the course of building a physical model of the robot, but which have become explicit in the H-Astral specification;
(iii) Software requirements which have been additionally specified.

The different control algorithms for the robot have been considered. Essentially, each block diagram models behavior of one part of the robot control system in one specific control mode. Combining this set of block diagram specifications in the H-Astral specification showed to be fairly straightforward. We also discussed the necessity to define required synchronization in controlling the movement of different links of the robot. Specification of these synchronizations turned out not to be a problem in H-Astral.

Some steps, for example, those in combining the specification of different control modes in a single process, cannot be performed automatically. That the reformulation process is not fully amenable to automation is felt as a major drawback.

In the beginning of this chapter, the development of a robot model has been discussed. Not directly apparent from the block diagram specifications, some important requirements only became clear when the simplifications made in the development of a physical process model were considered. An example of this concerns the position of links being (mechanically) constrained and certain movements which can only be made when a flip is made first. Although important, these simplifying assumptions are not always defined in the block diagram specification.

In the course of building a software requirements specification, not only the required calculations need to be defined but also the situations in which
they may or may not be performed. This concerns not only the conditions on the use of certain equations, which in the robot case are useful only when position is within a range determined by certain mechanical limits. Another example are singularities in the inverse of the Jacobian matrix.

However, what becomes most apparent from the H-Astral specification is its ability to specify additional and important software requirements. How a robot job is to be executed, error handling (alarm) and the ability to define required timing behavior under these varying circumstances is also important. From this perspective, we think H-Astral has something to offer which would be hard to incorporate (and certainly be less natural) and to define in a block diagram specification of the control system.
Chapter 6

Prototyping Astral Specifications

6.1 Introduction

The previous chapters elaborated on the use of H-Astral as a vehicle for software requirements specification. An H-Astral specification models behavior of the real-time system to be developed.

Implementation aspects have been identified in section 1.3 as being of main importance when the use of a formal specification in the software development process is considered. In software development, main interest is in the realization of the discrete components of an H-Astral specification. Attention is therefore restricted to (discrete) Astral specifications in this chapter.

In this chapter prototyping of Astral specifications is addressed. The Ada95 [7] programming language is used as an implementation language. As will be shown, large parts of the prototype can be generated from an Astral specification automatically.

Prototyping is regarded as a first step in the development of an implementation which implements a given Astral specification. The following definition of prototyping as given in [64] is used here: "the creation of a working model of a software system to clarify user requirements".

A prototype can provide valuable feedback about various aspects of an Astral specification. Our main interest is in the use of the executable prototype, derived from an Astral specification, to demonstrate whether or not the required timing behavior is realizable.

The prototyping approach has been applied in a case study concerning the realization of a train control system for a toy railroad. This experimental
facility is available in our laboratory together with a programming support environment called RACE [10]. The train control system has been used as a test bed to obtain practical experience with, and to demonstrate advantages and limitations of the prototyping approach.

An advantage of the proposed prototyping approach is that once a sufficiently detailed Astral specification (or parts thereof) is available, deriving a prototype is easy and straightforward. However, a major drawback associated with the developed approach is related to the structure of the resulting Ada95 implementation. Pre-runtime timing analysis of the prototype is difficult.

In this chapter we consider the use of hybrid automata in modeling and pre-runtime timing analysis of a prototype in the presence of a limited number of processors. Especially the case in which more than one Astral process is mapped on the same physical processor and the maximal parallelism assumption no longer holds is of interest. This design issue is sometimes also called the implicit concurrency vs. explicit concurrency issue [68].

As a basis for the solution to this timing analysis problem we take the automaton specification which results after the H-Astral to automaton translation defined in section 4.4.3 has been carried out. As will be shown, this automaton specification can be extended to model implementation aspects like worst-case execution time of operations (implementing transitions), the number of processors available and scheduling policy applied. Such an extended automaton specification provides the means to perform pre-runtime timing analysis.

Section 6.2 provides a short introduction to the Ada95 programming language. It highlights a number of language constructs while main emphasis is put on those constructs that are essential for understanding the Astral to Ada95 mapping. This mapping, which is used in generating a prototype from a Astral specification, is then presented in section 6.3. Practical experience, obtained in prototyping an Astral specification of a control system specification for a toy-railroad is discussed in section 6.3.2. In section 6.4 timing analysis of prototype-implementations is addressed and we discuss to what extent we have been able to use the hybrid-automaton formalism in analyzing the timing behavior of the Ada95-prototype.

6.2 The Ada95 Programming Language

In this subsection a number of primitives of the Ada95 language is discussed that are important for understanding the Astral to Ada95 mapping treated in the next section. For a comprehensive overview of the Ada95 programming
6.2. THE ADA95 PROGRAMMING LANGUAGE

language the reader is referred to, e.g., Barnes [7] or the language reference manual (LRM) [45].

The following Ada95 primitives are discussed:

1. Tasking (a task is the unit of parallelism of the Ada-language)
2. Protected object (data for which synchronized access is guaranteed) and associated operations (possibly with barriers).

Concurrency in Ada95 is expressed using the task construct. A task is part of a program equipped with its own thread of control. From a logical viewpoint, all tasks in an Ada95 program are assumed to execute in parallel. The task construct is a built-in construct of the language and its semantics have been defined in the LRM.

Consider the following excerpt from an Ada95 program to clarify the task concept in the Ada-language. This example shows the specification of an example task named ‘T1’ and its body.

```ada
task T1;
task body T1 is
    Have_To_Stop : Boolean := False;
begind
    while not Have_To_Stop loop
        delay(2*Period);
        Adjust_Train1;
    end loop;
end T1;
```

Ada95 offers two mechanisms for inter-task communication. Communication can be performed through a synchronization mechanism called rendezvous or by means of shared data through the protected object. Only the protected object is discussed here because the rendezvous mechanism is not used in the Astral to Ada95 mapping.

Synchronized access to shared data in Ada95 can be achieved through the use of a monitor-like construct, the protected object. A protected object encapsulates a data structure together with associated operations. Execution of these operations is guaranteed mutually exclusive. Entries, one form of operation for protected objects, can be guarded by conditions (called barriers). A task calling an entry is blocked until the barrier condition is satisfied. Protected objects with barriers can be used to implement signal/wait mechanisms.

Consider the following declaration of a protected object together with the associated operations:

```ada
protected Tr_001_State is

    entry Wait_For_Trans_Enabled(Tid : out Tr_001_Transidtype;
        Cstate : out Tr_001_Staterec);
```
entry Claim_Request(Address: out Unsigned_8;
   Item: out Item_Range);

entry Sync_Request;
entry Claim_Granted;
procedure Signal_Sync;
procedure Update_Occupation(Ocdata: in Toccup);
procedure Update_Local_State(Newstate: in Tr_001_Staterec);

private
   Tr_001_Data: Tr_001_Staterec;
end Tr_001_State;

protected body Tr_001_State is

   entry Wait_For_Trans_Enabled(Tid: out Tr_001_Transidtype;
                                 Cstate: out Tr_001_Staterec) when
   Tr_001_Data.Mode = Claim1 and Tr_001_Data.Cl_Grant is
   begin
      Tid := Firstclaimed;
      Cstate := Tr_001_Data;
   end Wait_For_Trans_Enabled;

       ............

end Tr_001_State;

The protected object Tr_001_State consists of a specification and a body, similar to the task construct discussed earlier. The specification is divided in two parts. In the first part the operations (which can be entries or procedures) on the encapsulated data are listed. The second part of the specification (after the keyword private) contains the declaration of the encapsulated data.

The body of the protected object contains the bodies of the operations. In the example above the body of the entry Wait_For_Trans_Enabled is shown. A call to an entry operation is blocked until the barrier condition (listed after the keyword when) evaluates to True and the operation can be executed only when no other operation of the protected object is executed at the same time.

6.3 Mapping Astral Constructs onto Ada95

6.3.1 The Astral to Ada95 Mapping

The Ada95 language, and especially the constructs discussed in the previous section, will be used to define a mapping of Astral to Ada95 constructs. However, inherent to the differences between the Astral and Ada95 languages is the impossibility to define a mapping that is complete, i.e., not every construct in Astral can be directly implemented by (a sequence of) Ada95 statements.
6.3. MAPPING ASTRAL CONSTRUCTS ONTO ADA95

An example of this are schedule constraints defined in the schedule-clause of a discrete process specification. The schedule constraints are used in the definition of a requirement which should be satisfied by the implementation. In general, assuring that such requirements are satisfied by the resulting implementation is a major design task which cannot be automated.

On the other hand implementing the behavioral part, i.e., state and transitions, of discrete process specifications in Ada95 seems well possible. Although this might seem a simple task at first sight a mapping of behavioral Astral constructs to Ada95 has to be developed with some care in order to avoid unwanted and anomalous behavior of the resulting implementation.

Ada95 is thus used as a high-level language to implement an executable, prototype implementation of an Astral specification. Before going into the details first an overview of the main issues which are to be dealt with in mapping Astral to Ada95 are presented:

(i) *Implementation of process-state*, each process has its own state which consists of a set of state variables;
(ii) *Time*, all process can obtain the current time by reading a global variable called ‘now’;
(iii) *Enabling conditions*, as soon as a transition of an idling process becomes enabled its execution should start immediately;
(iv) *Atomicity*, state updates are atomic and instantaneous;
(v) *Timing Behavior*, start and end time of transition executions (i.e., timing behavior) are precisely defined;
(vi) *Communication*, processes communicate by means of shared data and all state updates are immediately broadcast to all processes importing the state variables affected by the update.
(vii) *Parallelism*, transitions of different processes are executed in parallel (implicit concurrency assumption).

Each of these issues must be dealt with in the Astral-to-Ada95 mapping. Additional complexity follows from the inter-relationships between these issues. As an example of such an interrelationship, note that the start of a transition execution not only requires some kind of signaling mechanism to be implemented. The fact that it is started must also be atomically broadcast to all other processes.

Furthermore, although Ada95 offers a number of useful and high-level constructs it also has its limitations. One such limitation concerns the barrier-conditions of entry-operations in a protected object. In a barrier condition references to variables not being part of the private data of the object itself are not allowed [45].
How a prototype can be derived from the specification, is discussed by addressing each issue listed above in turn. Of each issue, not only its mapping to Ada95 constructs will be defined but also potential problems concerning its implementation in Ada95 and how these have been avoided will be discussed.

First consider the state of a process which consists of (i) the set of state variables declared in its variable clause, (ii) start- and end-times of transition executions, (iii) imported state variables and process attributes. The state of a process is implemented by a set of program variables. However, note the need for some kind of signaling mechanism to signal the event of a transition becoming enabled for execution. Entries of protected objects can be used to implement signal mechanisms which motivated the choice of using a protected object for implementing the state of a process. As an example, consider the fragment from an Astral specification and its proposed implementation in Ada95:

State of an Astral process

```
SPECIFICATION Tr_001
...

VARIABLE
  mode       : Tcontrol;
  cblock     : Blocks;
  nblock     : Blocks;
  claimreq   : BOOLEAN;
  speed      : Train_speed;
...
```

Implementation in Ada95

```
type Tr_001_Data is
  record
    Mode       : Tcontrol;
    Cblock     : Blocks;
    Nblock     : Blocks;
    Claimreq   : Boolean;
    Speed      : Train_Speed;
    ...       : Time;
    Start_St   : Time_List;
    Start_Up   : Time_List;
    End_St     : Time_List;
  end record;

protected Tr_001_State is
  private
    Procstate  : Tr_001_Data;
  end Tr_001_State;
```

Signaling the event of a transition becoming enabled can now be implemented by entries of the protected object. However, note that within a barrier it is impossible to reference variables other than those being part of the protected object itself. As noticed earlier, within a precondition of an Astral transition references are not limited to the state variables of the process itself. A precondition can also contain references to:

(i) Imported state-variables of other processes;
(ii) The current time, (references to the variable 'now');
(iii) start and end times of (imported) transitions.
For now, it is assumed that all this information is available within the data-part of the protected object. In that case, all data variables being referenced in a PRE-condition of a transition are contained in the protected object. Signaling a transition becoming enabled can thus be implemented by entry-operations.

The state update, required to be atomic can then be implemented by other update-procedures of the protected object. These can be used to perform the state update as defined by the post-condition of a transition. The update is atomic because synchronized access (execution of the operations is mutually exclusive) to the data of a protected object in Ada95 is guaranteed.

Thus, within the protected object body, a transition execution of an Astral process is implemented by one entry (enabling) and one procedure (implementing the state update). The following shows an example:

```plaintext
SPECIFICATION Tr_001
...

TRANSITION Startroute T_st
PRE
  mode = Stopped
POST
  <some_postcondition>
...

TRANSITION Updateposition T_up
PRE
  mode=Running &
  NOT occ_data[cblock] &
  occ_data[nblock]
POST
  <some_postcondition>
...

protected body Tr_001_State is
entry Wait_For_Trans_Enabled(
  Tid:out Tr_001_Transidtype;
  Cstate:out Tr_001_Data)
  when ...
or
  Procstate.Mode=Stopped
or
  (Procstate.Mode=Running and
  not Procstate.Occ_Data(Cblock) and
  Procstate.Occ_Data(Nblock))
or
  ...
  is
begin
  ...
end;

entry Startroute_End(
  Nstate:in Tr_001_Data) when
  Procstate.Now =
  Last(Procstate.End_St) is
  ...

entry Updateposition_End(
  Nstate:in Tr_001_Data) when
  Procstate.Now =
  Last(Procstate.End_Up) is
  ...
end Tr_001_State;
```

Timing of transition executions (i.e., timing behavior) is precisely defined by Astral's computational model. A transition execution, of an idling process, must be started at the moment its precondition becomes True. The entry-operation of a protected object implements the start of a transition execution. Thus, the prototype should therefore always be ready, i.e., waiting for each
of these entry operations such that the entry operation is carried out as soon
as its condition is satisfied.

When an entry operation is carried out it first performs a state update to
register the start of a transition execution. It then returns a copy of current
values of all state variables (i.e., the state at start time of the transition
execution). Then the operation, implementing the transition is executed
which operates on this copy of the state at start time of the transition. The
resulting state, after this operation has been executed, is required to satisfy
the postcondition of the transition being executed. Finally, at the time the
transition execution ends the actual state update (including updating the
end time of a transition execution) should be performed. This can be done
by calling the update-entry of the protected object.

For an Astral process instance, a task is created which first calls the
entry \texttt{Wait\_For\_Trans\_Enabled()}. The task is blocked until a transition is
enabled. Upon return from \texttt{Wait\_For\_Trans\_Enabled()}, the task then calls
the operation corresponding to the transition to be executed. As soon as this
operation has finished the update \texttt{<transitionname>\_End} entry of the tran-
sition is called. Then the task again executes a \texttt{Wait\_For\_Trans\_Enabled()}.  

Notice that, although the operation associated with the execution of a
transition can be executed at an arbitrary time between start and end time
of the transition, the state update is executed at the end time of the cor-
responding transition. The caller of this update entry is blocked by the
guard until the value of \texttt{`now'} (which is present within the protected ob-
ject) equals the time-instant at which the transition ends. The operation
\texttt{updateposition\_end()} is therefore executed at the time the transition ends
execution.

The solution to incorporate the value of the current time within the pro-
tected object necessitates the implementation of a separate clock-task. This
task is responsible for performing clock updates.

Until now, only the mapping of a single Astral process on Ada95 has
been discussed. But how is dealt with multiple process instantiations? In
order to signal the event of a transition becoming enabled a choice was made
to incorporate a representation of the current time (represented by the vari-
able \texttt{`now'}, start and end times of transition executions and imported state
variables within a single protected record.

Implementing the state of each process by a separate protected object
would necessitate replication of data. Replication of data, however, would
make it difficult to guarantee consistency. One serious problem, for example,
would be to implement the atomic update. For these reasons it has been
decided to implement the state of all process instances in the specification by
a single protected object. Because transition executions within a process do
not overlap, each process can be mapped on a separate Ada-task as illustrated above.

<table>
<thead>
<tr>
<th>The Astral-to-Ada95 mapping in short:</th>
</tr>
</thead>
<tbody>
<tr>
<td>State of all processes</td>
</tr>
<tr>
<td>(including process attributes)</td>
</tr>
<tr>
<td>Time ('now')</td>
</tr>
<tr>
<td>Start transition</td>
</tr>
<tr>
<td>End transition</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Transition execution</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Astral process instance</td>
</tr>
</tbody>
</table>

Figure 6.1: The Astral to Ada95 mapping

An overview of the mapping is presented in Figure 6.1. Although intuitively appealing a mapping which is performed along the guidelines given here should not be seen as the ultimate way in which Astral specifications should be implemented. The process structure of the Astral specification has been followed in defining the mapping, causing a clear correspondence between the discrete parts of the Astral specification and the Ada95 implementation. However, one should note that any implementation as long as it satisfies the required behavior defined in the Astral specification will be a correct implementation.

### 6.3.2 Prototyping Experiences

In our laboratory one of the experimental facilities available is a control system for a toy-railroad (see Figure 6.2) and an associated programming support environment, called RACE [10]. This forms a convenient test bed and through the use of the capabilities of RACE a prototype can be constructed relatively quickly. Based on a detailed Astral specification, an Ada95 prototype of the train control system has been constructed following the mapping as discussed in the previous section. The outcome of the practical experiments is discussed and evaluated in this section.

A major task in the control of a train is the prevention of train collisions. A large part of the train control system specification is concerned with the synchronization of train accesses to different parts of the track. If trains are
always in different parts (or blocks) of the physical track, collisions will never happen.

In the train control specification, a train waiting to enter a specific block first requests permission to enter this part. If permission is granted within \texttt{PERM\_DELAY} (a constant in the specification), the train can run straight into the next requested block. However, if permission is not granted within \texttt{PERM\_DELAY} time units after issuing the request the train should be halted.

In case a train controller has issued a request and awaits permission the value of its state variable ‘mode’ equals ‘Rclaiming’. This request is issued each time the train enters a new block and this is modeled by execution of a transition named ‘Updateposition’ in the specification of the train controller. The Astral specification of this process contains the following ‘Timeout’ transition:

\begin{verbatim}
TRANSITION Timeout T_to
PRE
  mode=Rclaiming & now >= End(Updateposition,1) + PERM\_DELAY
POST
  speed=HALT & mode=Claiming
\end{verbatim}

From the pre-condition it follows that this transition will be enabled when \texttt{PERM\_DELAY} time units have elapsed since the time instant a new block has been entered and a request for permission to enter the next block was issued. As can be derived from the postcondition the execution of the transition ‘Timeout’ leads to the train being halted. It will now be discussed how the prototype has been used to verify whether a train indeed halts in time in case a request is not serviced within \texttt{PERM\_DELAY} time units.
The support environment is depicted in Figure 6.3. It consists of three computing systems connected in a network. One computing system, called the logger, can be used to monitor start and end times of transition executions or to visualize a train’s behavior graphically (see Figure 6.4). Another computer, the physical controller, is responsible for managing the hardware interface. Finally on the third computer the prototype of the train controller process is executed.

By sending log-messages at the start and end of transition executions it is possible to monitor the transition executions of the prototype implementation. RACE offers the possibility of writing the contents of log messages to the console such that transition executions can be monitored at run-time (or analyzed afterwards by writing them to a file also).

An experiment has been performed, in which the train has been forced to stop due to a request for a block not being granted in time. The behavior of the prototype implementation has been monitored and written to a log-file. The following part of the log-file shows the log-messages generated by the train controller in this case:

```
======= Logger Start =====
```
From inspection of the generated log-messages it could be concluded and as was confirmed by observing actual train behavior, the train was stopped in time. The request not being granted within PERM_DELAY time units (0.4 s) in this case forced execution of the timeout-transition as required in the specification.

Prototyping of the train controller specification also revealed a serious error in the control algorithm. The results of the experiments showed that trains did not always halt in time. In some case a train halted in a block to which it was not given permission to enter. This was shown to be caused by the delay between the time the halt-command was issued and the train actually being stopped. This delay was considerable and could certainly not be neglected.

The train control algorithm has been adapted, in order for a train to be started it must have been given permission to enter both the next and second next block on its route. In case a running train passes a block boundary, a new request for entering a new second next block is issued. If this permission is not given within PERM_DELAY time-units the train is halted and will halts on the block currently occupied or on the next block. In this case it is guaranteed that it never halts on a block to which it has not been granted access.

### 6.3.3 Discussion

An Astral to Ada95 mapping for prototyping has been developed and subsequently used in prototyping a train-control system specification. The mapping has been shown useful and no major difficulties in developing an implementation from the specification were encountered.

The prototype implementation offers possibilities to test system behavior by playing specific scenarios, as was done in the experiment described earlier, in which a specific scenario has been ‘played’ forcing execution of the ‘Timeout’ transition. Behavior in that case was analyzed at (or after) run-time by
inspecting log messages. A second benefit of prototyping has been shown to be that it demonstrated errors being present in the specification. Through prototyping, the error was discovered and corrected.

There is however a major problem concerning the level of detail of the Astral specification which is necessary in order for the mapping to be feasible. The topic of how such a concrete design specification is to be developed starting from an abstract requirements specification has not been addressed.

Another disadvantage which becomes apparent is the analysis of timing behavior of the resulting prototype-implementation. The current approach only allows the study and analysis of timing behavior at (or after) run-time. The timing behavior of the resulting Ada95 implementation was only possible by analyzing logging data which contained timing figures concerning the behavior of the Ada95-implementation.

This is regarded as insufficient and a better means for analyzing the behavior pre-run time must be developed. Because of this we will elaborate this subject further in the following section.

6.4 Timing Analysis

6.4.1 Introduction

Prototyping Astral specifications can provide some evidence concerning the realizability of the required behavior. Practical experience revealed a major disadvantage, pre-runtime timing analysis of the prototype is difficult. It is desirable to have an analysis technique available which would allow us to perform pre-runtime timing analysis.

![Diagram of transition duration and operation]

Figure 6.5: Scheduling operations implementing a transition

In the prototype, a transition execution is implemented by some operation associated with the transition. There are two constraints on the execution of this operation. A release constraint, ensuring that the operation can be
executed only after the associated transition has started execution. Furthermore, it has a strict deadline: the result of the operation is required to be available before the atomic state-update is carried out (remember that it must be executed at the end time of the transition execution, see also Figure 6.5).

Clearly, there exists a scheduling problem: the required operations must be scheduled in such a way that they are always executed in the interval after their release (start of a transition execution) and before a certain deadline (end of a transition execution).

Goal of this section is to show that hybrid automata can be used to model the timing behavior of a prototype implementation under the application of a specific scheduling policy. Schedulability analysis, i.e., verifying whether in all possible behaviors under the given scheduling policy all operations are guaranteed to meet their deadline, then reduces to property verification of the automaton.

This work was inspired by research of Corbett which showed in [25] how timing behavior of Ada95 programs can be modeled and analyzed by using hybrid automata. Our approach, however, starts from the opposite direction. Instead of making an abstract model of the Ada95 implementation, we start with the hybrid automaton which results from the Astral to hybrid automaton transformation defined in chapter 4. This automaton will be called the abstract automaton.

This automaton is then refined to model the scheduling and resource consumption of operations implementing transition executions. Such a refined abstract automaton is then referred to as an implementation automaton.

Schedulability analysis is then a two-step process: Firstly, the composition of the implementation automaton and a scheduling automaton (modeling scheduling of operations onto processors) is taken. Secondly, analysis on the resulting automaton is performed to verify whether all operations will always meet their deadlines.

In the next subsection the construction of the implementation automaton is discussed. How scheduling policies can be defined by hybrid automata is addressed in section 6.4.3. Timing analysis through verification of the composition of the implementation automaton and the scheduling automaton is discussed in section 6.4.4. Experiences on using tools for automated verification of hybrid automata are summarized in the same section.

6.4.2 Construction of the Implementation Automaton

The automaton, resulting from the translation of a single Astral process \( P_i \in Dproc \) (see section 4.4.3), is the automaton:
6.4. TIMING ANALYSIS

\[ \text{AHA}_i = (\text{Var}_i, \text{Loc}_i, \text{Edge}_i, \text{Inv}_i, \text{Jump}_i, \text{Lab}_i, \text{Ev}_i, \text{Init}_i) \]

defined as follows:

- \( \text{Var}_i = \bigcup_{T_j \in \text{Trs}(P_i)} \{ \text{start} \_ T_j, \text{end} \_ T_j \} \cup \text{Svar}(P_i) \cup \{ \text{now} \} \cup \{ \text{c} \_ P_i \} \)
- \( \text{Loc}_i = \bigcup_{T_j \in \text{Trs}(P_i)} \{ \text{exec} \_ T_j \} \cup \{ \text{idle} \_ P_i \} \)
- \( \text{Edge}_i = \bigcup_{T_j \in \text{Trs}(P_i)} \{ (\text{idle} \_ P_i, \text{exec} \_ T_j) \} \cup \bigcup_{T_j \in \text{Trs}(P_i)} \{ (\text{exec} \_ T_j, \text{idle} \_ P_i) \} \)
- \( \text{Inv}_i = \{ \text{idle} \_ P_i \mapsto \text{now} = 1 \} \cup \bigcup_{T_j \in \text{Trs}(P_i)} \{ \text{exec} \_ T_j \mapsto \text{now} = 1 \wedge \text{c} \_ P_i = 1 \} \)
- \( \text{Jump}_i = \bigcup_{T_j \in \text{Trs}(P_i)} \{ (\text{idle} \_ P_i, \text{exec} \_ T_j) \mapsto [\text{Pre}(T_j)]' \wedge \text{c} \_ P_i = 0 \wedge \text{start} \_ T_j = \text{now}, (\text{exec} \_ T_j, \text{idle} \_ P_i) \mapsto [\text{Post}(T_j)] \wedge \text{c} \_ P_i = \text{Dur}(T_j) \wedge \text{end} \_ T_j = \text{now} \} \)
- \( \text{Lab}_i = \bigcup_{T_j \in \text{Trs}(P_i)} \{ \text{St} \_ T_j, \text{E} \_ T_j \} \)
- \( \text{Ev}_i = \bigcup_{T_j \in \text{Trs}(P_i)} \{ (\text{idle} \_ P_i, \text{exec} \_ T_j) \mapsto \text{St} \_ T_j, \text{exec} \_ T_j, \text{idle} \_ P_i \mapsto \text{E} \_ T_j \} \)
- \( \text{Init}_i = \{ \text{Idle} \_ P_i \mapsto \text{Init}(P_i) \wedge \text{now} = 0 \} \cup \bigcup_{T_j \in \text{Trs}(P_i)} \{ \text{exec} \_ T_j \mapsto \text{False} \} \)

The abstract Astral automaton \( \text{AA}_{\text{spec}} \) is the composition:

\[ \text{AA}_{\text{spec}} = \bigcup_{P_i \in \text{Dproc}} \text{AHA}_i \]

For now, assume that the worst-case execution time of the operation implementing a transition is known and given by \( W\text{cet}(T_j) \). The duration of the operations associated with transitions is modeled by the implementation automaton, which is based on the \( \text{AHA}_i \) (AHA). The implementation automaton (IHA) is defined as follows:

\[ \text{IHA}_i = (\text{Var}_i, \text{Loc}_i, \text{Edge}_i, \text{Inv}_i, \text{Jump}_i, \text{Lab}_i, \text{Ev}_i, \text{Init}_i) \]

in which:

- \( \text{Var}_i = \text{Var}_{\text{AHA}_i} \cup \{ \text{c} \_ \text{exec} \_ P_i \} \)
- \( \text{Loc}_i = \bigcup_{T_j \in \text{Trs}(P_i)} \{ \text{started} \_ T_j, \text{sched} \_ T_j, \text{ready} \_ T_j \} \cup \{ \text{idle} \_ P_i \} \)
- \( \text{Edge}_i = \bigcup_{T_j \in \text{Trs}(P_i)} \{ (\text{idle} \_ P_i, \text{started} \_ T_j) \} \cup \bigcup_{T_j \in \text{Trs}(P_i)} \{ (\text{started} \_ T_j, \text{sched} \_ T_j) \} \cup \bigcup_{T_j \in \text{Trs}(P_i)} \{ (\text{sched} \_ T_j, \text{started} \_ T_j) \} \cup \bigcup_{T_j \in \text{Trs}(P_i)} \{ (\text{sched} \_ T_j, \text{ready} \_ T_j) \} \cup \bigcup_{T_j \in \text{Trs}(P_i)} \{ (\text{ready} \_ T_j, \text{idle} \_ P_i) \} \)
CHAPTER 6. PROTOTYPING ASTRAL SPECIFICATIONS

- \( Inv_i = \{ idle_P_i \mapsto now = 1 \} \cup \)
  \( \cup_{T_j \in Trs(P_i)} \{ started_T_j \mapsto now = 1 \land c.P_i = 1 \)
  \land c.exec.P_i = 0 \} \cup \)
  \( \cup_{T_j \in Trs(P_i)} \{ sched_T_j \mapsto now = 1 \land c.P_i = 1 \)
  \land c.exec.P_i = 1 \} \cup \)

- \( Jump_i = \)
  \( \cup_{T_j \in Trs(P_i)} \{ (idle_P_i, started_T_j) \mapsto [Pre(T_j)]'' \land \)
  c.P_i = 0 \land c.exec.P_i = 0 \land start_T_j = now \} \cup \)
  \( \cup_{T_j \in Trs(P_i)} \{ (started_T_j, sched_T_j) \mapsto True \} \cup \)
  \( \cup_{T_j \in Trs(P_i)} \{ (sched_T_j, started_T_j) \mapsto \)
  c.exec.P_i'' < Wcet(T_j) \} \cup \)
  \( \cup_{T_j \in Trs(P_i)} \{ (ready_T_j, idle_P_i) \mapsto Post(T_j) \land \)
  c.P_i = Dur(T_j) \land end_T_j = now \} \cup \)
  \( \cup_{T_j \in Trs(P_i)} \{ (sched_T_j, ready_T_j) \mapsto \)
  c.exec.P_i'' = Wcet(T_j) \}

- \( Lab_{iha} = \)
  \( \cup_{T_j \in Trs(P_i)} \{ St.T_j, Sch.T_j, Y.T_j, Pr.T_j, E.T_j \} \)

- \( Eviha = \)
  \( \cup_{T_j \in Trs(P_i)} \{ (idle_P_i, started_T_j) \mapsto St_T.j \} \cup \)
  \( \cup_{T_j \in Trs(P_i)} \{ (started_T_j, sched_T_j) \mapsto Sch.T_j \} \cup \)
  \( \cup_{T_j \in Trs(P_i)} \{ (sched_T_j, started_T_j) \mapsto Pr.T_j \} \cup \)
  \( \cup_{T_j \in Trs(P_i)} \{ (sched_T_j, ready_T_j) \mapsto Y.T_j \} \cup \)
  \( \cup_{T_j \in Trs(P_i)} \{ (ready_T_j, idle_P_i) \mapsto E.T_j \} \)

- \( Init_{iha} = \{ Idle_P_i \mapsto Init(P_i) \land now = 0 \} \cup \)
  \( \cup_{T_j \in Trs(P_i)} \{ started_T_j \mapsto False \} \cup \)
  \( \cup_{T_j \in Trs(P_i)} \{ sched.T_j \mapsto False, ready_T_j \mapsto False \} \)

The Astral implementation automaton \( AA_{imp} \) is the composition:

\[
AA_{imp} = \|_{P_i \in D_{proc}} IHA_i.
\]

An overview of the transformation step from AHA to IHA is given in Figure 6.6. From this figure it becomes immediately clear that the IHA-automaton can be obtained from the AHA-automaton by replacing the \( exec.T_j \)-location by three separate locations:

- \( started.T_j \). When control is in this location Astral-transition \( T_j \) has been started;
- \( sched.T_j \). When control resides in \( sched.T_j \), the operation associated with \( T_j \) is executing;
- \( ready.T_j \). When the operation associated with \( T_j \) has finished and awaits the end of the Astral-transition \( T_j \).
6.4. TIMING ANALYSIS

As before, the labels $St.T_j$ and $E.T_j$ are assigned to edges modeling the start and end of an Astral transition execution. The label $Sch.T_j$ is assigned to the edge which models the action of scheduling the operation for execution on a processor. Preemption is modeled by the edge to which the label $Pr.T_j$ is assigned. Finally, finishing the execution of the operation associated with an Astral transition and yielding the processor is modeled by the edge labeled with $Y.T_j$.

$AHA_i$ for Astral process $P_i$  
$IHA_i$ for Astral process $P_i$

Figure 6.6: The step from AHA to IHA automaton

6.4.3 Construction of a Scheduling Automaton

In the implementation automaton discussed in the previous section, the scheduling policy has been left open. In this section we will discuss how a scheduling policy can be defined using a hybrid automaton. In general, a scheduler is responsible for granting requests for resources thereby guaranteeing certain scheduling constraints to be met. The discussion is restricted to the single processor case (i.e., all operations are scheduled on a single processor) but is easily generalized to the multi-processor case.

The start of a transition execution implies a request for the processor on
which the operation has to be scheduled. These requests are registered using an automaton, the request automaton, which has been defined as depicted in Figure 6.7.

![Request Automaton (RQ) and Scheduler Automaton (SA)](image)

Figure 6.7: Scheduler automaton

The request automaton will when composed with the $IHA_i$ of an Astral process $P_i$ synchronize with the start of a Astral transition execution because there exist edges labeled $St.T_j$ in both the request automaton and the implementation automaton. Upon the start of an Astral transition execution the request automaton will set the value of the state variable $req.P_i$ to True.

Actual grants of processor requests (i.e., scheduling a process on the processor) are specified by a scheduler automaton depicted on the right of Figure 6.7. Control of the scheduler automaton resides in $Proc.idle$ when all $req.P_i = False$. In the case of one or more requests the scheduler decides which of the outstanding requests is serviced and the enabled transition labelled $Sch.T_i$ is taken.

Preemption is modeled by the transition $Pr.T_j$. While a yield of the processor (the operation has finished execution) is modeled by the transition $Y.T_j$ and in this case the request, $req.P_i$, is assigned False.

The scheduling automaton is very general but it is easy to model other scheduling policies. For example, by introducing variables to keep process priorities and defining additional transition conditions priority-driven scheduling policies can be easily defined.

Timing analysis, then reduces to checking the absence of deadlock [16] of the automaton:

$$AS = \parallel_{P_i \in D_{proc}} IHA_i \parallel RQ \parallel SA.$$
6.4. TIMING ANALYSIS

6.4.4 Verification Experiences

The techniques discussed above provide us with interpretations of Astral specifications that incorporate the binding of Astral processes to resources. The resulting system (composition of a set of IHA's, a request automaton and a scheduler automaton) can now be used to decide whether or not implementation based on this interpretation is feasible. Goal is to verify whether or not this system is a correct refinement of the abstract automaton.

Obviously, this is the case if every transition always completes within its deadline. This means that for any transition $T_j$ of a process $P_i$, the $ready.T_j$ location is reached no later than the moment $c.P_i$ becomes $Dur(T_j)$. If an action does not complete within its deadline it will cause its process automaton to block in the $ready.T_j$ state (see the automaton depicted on the right of Figure 6.6). Verification should then demonstrate the absence of such blocking states.

Since we do not have a verification tool to verify our type of automata, we used the symbolic model checking tools HyTech [38] and Uppaal [51] to show the feasibility of our approach. HyTech allows verification of a wide range of properties of real-time and hybrid systems defined as hybrid automata [2] with non-urgent transitions. Uppaal is a tool for verification of timed automata with integer variables. It allows verification of a more restricted set of properties, but is generally more efficient.

The timed automata of Uppaal and the hybrid automata of HyTech constraining the elapsing of time in a different manner. They do not have (general) urgent transitions. When mapping our hybrid automata with urgent transitions to the timed (or hybrid) automata of Uppaal and HyTech, one has to express the urgency through the location invariants of timed automata or by introducing additional automata to force transition executions. This is done by putting invariants on locations that disallow the progress of time when outgoing urgent transitions are enabled.

This means that a $ready.T_j$ location is given an invariant $c.P_i \leq Dur(T_j)$. As a consequence, time will be blocked in case a deadline is not met, since in this case neither a transition can be taken nor can time progress in that location. Verification can therefore be done by performing non-zenoness analysis. If the automaton is non-zeno (time can always progress), then there are no blocking states as described above, and therefore the system is schedulable. The reverse implication is true only if the translation of the abstract automaton (AHA, depicted on the left of Figure 6.6) in HyTech- or Uppaal-automata has been proved to be non-zeno. Thus one first makes sure that the system is non-zeno, and subsequently checks whether the implementation scheme is feasible, by checking the non-zenoness property of the implemen-
6.5 Evaluation

In this chapter we developed a partial mapping of Astral to Ada95 constructs which enables the quick construction of a prototype. Based on a given Astral specification a prototype can be realized quickly. This enables one to study possible behaviors of the defined system.

From practical experiences we conclude that such a prototyping approach is useful. Through analysis of the behavior of the prototype an impression of possible system behaviors can be obtained. As has been demonstrated in the example case, such a prototype can be used to detect errors in the specification. Furthermore, a working prototype provides evidence for the feasibility of the system.

We faced a major problem in performing timing analysis of the prototype implementation before run-time. This subject has therefore been addressed in the second part of this chapter. It was shown that the hybrid automaton formalism is expressive enough to model timing behavior of the prototype implementation. The proposed approach to timing analysis is based on extending the hybrid automaton specification, that results from an Astral specification, with implementation details (timing figures) derived, for example, from a prototype implementation. Timing analysis then reduces to verification of blocking states of the resulting implementation automaton.

We have tried the use of two different model-checkers, Uppaal and HyTech, to perform analysis of implementation automata. For this purpose, Astral-to-Uppaal and Astral-to-HyTech translators have been developed (see also appendix D). The current generation of model-checking tools fails in verifying the non-zenoness property of an implementation automaton of realistically sized system specifications. However, the proposed approach is feasible because we succeeded in verifying the non-zenoness property for a small system specification [16].
Chapter 7

Summary and Conclusion

In this thesis we considered the use of a formal software specification language in the development of real-time embedded control systems. In this chapter, the thesis summary and conclusions are presented.

7.1 Summary

Emphasis in our research has been put on the application of available specification languages and the use of existing theory in the area of formal software specification. The context in which the research has been carried out is depicted in Figure 7.1.

First, the development process in the control engineering domain was considered in chapter 2. Attention focused on control system development based on the use of the block diagram modeling notation. A software engineering point of view was taken and demonstrated the existence of a discontinuity between control engineering and software engineering. In chapter 2 we discussed different classes of block diagram models. This provided the necessary background and made clear why and how they are being used in the development of a control system. Arguments were presented to indicate that the use of the block diagram notation in the development of a control system is worthwhile and justifiable. Although block diagrams could be used to define software requirements to some extent, it became clear that there are typical software requirements which cannot be defined by block diagrams. This indicated the necessity to use another specification language in the definition of software requirements.

Chapter 3 then considered software development, especially formal software requirements specification. First, the use of formal methods in software development was discussed. Then Astral, a language for requirements spec-
Formalization of real-time software was described. Formality is preferred because in multidisciplinary development (in which engineers with different backgrounds are involved) errors due to misunderstandings are more likely to be made. Astral was selected because its semantics were formally defined and formality was regarded essential in the given context. The simplicity of the Astral language was also considered important because this enables the use of the language by both control and software engineers.

Initially, we focused on how an Astral software requirements specification could be derived from a given block diagram. Reformulation of a block diagram specification in Astral should result in a first software requirements specification. The partial software requirements specification resulting from reformulation of block diagrams can then be completed by defining additional software requirements. The applicability of the block diagram notation in modeling and analysis of both discrete as well as continuous system components caused problems regarding reformulation in Astral. Reformulation and use of Astral in the control engineering domain would be facilitated if Astral would become usable for the specification of continuous system components also.

An extended version of Astral was developed in chapter 4, called H-Astral. The semantics of H-Astral were defined by a translation of H-Astral spec-
7.1. SUMMARY

ifications in a variant of the hybrid automata developed by Alur & Henzinger [3]. A mapping of H-Astral processes on hybrid automata was then discussed. This mapping is partial because the Astral to hybrid automaton translation has been defined for a limited class of schedule constraints only. We discussed an experiment concerning automatic verification of a hybrid automaton specification of an inverted pendulum control system. Current tools for automatic verification were shown to be useful to a limited extent in verifying this example specification. But, since the theory of hybrid automata has only recently been developed, this situation might improve in the future.

The applicability of H-Astral in modeling control systems was evaluated in chapter 5. This chapter reported the results of a case study in which H-Astral was used for the specification of a robot control system. The reformulation of a set of already developed block diagram models in H-Astral was discussed and resulted in a first but incomplete H-Astral software requirements specification.

Various requirements which were not defined by the available block diagram specifications of the system were added to the requirements specification in H-Astral. Some of these requirements could have been defined in block diagrams but were not defined, for a variety of reasons, such as to facilitate mathematical analysis of the block diagram specifications. Of other requirements it is doubtful whether it would be possible to define them in a block diagram specification. These requirements concern, for example, the way a robot job should be carried out, the 'flip'-movement and alarm-handling. H-Astral has been shown particularly useful in defining a number of these important software (timing) requirements for the robot control system. However, reformulation of block diagrams must be carried out manually which is regarded as being a disadvantage.

A simulator for H-Astral specifications has been developed. Simulation was shown to be useful because errors in the specification were more easily detected through simulation. The ability to simulate behavior of the controlled processes was also shown useful. Firstly, this enabled us to validate the H-Astral specification by comparing simulation results of the H-Astral specification with those obtained from simulating block diagram specifications. Secondly, simulation of (environmental) physical processes showed to be useful in detecting less obvious errors in the specification of the discrete controller. In our example, simulation results showed that the robot could be controlled such that it would get outside its mechanically allowed range. We think that those errors are less easily detected (and elicited) in case environmental processes would remain unmodeled in the specification.

Once an H-Astral specification is available, prototyping was shown to be
beneficial and applicable for realistically sized systems. Especially, the use of typical Ada95 primitives like the protected object and task made it possible to generate a prototype from the specification easily and quickly. We described our experiences in prototyping and showed how timing behavior could be studied at run time. Timing analysis of the prototype before run time, through automatic verification of the hybrid automaton model is possible but severely limited with respect to the size of the H-Astral specification.

7.2 Conclusion

The focus of our research has been on the use of a software requirements specification language in the development of real-time embedded control systems. Initially (see section 1.3), the focus of our research has been formulated as follows:

\[\text{To investigate the use of a formal software requirements specification language in and to develop technology to support the development process of embedded control applications.}\]

From our research we conclude that it makes sense to employ a formal software specification language for real-time embedded control applications. In particular:

1. We think to have found a useful way to construct an initial software requirements specification based on a block diagram specification of the control system. Based on the software specification language Astral, H-Astral has been developed which allows the transfer of block diagrams to a software requirements specification. Useful feedback through simulation and prototyping can then be generated. Throughout this thesis examples have been given of how this can improve requirements elicitation;

2. H-Astral offers the possibility to study behavior of a discrete software specification in relation with a continuous environment. This facilitates the use of H-Astral in the CE-domain;

3. In the construction of a requirements specification for a robot control system, we found it useful to have an abstract language like H-Astral available for requirements specification. Simulation was shown to be useful to detect and correct errors in the specification of a robot control system.
7.2. CONCLUSION

One important conclusion follows from our experiences with the use tools for the analysis and simulation of H-Astral specifications. Current experiences demonstrate that this is difficult and revealed the trade-off involved between the desire to accommodate the needs of the specifier, and the desire for having analyzable specifications. Based on current experiences, we do not think that our choice to enable the specification of continuous system components was wrong. The research results demonstrate that this can work out positively in bridging the signaled gap between control and software engineering.

A definitive answer, however, requires more practical experience on the use and (automated) analysis of H-Astral specifications. Based on these experiences we can then judge whether the advantages outweigh the disadvantages regarding the use of a hybrid system specification language, and whether this choice indeed has been the right one.

From our research we can also conclude that we did not completely succeed in our goals with respect to the following:

1. In the introductory chapter we signaled the existence of a discontinuity. However, we cannot claim to have fully removed the discontinuity from the development process of embedded control systems, we do think to have created a possibility to improve the current situation through the use of the H-Astral specification language.

2. The decision to use H-Astral has limited us to modeling behavior of discrete system components through transitions with duration greater than zero. This implies that reformulation of block diagrams in H-Astral must be performed manually because the operations modeled by a block are carried out instantaneously. This (semantic) incompatibility still leaves a discontinuity in the development process.

Discontinuities in the development seem inevitable because they result from differences in goals and (conflicting) interests of engineers with different backgrounds involved in the development of a system. As an example: from a software engineering point of view it is justifiable to associate durations with actions, as required of a transition specification in Astral and not abstract from them to facilitate analysis like is done in the block diagram notation. Trying to solve this discontinuity is regarded as a major issue to be addressed in future research.

Related research is concerned with techniques which can be employed for the (semi-)automatic analysis of H-Astral specifications. Our experiences indicate that not only the specification language but also the facilities for verifying specifications determine its usability to a large extent. In order to be useful in practice, it is thus important not only to have a well-defined
language but also to have tool support available.

Based on our positive experiences with the prototype tools (an overview of these tools is presented in appendix D) we think their use should be addressed more extensively in future research. One topic of future research could focus on improving tool support. The complexity of the H-Astral specifications that can be handled by the current tools is too low and application of these tools for practically sized specifications is not yet feasible. Another topic of research could focus on the use of tools in an overall (multidisciplinary) development process of real-time embedded control applications. First thoughts on these topics of research are reported in [15].
Appendix A

Syntax Definition of Hybrid-ASTRAL

The syntax of H-Astral is defined using syntax diagrams (also called railroad diagrams). A syntax diagram consists of a number of interconnected square boxes and boxes with rounded corners. The square boxes represent terminals and boxes with rounded corners represent the nonterminals of the language. The sequence of terminals and nonterminals that can be produced following a path starting at the left of a syntax diagram and following the lines is a valid sentence of the language. A valid H-Astral program is one which can be obtained starting with the rail diagram named program below.

\[
\text{program}
\]

\[
\text{globalspec} \quad \text{pspecs}
\]

\[
\text{globalspec}
\]

\[
\text{GLOBAL SPECIFICATION} \quad \text{Id} \quad \text{PROCESSES} \quad \text{pdecls}
\]

\[
\text{TYPE} \quad \text{typedefs} \quad \text{conscl} \quad \text{defcl} \quad \text{invcl} \quad \text{ctrcl}
\]

\[
\text{schedcl} \quad \text{END GLOBAL SPECIFICATION}
\]

\[
\text{pdecls}
\]

\[
\text{Id} \quad : \quad \text{proctype}
\]
typedefs

conscl

defcl

invcl

ctrcl

schedcl

consdefs

funcdefs

typedesc
ifprefix
- \texttt{IF} ( \texttt{pred} ) \texttt{THEN} \texttt{fparmlist} \texttt{;} \\

funcparams
- \texttt{fparmlist} \texttt{;} \\

strelmlist
- \texttt{Id} \texttt{;} \texttt{typename} \texttt{;} \\

fparmlist
- \texttt{Id} \texttt{;} \texttt{typedecl} \texttt{;} \\

typedecl
- \texttt{REAL} \texttt{INTEGER} \texttt{BOOLEAN} \texttt{IDTYPE} \texttt{Time} \texttt{Id} \\

relop
- \texttt{>=} \texttt{<=} \texttt{<} \texttt{>} \texttt{<>} \texttt{=}
idornumber

```
Number
Id
```

statevardecls

```
Id :: typename ;
```

pred

```
expr
  expr relop expr
  { pred }
  ( pred )
  pred & pred
  pred | pred
  expr ISIN expr
  pred IMPLIES pred
  pred IFF pred
  NOT pred
forallpred
existpred
```

forallpred

```
FORALL fparmlist ( pred )
```

existpred

```
EXISTS fparmlist ( pred )
```
localdefs
- conscl defcl invcl ctrcl schedcl

statevardecl
- VARIABLE statevardecls

init
- INITIAL pred ;

typename
- ARRAY [ discrange ] OF typedcl

transspecs
- transspec

transspec
- TRANSITION Id transparams idornumber
- PRE pred POST pred

transparams
- ( fpamlist )

relationsec
- RELATION caselist END RELATION ;

caselist
- CASE pred : relations

relations
- expr = expr ;
Number

\[
\text{digit} \quad . \quad \text{digit}
\]

Id

\[
\text{letter} \quad / \quad \text{letterordigit}
\]

digit \quad = \quad 1..9

letter \quad = \quad a..z \mid A..Z \mid _

letterordigit \quad = \quad \text{letter} \mid \text{digit}

The following words are reserved and may not be used as identifiers (Id):

ARRAY, AXIOM, BEGIN, BOOLEAN, CASE, CLAUSE, CONSTANT, CONSTRAINT, CONTINUOUS, DEFINE, DEFINITION, DISCRETE, ELSE, ELSIF, END, ENDIF, ENVIRONMENT, EXISTS, EXPORT, FALSE, FORALL, FUNCTION, GLOBAL, IDTYPE, IF, IMPLIES, IMPORT, INITIAL, INTEGER, INVARIANT, IS, ISIN, LIST, MOD, NOT, OF, POST, PRE, PROCESSES, REAL, RELATION, RETURN, SCHEDULE, SPECIFICATION, STRUCTURE, THEN, TRANSITION, TRUE, TYPE, TYPEDEF, VAR, VARIABLE, WHILE
Appendix B

Transfer Function Manipulations

B.1 Introduction

In this appendix the transformation of transfer functions (see chapter 2) into
time dependent functions is discussed. Continuous phenomena are defined
by transfer functions in H-Astral. However, only time dependent functions
can be used in hybrid automaton specifications.

In mapping H-Astral specifications on hybrid automata a transformation
of transfer functions is thus required. Techniques, originating from the con-
trol engineering domain, can be employed to carry out the transformation of a
transfer function and results in a set of equations describing continuous-time
behaviour.

Furthermore, in a H-Astral specification, it is allowed to describe contin-
uous system behaviour by taking the composition of a number of transfer
functions each defining system behaviour under different conditions (also
denoted modes). Such a composition requires additional consideration be-
cause in general the composition immediately results in a non-linear system
(see chapter 2). In this appendix the transformation of transfer functions
is defined and it also indicates the limitations associated with the proposed
transformation and composition of transfer functions.

The incorporation of continuous system models into H-Astral was moti-
vated by our goal to improve on current practice in requirements specification
of real-time embedded control software. Improvements in requirements spec-
ification are expected from the ability to study (discrete) controller behaviour
in relation with a (possible) continuous environment. In the integrated devel-
opment approach continuous models are developed using conventional control
engineering techniques. One means of defining the relation between input and output signals of a system is the transfer function, graphically represented by so-called block diagrams. In order to make H-Astral better suited for use in the control engineering domain, we choose to enable the specification of continuous phenomena by transfer functions in H-Astral.

The transformation consists of three successive steps:

1. Using an existing technique to obtain state-space equations from a given transfer-functions;
2. Specification (by the control engineer) of the conditions under which the obtained state-space representation is a valid real-world model
3. Incorporation of the state-space equations from step 1 and the conditions formulated in step 2 in a hybrid automaton.

B.2 Laplace Transform To State-Space Description

In control engineering a technique called the Laplace transformation is used to facilitate calculations in which continuous-time differential equations are being used. For many system components the relation between input and output signal is defined using the Laplace transform [19]. Physical components of a system can be modelled by a differential equation and these equations can be manipulated and analysed through the use of the Laplace transform.

The Laplace transformation of a differential equation results in a transfer function defined as follows: transfer function = the ratio of the output to input of a given component, i.e., \( H(s) = \frac{Y(s)}{X(s)} \)

Here attention will be restricted to transfer functions of the following kind (that can be used in modelling most physically realizable, linear systems):

\[
\frac{Y(s)}{U(s)} = \frac{b_{n-1}s^{n-1} + b_{n-2}s^{n-2} + \cdots + b_0}{s^n + a_{n-1}s^{n-1} + \cdots + a_0}
\] (B.1)

In the formalism of hybrid automata continuous phenomena are defined using piecewise-continuous, differentiable functions of the following kind: \( f : [0, \delta] \rightarrow \text{Val} \). Obviously, there is a clear discrepancy between the transfer function in equation B.1 and the form of the time-dependent functions used (needed) in hybrid automata. It is therefore needed to define devise a transformation of transfer functions into continuous functions in the time domain. In existing control engineering literature we found a number
of useable techniques to carry out this transformation and one of them is discussed here together with the conditions under which it can be applied.

In [22, 29] a straightforward transformation is described resulting in the following following time-domain relation when equation B.1 is transformed:

\[
\begin{pmatrix}
\dot{x}_0 \\
\vdots \\
\dot{x}_{n-2} \\
\dot{x}_{n-1}
\end{pmatrix} = 
\begin{pmatrix}
0 & 1 & 0 & \cdots & 0 \\
0 & 1 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 1 & 0 \\
-a_0 & -a_1 & \cdots & -a_{n-2} & -a_{n-1}
\end{pmatrix}
\begin{pmatrix}
x_0 \\
x_1 \\
\vdots \\
x_{n-2} \\
x_{n-1}
\end{pmatrix} + 
\begin{pmatrix}
x(t) \\
0 \\
\vdots \\
x(t) \\
0
\end{pmatrix} u(t)
\]

\[y(t) = (b_0 b_1 \cdots b_{n-2} b_{n-1}) \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{pmatrix} + c\]

(B.2)

The coefficients of the matrices have been derived from the coefficients of the transform function. The transformation leads to the introduction of \(n\) additional (continuous) state variables \(x_0 \ldots x_{n-1}\). \(y(t)\) denotes the output signal and \(u(t)\) the input signal.

In [19] another technique is discussed based upon rewriting the transfer function as a signal flow diagram. Application of this technique results in equations of the same kind as equations B.2.

In each case the transfer function of the system \(H(s)\) is transformed resulting in a set of coupled equations:

\[
\begin{align*}
\dot{x}_0 &= f_0(x_2) \\
\dot{x}_1 &= f_1(x_2, x_3) \\
\vdots &= \vdots \\
\dot{x}_{n-1} &= f_{n-1}(x_0, \ldots, x_{n-2}, u(t)) \\
y &= f(x_0, \ldots, x_{n-1})
\end{align*}
\]

Besides solving this set of coupled equations analytically they are also often solved numerically to obtain the system's response \(y(t)\) upon the input \(r(t)\). Numerical techniques for solving these equations are amongst others applied in popular mathematical and simulation package like Matlab [56] and Simulink [57].
B.3 Defining a Hybrid Automaton

The set of equations like B.3 can be incorporated in an invariant of a location of the hybrid automaton, by simply taking the conjunction of all equations and the introduction of additional state variables $x_0, \ldots, x_{n-1}$.

The presence of higher-order derivatives in a hybrid automaton has already been discussed in [39]. Theoretically this is feasible but analysis of hybrid automata in the presence of higher order derivatives is still complex. The resulting hybrid automata is called non-linear because it allows specification of data-dependencies between continuous variables, i.e., the first time derivative of a continuous variable is not constant but dependent on the value of other (continuous) state variables [39].

In case behavior of a continuous H-Astral process is defined by multiple modes the following is applicable to the jump condition between different locations (assuming one output, $y \in Var$):

- An additional state variable $y_{offset}$ is defined;
- Each location of the hybrid automaton models behavior of the continuous process in a particular mode. The transfer function corresponding to this mode is transformed into state-space form, resulting in a set of equations analogous to those defined in equation B.3. The invariant of each location is defined equal to the conjunction of all these equations.
- The equation $y = f_n(x_0, \ldots, x_{n-1})$ in the invariant of each location is then replaced by $y = f_n(x_0, \ldots, x_{n-1}) + y_{offset}$
- The jump condition for each edge of the hybrid automaton then equals: $x_0 = 0 \land \ldots x_{n-1} = 0 \land y_{offset} = y$
Appendix C

Robot Control System Specification

/* robot.ast
   DATE CREATED : April 1995 AUTHOR : Klaas Brink
   DATE LAST MODIFIED : 13 - 03 - 1997 BY : Klaas Brink
*/
/*==============================================================================*/
GLOBAL SPECIFICATION Robot_Control_System

PROCESSES
/* CONTINUOUS
   /* Physical Process Models (lowest level, 0)
   axis1 : Axis1;
   axis2 : Axis2;
   axis3 : Axis3;
   axis4 : Axis4;
   force : Force;
   */
/* DISCRETE
   gripper : Gripper;
   /* Sensors (lowest level, 0)
   ax1_sensor : Ax1_Sensor;
   ax2_sensor : Ax2_Sensor;
   ax3_sensor : Ax3_Sensor;
   ax4_sensor : Ax4_Sensor;
   f_sensor : F_Sensor;
   /* Actuators (lowest level, 0)
   ax1_act : Ax1_Actuator;
   ax2_act : Ax2_Actuator;
   ax3_act : Ax3_Actuator;
   ax4_act : Ax4_Actuator;
   */
/* Control Loop Processes (level 1)
   ax1_ctr : Ax1_Control;
   ax2_ctr : Ax2_Control;
   ax3_ctr : Ax3_control;
   ax4_ctr : Ax4_control;
   gr_ctr : Gr_control;
   */
/* Control Management Processes (level 2)
   ax12_cman : Ax12_Manager;
   ax3_cman : Ax3_Manager;
   ax4_cman : Ax4_Manager;
   */
/* Highest Level, Environment Interface Process (level 3)
   t_man : Task_Manager;
   */

TYPE
   CtrModeType = {JointPos, JointVel, CartPos, CartVel, Force, Stopped};
   Gr_State = {Opened, Closed};
ManState = {Idle, Executing, Ready, Check};
ManMode = {Stopped, JointPos, CartPos, CartVel, Force, Check, Check_Done, Flip};
/* The following type definitions are used to express the constraints that should be met in order to end a robot-job successfully. */
BoolOp = {LT, LEQ, GT, GEQ, NEQ, EQ};
Operand = {ax1pos, ax2pos, ax3pos, ax4pos, ax1vel, ax2vel, ax3vel, ax4vel, xpos, ypos, zpos, xvel, yvel, zvel, fx, fy, fz, ax4torque};
Predicate IS STRUCTURE OF
  ( predsym : BoolOp;
   variable : Operand; /* Criterion variable */
   val : REAL;)
PredList IS LIST OF Predicate; /* Conjunction of predicates = constraint */
ConsList IS LIST OF PredList; /* A disjunction of constraints */
TaskRec IS STRUCTURE OF
  (ax1setpoint : REAL;
   ax1mode : CtrModeType;
   ax2setpoint : REAL;
   ax2mode : CtrModeType;
   ax3setpoint : REAL;
   ax3mode : CtrModeType;
   ax4setpoint : REAL;
   ax4mode : CtrModeType;
   tskcons : ConsList;
   tsk_timeout : REAL;)
TaskList IS LIST OF TaskRec; /* Contains the robot-tasks to be executed */
TaskManState = {Idle, Executing};

CONSTANT
  T_sample = 0.001; /* Sample time in all control modes except 'Force' */
  T_sforce = 0.0045; /* Sample time for force control */
  L1 = 0.445;
  L2 = 0.355; /* See Rieswijk, App. D, p. 299 */
  ERROR = 99999.00;

DEFINITION
FUNCTION ABS(pval:REAL):REAL
VAR rval : REAL;
BEGIN
  IF (pval < 0) THEN
    rval := -1*pval;
  ELSE
    rval := pval;
  END;
RETURN rval;
END

FUNCTION P_Contr(K_p:REAL; des_val:REAL; curr_val:REAL;
                   contr_val:REAL; error:REAL;):REAL
VAR corr : REAL;
BEGIN
  /* Implementation of P-Controller e.g. Bucek, p. 250 */
  corr := K_p*(des_val - curr_val) - K_p*error;
  error := des_val - curr_val;
  RETURN curr_val + corr;
END

FUNCTION PI_Contr(K_p:REAL; K_i:REAL; des_val:REAL; curr_val:REAL;
                   contr_val:REAL; error:REAL; deltaT:REAL;):REAL
VAR corr : REAL;
BEGIN
    /* Implementation of PI-Controller e.g. Bueck, p. 250 */
    corr := (K_p+K_i*deltaT/2)*(des_val-curr_val)-
             (K_i*deltaT/2-K_p)*error;
    error := des_val-curr_val;
    RETURN curr_val+corr;
END

FUNCTION Limiter(slope:REAL; delt:REAL;dvalue:REAL;
                  cvalue:REAL):REAL
VAR
    min : REAL;
    max : REAL;
    rvalue : REAL;
BEGIN
    max := cvalue + slope*delt;
    min := cvalue - slope*delt;
    IF (dvalue<min) THEN
        rvalue := min;
    ELSE IF (dvalue>max) THEN
        rvalue := max;
    ELSE
        rvalue := dvalue;
    ENDIF
END
END

GLOBAL SPECIFICATION
                        */
CONTINUOUS SPECIFICATION Axis1
IMPORT u_axiservo;
EXPORT ax1_angle,ax1_omega;
CONSTANT
    /* Taken from table 4.1 Klomp or from table 2.9 Rieswijk */
    K_m1 = 0.240;
    K_m2 = 1.45;
    N_1 = 129;
    J_m1 = 0.00108;
    F_vm1 = 0.001;
    J_L1 = 8.9;
    F_vL1 = 0.9;
    K_h1 = 81000;
    D_h1 = 96;

VARIABLE
    ax1_angle : REAL;
    ax1_omega : REAL;
    initial TRUE;
RELATION
    CASE TRUE :
        x1_3 = J_m1*J_L1;
        x1_2 = J_m1*(F_vL1+D_h1)+J_L1*(F_vm1+(K_m1^2/R_m1))
               + (D_h1/N_1^2); 
        x1_1 = F_vL1*(F_vm1+K_m1^2/R_m1)+
               D_h1*(F_vm1+(K_m1^2/R_m1)+
               F_vL1/N_1^2)+
               K_h1*(J_m1+(J_L1/N_1^2)); 
        x1_0 = K_h1*(F_vm1 + (K_m1^2/R_m1)+F_vL1/N_1^2));
        H_ax1 = (K_m1/R_m1)*(J_L1*s^2+F_vL1*s+K_h1)/
                (xi_3*s^3+xi_2*s^2+xi_1*s+xi_0);
    ax1_omega/u_axiservo = H_ax1;
    ax1_angle/u_axiservo = H_ax1 * 1/s;
END RELATION;
END Axis1
                        */
CONTINUOUS SPECIFICATION Axis2
IMPORT u_ax2servo;
EXPORT ax2_angle,ax2_omega;
CONSTANT
    /* Taken from table 4.1 Klomp or from table 2.9 Rieswijk */
\[ K_{m2} = 0.099; \]
\[ R_{m2} = 1.87; \]
\[ N_2 = 101; \]
\[ J_2 = 0.00042; \]
\[ F_{w2} = 0.005; \]

VARIABLE
\[ ax2\_angle : \text{REAL}; \]
\[ ax2\_omega : \text{REAL}; \]

INITIAL
TRUE;

RELATION
CASE
TRUE : \( H_{\text{ax2}} = 1/\text{s}; \)
\[ H_{\text{ax2}} = K_{m2}\times N_2^{-2}/J_2\times R_{m2}\times s + F_{v2}\times R_{m2} + K_{m2}\times N_2^{-2}; \]
\[ ax2\_omega/ax2\_servo = H_{\text{ax2}}; \]
\[ ax2\_angle/ax2\_servo = H_{\text{ax2}} \times 1/\text{s}; \]

END RELATION;
END Axis2

\/* \text{CONTINUOUS SPECIFICATION Axis3} */

\/* Two models are used one models behaviour in contact situations and another in non-contact situations */

IMPORT u_axis3servo, contactz;
EXPORT ax3_pos, ax3_vel;

CONSTANT
\[ \tau_{i} = 0.0177; \] /* See Klomp, table 3.3 */
\[ K_{m} = 0.105; \] /* See Klomp, table 3.1 */
\[ K_{p} = 18.0; \] /* See Klomp, table 3.3 */
\[ B = 0.0004; \] /* See Klomp, table 3.2 */
\[ R = 4.0; \] /* See Klomp, table 3.2 */
\[ k_{e} = 90000; \] /* See Klomp, page 42 */
\[ k_{m} = 0.0029; \] /* See Klomp, table 3.1 */
\[ n = 0.0015; \] /* See Klomp, table 3.2 */
\[ J_{m} = 0.00005; \] /* See Klomp, table 3.1 */
\[ k_{ta} = 0.0208; \] /* See Klomp, table 3.3 */
\[ m_{l} = 6.9; \] /* See Klomp, table 3.2 */
\[ J = J_{m} + m_{l}\times n^{-2}; \]
\[ a = K_{m}\times k_{p}\times k_{ta}\times k_{e}\times n^{-2}\times R\times \tau_{i}; \]
\[ K_{DCvc} = n\times K_{m}\times k_{p}/a; \]
\[ K_{DCvn} = n/k_{ta}; \]

VARIABLE
\[ ax3\_pos : \text{REAL}; \]
\[ ax3\_vel : \text{REAL}; \]

INITIAL
\[ ax3\_pos = \text{init}\_pos; \]

RELATION
CASE
NOT contactz : \( H_{\text{ax3}} = 1; \)
\[ H_{\text{ax3}} = K_{DCvn}(1+\tau_{i}\times s)/ \]
\[ 1+(K_{m}^{-2}+K_{m}\times k_{p}\times k_{ta}\times B\times R\times k_{e}\times L\times n^{-2})/a)\times \tau_{i}\times s+ \]
\[ ((J+2BL)/a)\times \tau_{i}\times s^2+ \]
\[ (J \times L/a)\times \tau_{i}\times s^3; \]
\[ ax3\_vel/ax3\_servo = H_{\text{ax3}}; \]
\[ ax3\_pos/ax3\_servo = H_{\text{ax3}} \times 1/\text{s}; \]

CASE
contactz : \( H_{\text{ax3}} = 1/\text{s}; \)
\[ H_{\text{ax3}} = K_{DCvc}(1+s\times \tau_{i})/ \]
\[ 1+(K_{m}^{-2}+K_{m}\times k_{p}\times k_{ta}\times B\times R\times k_{e}\times L\times n^{-2})/a)\times \tau_{i}\times s+ \]
\[ ((J + R\times B \times L)/a)\times \tau_{i}\times s^2+ \]
\[ (J \times L/a)\times \tau_{i}\times s^3 \]
\[ ]; \]
\[ ax3\_vel/ax3\_servo = H_{\text{ax3}}; \]
ax3_pos/u_ax3servo = H_ax3 * 1/s;

END RELATION;
END Axis3

CONTINUOUS SPECIFICATION Axis4
IMPORT u_ax4servo;
EXPORT ax4_angle, ax4_omega;
CONSTANT
  K_m4 = 0.056;  /* See Klomp, table 4.2, p. 79 */
  R_m4 = 2.3;  /* See Klomp, table 4.2, p. 79 */
  L_m4 = 0.0048;  /* See Klomp, table 4.2, p. 79 */
  J_m4 = 0.000051;  /* See Klomp, table 4.2, p. 79 */
  B_m4 = 0.00039;  /* See Klomp, table 4.2, p. 79 */
  n_4 = 0.03;  /* See Klomp, table 4.2, p. 79 */
  k_ta4 = 0.0281;  /* See Klomp, table 4.3, p. 82 */
  k_p4 = 11.6;  /* See Klomp, table 4.3, p. 82 */
  tau_i4 = 0.025;  /* See Klomp, table 4.3, p. 82 */

VARIABLE
  ax4_angle : REAL;
  ax4_omega : REAL;
INITIAL
  TRUE;
RELATION
CASE
  /* See block diagram of fig. 3.2, 3.5, Eq. 3.3 of Klomp */
  TRUE : H_ax4 = n_4*K_m4*k_p4*(1+tau_i4*s)/
    (K_m4*k_p4*k_ta4+ (K_m4^2*K_m4*k_p4*k_ta4+B_m4*R_m4)*tau_i4*s+
      (J_m4*R_m4+B_m4*L_m4)*tau_i4*s^2+
      J_m4*L_m4*tau_i4*3*s^3
    );
  ax4_angle/u_ax4servo = H_ax4;
  ax4_omega/u_ax4servo = H_ax4 * 1/s;
END RELATION;
END Axis4

CONTINUOUS SPECIFICATION Force
/* Contarearea is specified through specification of x,y */
/* and z coordinates. See master thesis van Gimst. Expression for */
/* disturbance torques (Klomp, p. 86) have been omitted */
IMPORT ax1_angle, ax2_angle, ax3_pos;
EXPORT contactx, contacty, contactz, f_x, f_y, f_z;
CONSTANT
  /* Constants that specify location of object in Environment */
  obj_xpos = 0;
  obj_ypos = 0;
  obj_zpos = 0;
  k_ex = 90000;  /* == 90.10^-3 */
  k_ey = 90000;
  k_ez = 90000;
  /* Assume stiff object, actual stiffness is situation dependent */
  see Klomp, p. 42 and 57 */
VARIABLE
  contactx : BOOLEAN;
  f_x : REAL;
  x_pos : REAL;
  contacty : BOOLEAN;
  f_y : REAL;
  y_pos : REAL;
  contactz : BOOLEAN;
  f_z : REAL;
INITIAL
  TRUE;
RELATION
APPENDIX C. ROBOT CONTROL SYSTEM SPECIFICATION

/*
 contactz = (ax3_pos <= obj_zpos);
 contacty = (y_pos >= obj_ypos);
 contactx = (x_pos >= obj_xpos);
 */

CASE /* contactz & contacty & contactx */
{ax3_pos <= obj_zpos} &
{ y_pos >= obj_ypos} &
{ x_pos >= obj_xpos} :
    contactz=TRUE; contacty=TRUE; contactx=TRUE;
    x_pos = 1.2*cos(ax1_angle+ax2_angle) + 1.1*cos(ax1_angle);
    y_pos = -1*1.2*sin(ax1_angle+ax2_angle) - 1.1*sin(ax1_angle);
    f_z = k_ez*(obj_zpos-ax3_pos);
    f_y = k_ey*(y_pos - obj_ypos);
    f_x = k_ex*(x_pos-obj_xpos);
CASE /* NOT contactz & contacty & contactx */
{ax3_pos > obj_zpos} &
{ y_pos >= obj_ypos} &
{ x_pos >= obj_xpos} :
    contactz=FALSE; contacty=TRUE; contactx=TRUE;
    x_pos = 1.2*cos(ax1_angle+ax2_angle) + 1.1*cos(ax1_angle);
    y_pos = -1*1.2*sin(ax1_angle+ax2_angle) - 1.1*sin(ax1_angle);
    f_z = 0;
    f_y = k_ey*(y_pos - obj_ypos);
    f_x = k_ex*(x_pos-obj_xpos);
CASE /* contactz & NOT contacty & contactx */
{ax3_pos <= obj_zpos} &
{ y_pos < obj_ypos} &
{ x_pos >= obj_xpos} :
    contactz=TRUE; contacty=FALSE; contactx=TRUE;
    x_pos = 1.2*cos(ax1_angle+ax2_angle) + 1.1*cos(ax1_angle);
    y_pos = -1*1.2*sin(ax1_angle+ax2_angle) - 1.1*sin(ax1_angle);
    f_z = k_ez*(obj_zpos-ax3_pos);
    f_y = 0;
    f_x = k_ex*(x_pos-obj_xpos);
CASE /* NOT contactz & NOT contacty & contactx */
{ax3_pos > obj_zpos} &
{ y_pos < obj_ypos} &
{ x_pos >= obj_xpos} :
    contactz=FALSE; contacty=FALSE; contactx=TRUE;
    x_pos = 1.2*cos(ax1_angle+ax2_angle) + 1.1*cos(ax1_angle);
    y_pos = -1*1.2*sin(ax1_angle+ax2_angle) - 1.1*sin(ax1_angle);
    f_z = 0;
    f_y = 0;
    f_x = k_ex*(x_pos-obj_xpos);
CASE /* contactz & contacty & NOT contactx */
{ax3_pos <= obj_zpos} &
{ y_pos >= obj_ypos} &
{ x_pos < obj_xpos} :
    contactz=TRUE; contacty=TRUE; contactx=FALSE;
    x_pos = 1.2*cos(ax1_angle+ax2_angle) + 1.1*cos(ax1_angle);
    y_pos = -1*1.2*sin(ax1_angle+ax2_angle) - 1.1*sin(ax1_angle);
    f_z = k_ez*(obj_zpos-ax3_pos);
    f_y = k_ey*(y_pos - obj_ypos);
    f_x = 0;
CASE /* NOT contactz & contacty & NOT contactx */
{ax3_pos > obj_zpos} &
{ y_pos >= obj_ypos} &
{ x_pos < obj_xpos} :
contactz=FALSE; contacty=TRUE; contactx=FALSE;
x_pos = l_2*cos(ax1_angle+ax2_angle) + l_1*cos(ax1_angle);
y_pos = -1*l_2*sin(ax1_angle+ax2_angle) - l_1*sin(ax1_angle);
f_z = 0;
f_y = k_ey*(y_pos - obj_ypos);
f_x = 0;
CASE /* contactz & NOT contacty & NOT contactx */
  {ax3_pos <= obj_zpos} &
  { y_pos < obj_ypos} &
  { x_pos < obj_xpos} :
    contactz=TRUE; contacty=FALSE; contactx=FALSE;
x_pos = l_2*cos(ax1_angle+ax2_angle) + l_1*cos(ax1_angle);
y_pos = -1*l_2*sin(ax1_angle+ax2_angle) - l_1*sin(ax1_angle);
f_z = k_ez*(obj_zpos-ax3_pos);
f_y = 0;
f_x = 0;
CASE /* NOT contactz & NOT contacty & NOT contactx */
  {ax3_pos > obj_zpos} &
  { y_pos < obj_ypos} &
  { x_pos < obj_xpos} :
    contacty=FALSE; contactz=FALSE; contactx=FALSE;
x_pos = l_2*cos(ax1_angle+ax2_angle) + l_1*cos(ax1_angle);
y_pos = -1*l_2*sin(ax1_angle+ax2_angle) - l_1*sin(ax1_angle);
f_z = 0;
f_y = 0;
f_x = 0;
END RELATION;
END Force

SPECIFICATION Ax1_Sensor
/*/ Sensor for axis 1 speed and position */
IMPORT ax1_angle,ax1_omega,T_sample;
EXPORT ax1_angle,ax1_vel, Read_Ax1angle;
CONSTANT T_axis = 0.0001; /* == T_sample/10 */

VARIABLE
  axis1_angle : REAL;
  axis1_vel : REAL;
INITIAL
  TRUE;
TRANSITION Read_Ax1angle T_axis
PRE
  Start(Read_Ax1angle,1) + T_sample = now | now = 0
POST
  axis1_vel = ax1_omega & axis1_angle=ax1_angle
END Ax1_Sensor;

SPECIFICATION Ax1_Actuator
IMPORT cu_axis servo, T_sample;
EXPORT u_axis servo;
CONSTANT
  T_oax1 = 0.0001; /* == T_sample/10 */
VARIABLE
  u_axis servo : REAL;
INITIAL
  u_axis servo=0;
TRANSITION Output T_oax1
PRE
  cu_axis servo <> u_axis servo
POST
  u_axis servo = cu_axis servo
END Ax1_actuator;

SPECIFICATION Ax2_Sensor
APPENDIX C. ROBOT CONTROL SYSTEM SPECIFICATION

/* Sensor for axis 2 speed and position */
IMPORT ax2_angle, ax2_omega, T_sample;
EXPORT ax2s_angle, ax2s_vel, Read_Ax2angle;
CONSTANT
T_ax2s = 0.0001; /* == T_sample/10 */
VARIABLE
ax2s_angle : REAL;
ax2s_vel : REAL;
INITIAL
TRUE;
TRANSITION Read_Ax2angle T_ax2s
PRE
Start(Read_Ax2angle)+T_sample=now | now=0
POST
ax2s_vel = ax2_omega & ax2s_angle = ax2_angle
END Ax2_Sensor;

SPECIFICATION Ax2_Actuator
IMPORT cu_ax2servo, T_sample;
EXPORT u_ax2servo;
CONSTANT
T_oax2 = 0.0001; /* == T_sample/10 */
VARIABLE
u_ax2servo : REAL;
INITIAL
u_ax2servo=0;
TRANSITION Output T_oax2
PRE
u_ax2servo <> cu_ax2servo
POST
u_ax2servo = cu_ax2servo
END Ax2_actuator;

SPECIFICATION Ax3_Sensor
/* Sensor for axis 3 speed and position */
IMPORT ax3_pos, ax3_vel, T_sample;
EXPORT ax3s_pos, ax3s_vel, Read_Ax3pos;
CONSTANT
T_ax3s = 0.0001; /* == T_sample/10 */
VARIABLE
ax3s_pos : REAL;
ax3s_vel : REAL;
INITIAL
TRUE;
TRANSITION Read_Ax3pos T_ax3s
PRE
Start(Read_Ax3pos)+T_sample=now & now=0
POST
ax3s_vel = ax3_vel | ax3s_pos=ax3pos
END Ax3_Sensor;

SPECIFICATION Ax3_Actuator
IMPORT cu_ax3servo, T_sample;
EXPORT u_ax3servo;
CONSTANT
T_oax3 = 0.0001; /* == T_sample/10 */
VARIABLE
u_ax3servo : REAL;
INITIAL
u_ax3servo=0;
TRANSITION Output T_oax3
PRE
cu_ax3servo <> u_ax3servo
POST
u_ax3servo = cu_ax3servo
END Ax3_actuator;
SPECIFICATION Ax4_Sensor
/* Sensor for axis 4 speed and position */
IMPORT ax4_angle,ax4_omega,T_sample;
EXPORT ax4s_angle,ax4s_vel,Read_Ax4angle;
CONSTANT
  T_ax4s = 0.0001; /* == T_sample/10 */
VARIABLE
  ax4s_angle : REAL;
  ax4s_vel : REAL;
INITIAL
  TRUE;
TRANSITION Read_Ax4angle T_ax4s
PRE
  Start(Read_Ax4angle)+T_sample=now | now=0
POST
  ax4s_vel = ax4_omega & ax4s_angle = ax4_angle
END Ax4_Sensor;
/* =================================================================== */
SPECIFICATION Ax4_Actuator
IMPORT cu_ax4servo, T_sample;
EXPORT u_ax4servo;
CONSTANT
  T_oax4 = 0.0001; /* == T_sample/10 */
VARIABLE
  u_ax4servo : REAL;
INITIAL
  u_ax4servo=0;
TRANSITION Output T_oax4
PRE
  u_ax4servo <> cu_ax4servo
POST
  u_ax4servo = cu_ax4servo
END Ax4_actuator;
/* =================================================================== */
SPECIFICATION F_Sensor
IMPORT f_x,f_y,f_z,T_sforce;
EXPORT fs_x,fs_y,fs_z,Read_Force;
CONSTANT
  T_rf = 0.0001; /* == T_sforce/10 */
VARIABLE
  fs_x : REAL;
  fs_y : REAL;
  fs_z : REAL;
INITIAL
  TRUE;
TRANSITION Read_Force T_rf
PRE
  Start(Read_Force)+T_sforce=now | now=0
POST
  /* The force sensor has a delay of 0.010 (s), see p. 55 of Klomp */
  fs_x = Past(f_x,0.010) & fs_y = Past(f_y,0.010) &
  fs_z = Past(f_z,0.010)
END F_Sensor;
/* =================================================================== */
SPECIFICATION Gripper
IMPORT gr_command, Gr_State;
EXPORT u_gr;
CONSTANT
  T_gr : Time;
  T_cg : Time;
  OV = 1; /* Gripper Open Value */
  CV = 0; /* Gripper Close value */
VARIABLE
  u_gr : REAL;
  gripper_state : Gr_State;
INITIAL
  u_gr = OV & gripper_state = Opened;
TRANSITION Open  T_{\text{og}}
PRE
gr\_command = Open
POST
u\_gr = \text{OV} & \text{Gr\_State = Opened}
TRANSITION Close  T_{\text{cg}}
PRE
gr\_command = Close
POST
u\_gr = \text{CV} & \text{gripper\_state = Closed}
END Gripper;

SPECIFICATION Axl\_Control

IMPORT
axlnewsetpoint,axlsetpoint,
cax2setpoint,ax2phi,
axidesmode,axiconsumset, /* From Axl\_2-Control-manager */
axis\_angle,Read\_Axialangle, /* From Axl\_Sensor */
tm\_alarm, /* From Task-Manager */
T\_sample,ABS,CtrModeType,
P\_Contr,PI\_Contr,Limiter,
L1,L2;
EXPORT cu\_axis\_servo,axlsetprequest,ax1phi;

CONSTANT
/* Value of K_{ic} and K_{ij} are not given in literature
 and have been defined to equal 10 here */
T\_cm : Time; T\_al : Time; T\_rp : Time;
T\_cp : Time; T\_cpi : Time;
T\_wr : Time; T\_sp : Time; T\_jp : Time;
T\_jpi : Time; T\_cv : Time; T\_cf : Time;
Max\_React\_Time = 5;
K_{pc} = 75; /* See Klomp p. 93 */
K_{ic} = 10;
Idist = 0.004; /* Idist = (Radians), See Klomp, p. 89 */
K_{pj} = 46; K_{ij} = 10; /* See Klomp p. 89 */
K_{pv} = 100; /* See Klomp p. 98 */
K_{pfi} = 75; /* See Klomp p. 104 and p. 93 */
Max\_Slope = 80; /* Max\_Slope = (V/s), See Klomp p. 89 */

FUNCTION InvJacPos\_Ctr(desx:REAL;desy:REAL;pax1phi:REAL;
pax2phi:REAL;Kp:REAL]):REAL
VAR actx : REAL; acty : REAL; errx : REAL;
err : REAL; phi1 : REAL;
BEGIN
actx := L2*sin(pax1phi)*pax2phi + L1*sin(pax1phi);
acty := -1*L2*sin(pax1phi)*pax2phi - L1*sin(pax1phi);
errx := desx - actx;
err := desy - acty;
phi1 := L2*cos(pax1phi)*pax2phi)*errx-L2*sin(pax1phi)*pax2phi)*errx-
phi1 := 1/(L1*L2*sin(pax2phi))*phi1;
phi1 := Kp*phi1;
RETURN phi1;
END

INVARIANT caxlmode = Stopped | caxlmode=JointPos | caxlmode=CartPos |
caxlmode = Cart\_Vel | caxlmode=Force;
/* caxlmode ISIN \{Stopped,JointPos,CartPos,\text{Cart\_Vel,Force}\} */
/* Only the velocity control part of the Force Control algorithm
 is specified in this control process */

SCHEDULE
/* Upper (time) bound specified for reacting upon alarm */
FORALL ti:Time;j:INTEGER;
\[(\exists k : \text{INTEGER}; (\text{Start}(\text{Ax1\_Alarm}, j) = t_1 \& \text{now} > t_1 + \text{Max\_React\_Time}) \Rightarrow
\{ \text{End}(\text{Ax1\_Stop}, k) \geq t_1 \& \text{End}(\text{Ax1\_Stop}, k) \leq t_1 + \text{Max\_React\_Time} \})\)\]

\begin{align*}
\text{VARIABLE} & \\
\text{axiphi} & : \text{REAL} ; \\
\text{micontrolval} & : \text{REAL} ; \\
\text{axisetprequest} & : \text{BOOLEAN} ; \\
\text{caxiwrite} & : \text{BOOLEAN} ; \\
\text{cu\_axisservo} & : \text{REAL} ; \\
\text{caximode} & : \text{CtrModeType} ; \\
\text{error} & : \text{REAL} ; \\
\text{INITIAL} & \\
\text{cu\_axisservo} = 0 \& \text{NOT axisetprequest} \& \text{caximode} = \text{Stopped} ; \\
\text{TRANSITION} & \text{Change\_Ctrmode T\_cm} \\
& /* \text{Mode-Change} */ \\
\text{PRE} & \text{caximode} \leftrightarrow \text{axidesmode} \& \text{NOT tmalarm} \& \text{axiconsmet} \\
\text{POST} & \text{caximode} = \text{axidesmode} \& \text{NOT axisetprequest} \& \text{error} = 0 \& \text{micontrolval} = 0 \\
& \text{TRANSITION Ax1\_Alarm T\_al} \\
& \text{PRE} \text{tmalarm} \& \text{caximode} \leftrightarrow \text{Stopped} \\
& \text{POST} \text{caximode} = \text{Stopped} \\
& \text{TRANSITION ReadPhi T\_rp} \\
& \text{PRE} \text{caximode} \leftrightarrow \text{Stopped} \& \text{NOT axinewsetpoint} \& \text{EXISTS j:INTEGER; (End}(\text{Read\_Axangle}, j)) = t \& \text{NOT}(\text{EXISTS k:INTEGER; (t < Start}(\text{ReadPhi}, k)) \}
\end{align*}

\begin{align*}
\text{POST} & \text{axiphi} = \text{axis\_angle} \& \text{axisetprequest} \\
& \text{TRANSITION Ctr\_ValCartPosP T\_cp} \\
& \text{PRE} \text{axinewsetpoint} \& \text{caximode} = \text{CartPos} \& \text{ABS}(\text{axisetpoint} - \text{axiphi}) > \text{Idist} \\
& \text{POST} \text{micontrolval} = \text{PI\_Contr}(K\_pc, \text{axisetpoint}, \text{axiphi}, \text{micontrolval}, \text{error}) \& \text{NOT axisetprequest} \& \text{caxiwrite} \\
& \text{TRANSITION Ctr\_ValCartPosPI T\_cpi} \\
& \text{PRE} \text{axinewsetpoint} \& \text{caximode} = \text{CartPos} \& \text{ABS}(\text{axisetpoint} - \text{axiphi}) <= \text{Idist} \\
& \text{POST} \text{micontrolval} = \text{PI\_Contr}(K\_pj, \text{axisetpoint}, \text{axiphi}, \text{micontrolval}, \text{error}, \text{T\_sample}) \& \text{NOT axisetprequest} \& \text{caxiwrite} \\
& \text{TRANSITION Ctr\_ValJointPosP T\_jp} \\
& \text{PRE} \text{axinewsetpoint} \& \text{caximode} = \text{JointPos} \& \text{ABS}(\text{axisetpoint} - \text{axiphi}) > \text{Idist} \\
& \text{POST} \text{micontrolval} = \text{PI\_Contr}(K\_pj, \text{axisetpoint}, \text{axiphi}, \text{micontrolval}, \text{error}) \& \text{NOT axisetprequest} \& \text{caxiwrite} \\
& \text{TRANSITION Ctr\_ValJointPosPI T\_jpi} \\
& \text{PRE} \text{axinewsetpoint} \& \text{caximode} = \text{JointPos} \& \text{ABS}(\text{axisetpoint} - \text{axiphi}) <= \text{Idist} \\
& \text{POST} \text{micontrolval} = \text{PI\_Contr}(K\_pj, K\_ij, \text{axisetpoint}, \text{axiphi}, \text{micontrolval}, \text{error}, \text{T\_sample}) \& \text{NOT axisetprequest} \& \text{caxiwrite} \\
& \text{TRANSITION Ctr\_ValCartVelP T\_cv} \\
& \text{PRE} \text{axinewsetpoint} \& \text{caximode} = \text{CartVel}
POST
  micontrolval = P_Contr(K_pv,caxisetpoint,axiphi,micontrolval,error)
  & NOT axissetrequest & caxiwrite
TRANSITION Ctr_ValForce       T_cf
PRE
  axinewsetpoint & caximode = Force
POST
  micontrolval = InvJacPos_CtrAx1(caxisetpoint,cax2setpoint,axiphi,
       ax2phi,K_pf1) & NOT axissetrequest & caxiwrite
TRANSITION Write            T_wr
PRE
  caxiwrite & caximode <> Stopped
POST
  cu_axiservo = Limiter(Max_Slope,T_sample,micontrolval,cu_axiservo) &
                 NOT caxiwrite
TRANSITION Ax1_Stop         T_st
PRE
  caximode = Stopped
POST
  cu_axiservo = 0
END Ax1_Control;
/*============================================================================ */
SPECIFICATION Ax2_Control
/* Maximum Velocity in mode Cartvel is currently not restricted */
IMPORT
  ax2newsetpoint,cax2setpoint,
  caxisetpoint,
  ax2dsmode,ax2consmet,          /* From Ax1_2-Control-manager */
  ax2s_angle,Read_Ax2angle,     /* From Ax2_Sensor */
  tm_alarm,                     /* From Task-Manager */
  T_sample,ABS,CtrlModeType,
  P_Contr,P1_Contr,Limiter,
  L1,L2,sin,cos;
EXPORT cu_ax2servo,ax2setprequest,ax2phi;
CONSTANT
  /* Value of K_ic could not be resolved from literature it has been defined equal to 10 */
  T_cm : Time; T_al : Time; T_rp : Time;
  T_cp : Time; T_cpi : Time; T_jp : Time;
  T_jpi : Time; T_cv : Time; T_wr : Time;
  T_st : Time; T_cf : Time;
  Max_React_Time = 5;
  K_pc = 64.5; K_ic = 10;     /* See Klomp p. 93 */
  K_pj = 42;                   /* See Klomp p. 89 */
  K_pv = 96;                   /* See Klomp p. 98 */
  K_pf2 = 64.5;                /* See Klomp p. 104 and p. 93 */
  Max_Slope = 80;              /* See Klomp p. 89 */
  I_dist = 0.004;              /* See Klomp, fig. 4.14,4.19,4.20 */
DEFINITION
FUNCTION InvJacPos_CtrAx2(desx:REAL;desy:REAL;pax1phi:REAL;
                           pax2phi:REAL;Kp:REAL;):REAL
VAR
  actx : REAL; acty : REAL;
  errx : REAL; erry : REAL;
BEGIN
  actx := L2*cos(pax1phi+pax2phi) + L1*cos(pax1phi);
  acty := -1*L2*sin(pax1phi+pax2phi) - L1*sin(pax1phi);
  errx := desx - actx;
  erry := desy - acty;
  pax2phi := (-1*L2*cos(pax1phi+pax2phi)-L1*cos(pax1phi))*errx;
  pax2phi := pax2phi+L1*sin(pax1phi)+L2*sin(pax1phi+pax2phi))*erry;
  pax2phi := (1/L1*L1*sin(pax2phi)))*pax2phi;
  pax2phi := Kp*pax2phi;
RETURN pax2phi;
END

IN Variant cax2mode=Stopped | cax2mode=JointPos | cax2mode=CartPos | cax2mode=CartVel | cax2mode=Force;

/* ax2mode ISIN {Stopped,JointPos,CartPos,CartVel,Force}; */
/* Only the velocity control part of the Force-control algorithm is specified in this control process */

SCHEDULE

/* Upper (time) bound specified for reacting upon alarm */
FOR ALL t2:Time;j:INTEGER;
(EXISTS k:INTEGER( { Start(Ax2_Alarm,j)=t2 &
now > t2+Max_React_Time } =>
{ End(Ax2_Stop,k) >= t2 &
   End(Ax2_Stop,k) <= t2+Max_React_Time } )
);

/* End(Ax2_Stop,k) ISIN [t,t+Max_React_Time] */

VARIABLE
ax2phi : REAL;
m2controloval : REAL;
ax2setprequest : BOOLEAN;
cax2write : BOOLEAN;
cu_ax2servo : REAL;
cax2mode : CtrModeType;
error : REAL;

INITIAL
   cu_ax2servo = 0 & NOT ax2setprequest & cax2mode = Stopped;
TRANSITION Change_Ctrmode T_cm
/* Mode-Change */
PRE
ax2mode <> ax2desmode & NOT tmalarm & ax2consmet
POST
ax2mode = ax2desmode & NOT ax2setprequest &
error = 0 & m2controloval = 0
TRANSITION Ax2_Alarm T_al
PRE
tmalarm & ax2mode <> Stopped
POST
ax2mode = Stopped
TRANSITION ReadPhi T_rp
PRE
ax2mode <> Stopped & NOT ax2newsetpoint &
(EXISTS j:INTEGER(End(Read_Ax2angle,j)=t &
   NOT(EXISTS k:INTEGER((t < Start(ReadPhi,k))))
))
POST
ax2phi = ax2s_angle & ax2setprequest
TRANSITION Ctr_ValCartPosP T_cp
PRE
ax2newsetpoint & ax2mode = CartPos & ABS(cax2setpoint-ax2phi) > Idist
POST
m2controloval = P_Contr(K_pc,cax2setpoint,ax2phi,m2controloval,error) &
NOT ax2setprequest & cax2write
TRANSITION Ctr_ValCartPosPI T_cpi
PRE
ax2newsetpoint & ax2mode = CartPos & ABS(cax2setpoint-ax2phi) <= Idist
POST
m2controloval = PI_Contr(K_pc,K_ic,cax2setpoint,ax2phi,m2controloval,
error,T_sample) & NOT ax2setprequest & cax2write
TRANSITION Ctr_ValJointPosP T_jp
PRE
ax2newsetpoint & ax2mode = JointPos & ABS(cax2setpoint-currphi)>Idist
POST
m2controloval = P_Contr(K_pj,cax2setpoint,ax2phi,m2controloval,error) &
NOT ax2setprequest & cax2write
TRANSITION Ctr_ValJointPosPI T_jpi
PRE
   ax2newsetpoint & ax2mode = JointPos & ABS(cax2setpoint-currphi)<=Idist
POST
   m2controloval = PI_Contr(K_pj,K_ij,cax2setpoint,ax2phi,m2controloval, 
   error,T_sample) & NOT ax2setprequest & cax2write
TRANSITION Ctr_ValCartVelP   T_cv
PRE
   ax2newsetpoint & ax2mode = Jointvel
POST
   m2controloval = P_Contr(K_pv,cax2setpoint,ax2phi,m2controloval,error) & 
   NOT ax2setprequest & cax2write
TRANSITION Ctr_ValForce   T_cf
PRE
   ax2newsetpoint & ax2mode = Force
POST
   m2controloval = InvJacPos_CtrAx2(caxisetpoint,cax2setpoint,ax1phi, 
   ax2phi,K_pf2) & NOT ax1setprequest & caxwrite
TRANSITION Write   T_wr
PRE
   cax2write & ax2mode <> Stopped
POST
   cu_ax2servo = Limiter(Max_Slope,T_sample,m2controloval,cu_axiservo) & 
   NOT cax2write
TRANSITION Ax2_Stop   T_st
PRE
   ax2mode = Stopped
POST
   cu_ax2servo = 0
END Ax2_Control;

/* ==--------------------------------------------------------------------- */
SPECIFICATION Ax3_Control
/* Maximum Velocity in mode Cartvel is currently not restricted */
IMPORT
   ax3newsetpoint,cax3setpoint,
   ax3dismode,ax3consnet,   /* From Ax3-Control-manager */
   ax3s_pos,Read_Ax3pos,   /* From Ax3-Sensor */
   ts_z,Read_Force,        /* From F_Sensor */
   talarm, /* From Task-Manager */
   T_sample,ABS,CtrModeType,
   P_Contr,PI_Contr,Limiter;
EXPORT cu_ax3servo,a3setprequest,cax3pos,a3force;
CONSTANT
   /* Value of K_ic could not be resolved from literature it 
      has been defined equal to 10 */
   T_ccm : Time; T_re : Time; T_rf : Time;
   T_cp : Time; T_cpi : Time; T_cf : Time;
   T_zcv : Time; T_al : Time; T_st : Time;
   K_pc = 200; /* See Klomp, p. 52 */
   K_pv = 400; /* See Klomp, p. 54 */
   K_ic = 10;
   K_pf = 500; /* See Klomp, p. 58 */
   Idist = 0.004; /* See Klomp, p. 51 */
   Max_Slope = 80; /* See Klomp, p. 50 */
   Max_React_Time = 5;
INVARIANT cax3mode=Stopped | cax3mode=CartPos | cax3mode=CartVel |
   cax3mode=Force;
   /* cax3mode ISIN {Stopped,CartPos,CartVel,Force} */
SCHEDULE
   /* Upper (time) bound specified for reacting upon alarm */
   FORALL t3:Time;j:INTEGER;
   ( EXISTS k:INTEGER;( { Start(Ax3_Alarm,j)=t3 & 
       now > t3+Max_React_Time) 
       => /* (End(Ax3_Stop,k) ISIN [t,t+Max_React_Time]) */ */
*/
{ End(Ax3_Stop, k) >= t3 &
  End(Ax3_Stop, k) <= t3 + Max_React_Time
}

VARIABLE
cax3pos : REAL;
ax3force : REAL;
m3controlval : REAL;
ax3setprequest : BOOLEAN;
cax3fread : BOOLEAN;
cax3write : BOOLEAN;
cu_ax3servo : REAL;
cax3mode : CtrModeType;
error : REAL;

INITIAL
cu_ax3servo = 0 & NOT ax3setprequest & cax3mode = Stopped;

TRANSITION Change_Ctrmode T_ccm
/* Mode-Change */
PRE cax3mode <> ax3desmode & NOT tmalarm & ax3consmet
POST cax3mode = ax3desmode & NOT ax3setprequest & error = 0 &
m3controlval = 0

TRANSITION Ax3_Alarm T_al
PRE tmalarm & cax3mode <> Stopped
POST cax3mode = Stopped

TRANSITION Readpos T_re
PRE { cax3mode = CartPos | cax3mode = CartVel } & NOT ax3newsetpoint &
  EXISTS t4:Time; j:INTEGER; (End(Read_Ax3pos,j) = t4
  & NOT(EXISTS k:INTEGER; (t4 < Start(Readpos,k))
}

POST cax3pos = ax3s_pos & ax3setprequest

TRANSITION Readforce T_rf
PRE cax3mode = Force &
  EXISTS j:INTEGER; (End(Read_Force, j) = t
  & NOT(EXISTS k:INTEGER; (t < Start(Readforce,k))
}

POST ax3force = fs_z & ax3setprequest

TRANSITION Ctr_ValCartPos P T_cp
PRE ax3newsetpoint & cax3mode = CartPos & ABS(ax3setpoint - cax3pos) > Idist
POST m3controlval = P_Contr(K_pc, ax3setpoint, cax3pos, m3controlval, error) &
  NOT ax3setprequest & cax3write

TRANSITION Ctr_ValCartPosP T_cpi
PRE ax3newsetpoint & cax3mode = CartPos & ABS(ax3setpoint - cax3pos) <= Idist
POST m3controlval = PI_Contr(K_pc, K_ic, ax3setpoint, cax3pos, m3controlval,
  error, T_sample) & NOT ax3setprequest & cax3write

TRANSITION Ctr_ValCartVel P T_zcv
PRE ax3newsetpoint & cax3mode = CartVel
POST m3controlval = P_Contr(K_pv, ax3setpoint, cax3pos, m3controlval,
  error, T_sample) & NOT ax3setprequest & cax3write

TRANSITION Ctr_ValForceP T_cf
PRE cax3mode = Force & ax3fread
POST
m3controlval = P_Contr(K_pf, cax3setpoint, ax3force, m3controlval, error) &
NOT ax3freed & cax3write
TRANSITION Write        T_wr
PRE
   cax3write & cax3mode => Stopped
POST
   cu_ax3servo = Limiter(Max_Slope,T_sample,m3controlval,cu_ax3_servo') &
   NOT cax3write
TRANSITION Ax3_Stop    T_st
PRE
   cax3mode = Stopped
POST
   cu_ax3servo = 0
END Ax3_Control;

/* Maximum Velocity in mode Cartvel is currently not restricted */
IMPORT
ax4newsetpoint, ax4setpoint,
ax4desmode,ax4consmet,   /* From Ax4-Control-manager */
ax4s_angle, Read_Ax4angle,   /* From Ax4-Sensor */
fs_x,fs_y, Read_Force,       /* From F_sensor */
tmalarm,                    /* From Task-Manager */
T_sample, ABS,CtrModeType,
P_Contr,PI_Contr,Limiter;
EXPORT cu_ax4servo,ax4setprequest,ax4phi;
CONSTANT
   /* Value of K_ij2 and b_0 could not be resolved from literature
    * they have been defined equal to 10 */
   T_ccm : Time; T_al : Time; T_rp : Time;
   T_rf : Time; T_cp : Time; T_cpi : Time;
   T_cf : Time; T_wr : Time; T_st : Time;
   Max_React_Time = 5;
   K_pj = 14; K_ij2 = 5;   /* See Klomp, p. 87 */
   K_f = 40;                /* See Klomp, p. 102 */
   b_0 = 10;                /* See Klomp, fig. 4.29, eq. 4.18, p. 101 */
   Max_Slope = 80;          /* (V/s) See Klomp, p. 87 */
   L_dist = 0.003;          /* (rad) See Klomp, p. 87 */
INVARIANT cax4mode=Stopped | cax4mode=JointPos | cax4mode=Jointvel | cax4mode=Force;
   /* cax4mode ISIN {Stopped,JointPos,Jointvel,Force} */
SCHEDULE
   /* Upper (time) bound specified for reacting upon alarm */
FORALL t5:Time;j:INTEGER;
   (EXISTS k:INTEGER;{( Start(Ax4_Alarm,j)=t5 &
      now > t5+Max_React_Time }
    =>
    { End(Ax4_Stop,k) >= t5 &
      End(Ax4_Stop,k) <= t5+Max_React_Time }
   )
   /* End(Ax4_Stop,k) ISIN [t,t+Max_React_Time] */
);

VARIABLE
ax4phi : REAL;
currrxforce : REAL;
curryforce : REAL;
m4controlval : REAL;
ax4setprequest : BOOLEAN;
cax4freed : BOOLEAN;
cax4write : BOOLEAN;
cu_ax4servo : REAL;
cax4mode : CtrModeType;
error : REAL;
INITIAL
cu_ax4servo = 0 & NOT ax4setprequest & cx4mode = Stopped;
TRANSITION Change_Ctr mode T_ccm
PRE ax4mode <> cx4desmode & NOT tmalarm & ax4consmet
POST ax4mode = cx4desmode & NOT ax4setprequest &
error = 0 & m4controlval = 0
TRANSITION Ax4_Alarm T_al
PRE tmalarm & cx4mode <> Stopped
POST cx4mode = Stopped
TRANSITION Readphi T_rp
PRE cx4mode <> Stopped & NOT ax4newsetpoint &
EXISTS t6:Time;j:INTEGER;(End(Read_Ax4angle,j)=t6
 & NOT(EXISTS k:INTEGER;(t6 < Start(Readphi,k)))
)
POST ax4phi = ax4s_angle & ax4newsetpoint
TRANSITION Readforce T_rf
/* Essentially this is a torque-controller see p. 100 e.v. of Klomp */
PRE cx4mode = Force &
EXISTS t7:Time;j:INTEGER;(End(Read_Force,j)=t7
 & NOT(EXISTS k:INTEGER;(t7 < Start(Readforce,k)))
)
POST currxforce = fs_x & curryforce = fs_y & ax4fread
TRANSITION Ctr_ValJointPosP T_cp
PRE ax4newsetpoint & cx4mode = JointPos & ABS(ax4setpoint-ax4phi) > Idist
POST m4controlval = P_contr(K_pj,ax4setpoint,ax4phi,m4controlval,error) &
NOT ax4setprequest & cx4write
TRANSITION Ctr_ValJointPosPI T_cpi
PRE ax4newsetpoint & cx4mode = JointPos & ABS(ax4setpoint-ax4phi)<=Idist
POST m4controlval = PI_Contr(K_pj,K_ij2,ax4setpoint,ax4phi,m4controlval,
error,T_sample) & NOT ax4setprequest & cx4write
TRANSITION Ctr_ValForce T_cf
/* This is essentially a torque controller, see Klomp p. 102 */
PRE cx4mode = Force & ax4fread & ax4newsetpoint
POST m4controlval=P_Contr(K_f,ax4setpoint,currxforce*bx,m4controlval,error)
TRANSITION Write T_wr
PRE cx4write & cx4mode <> Stopped
POST cu_ax4servo = Limiter(Max_Slope,T_sample,m4controlval,cu_ax4servo')
& NOT cx4write
TRANSITION Ax4_Stop T_st
PRE cx4mode = Stopped
POST cu_ax4servo = 0
END Ax4_Control;
/* ============================================================== */
SPECIFICATION Ax12_Manager
IMPORT
ax1setprequest,ctskax1setpoint,
ctskax1mode,a1ttskwaiting,ctskax1cons,
a2setprequest,ctskax2setpoint,
ctskax2mode,a2ttskwaiting,ctskax2cons,
ax1s_angle, ax2s_angle, fs_x, fs_y,
flip_grant, Limiter, CtrModeType, t_sample,
T_fsample, L1, L2, acos, atan2, sqrt,
ManStateType, ManModeType;

EXPORT
ax1newsetpoint, cax1setpoint,
ax1desmode, ax1consmet,
ax2newsetpoint, cax2setpoint,
ax2desmode, ax2consmet,
req_flip,
x_coor, y_coor, x_vel, y_vel;

CONSTANT
T_ns : Time; T_rd : Time; T_et : Time;
T_st : Time; T_n1 : Time; T_n2 : Time;
T_cl : Time; T_c2 : Time; T_dp : Time;
T_cp : Time; T_fp : Time; T_rf : Time;
T_fd : Time; T_fg : Time; T_se : Time;
T_cf : Time; T_nf : Time; T_n3 : Time;
K_fx = 20; K_fy = 20; /* See Klomp, page 105 */
K_e = 40; /* See figure 4.33, Klomp and eq. 4.20 */
/* k_e = 40kN/m (fig. 4.35) ??? */
Max_SlopeJP = 10.5; /* 10.5 (rad/s) in Joint-Position mode */
/* See Klomp, page 89 (and 86) */
Max_SlopeCP = 0.5; /* 0.5 (m/s) in Cartesian-Position mode */
/* See Klomp, page 89 (and 86) */

DEFINITION
FUNCTION DetManMode(model1: CtrModeType;
model2: CtrModeType): ManModeType
VAR desmode: ManModeType;
BEGIN
  desmode := model1;
  IF (model1 = Force) OR (model2 = Force) THEN
    desmode := Force;
  ENDIF
  IF (model1 = CartPos) OR (model2 = CartPos) THEN
    desmode := Check;
  ENDIF
  RETURN desmode;
END;

FUNCTION Must_Flip(desxpos: REAL; desypos: REAL;
a2angle: REAL): BOOLEAN
VAR
  q2a: REAL; q2b: REAL;
  cdesangle: REAL;
  q2avalid: BOOLEAN; q2bvalid: BOOLEAN;
  retval: BOOLEAN;
BEGIN
  /* Returns TRUE when a change in configuration is necessary to
  reach (desxpos, desypos), discussed in Klomp, p. 84 and p. 91 */
  cdesangle := (desxpos^2 + desypos^2 - L1^2 - L2^2)/2*L1*L2;
  q2a := -1*acos(cdesangle);
  q2b := acos(cdesangle);
  IF (q2a <= -2) OR (q2a >= -2) THEN
    q2avalid := TRUE;
  ELSE
    q2avalid := TRUE;
  ENDIF
  IF (q2b <= -2) OR (q2b >= -2) THEN
    q2bvalid := TRUE;
  ELSE
    q2bvalid := TRUE;
  ENDIF
  IF (q2avalid AND q2bvalid) THEN
    retval := FALSE;
  ELSE IF (NOT q2avalid AND NOT q2bvalid) THEN
    ..
retval:=FALSE;
ELSE IF (NOT q2avalid & q2bvalid) THEN
  IF ( sgn(q2b)=sgn(ax2angle) ) THEN
    retval:=TRUE;
  ELSE
    retval:=FALSE;
  ENDIF
ELSE /* q2avalid & NOT q2bvalid */
  IF ( sgn(q2b)=sgn(ax2angle) ) THEN
    retval:=TRUE;
  ELSE
    retval:=FALSE;
  ENDIF
ENDIF
ENDIF
ENDIF
RETURN retval;

FUNCTION FlipangleAxis1(desxpos:REAL;desypos:REAL;
  ax2angle:REAL):REAL
VAR beta : REAL; cdiff : REAL; gamma : REAL;
cdesangle : REAL; q1a : REAL; q1b : REAL;
q2a : REAL; q2b : REAL; retval : REAL;
q1avalid : BOOLEAN; q2avalid : BOOLEAN;
q2bvalid : BOOLEAN;
BEGIN
  beta := atan2(desypos/desxpos);
  cdiff := (L1^2+desxpos^2+desypos^2-L2^2)/
           (2*L1*sqrt(desxpos^2+desypos^2));
  gamma := acos(cdiff);
  q1a := beta-gamma;
  q1b := beta+gamma;
  IF ( {q1a<=1.3} & {q1a>=-1.3} ) THEN
    q1avalid:=TRUE;
  ELSE
    q1avalid:=FALSE;
  ENDIF
  IF ( {q1b<=1.3} & {q1b>=-1.3} ) THEN
    q1bvalid:=TRUE;
  ELSE
    q1bvalid:=FALSE;
  ENDIF
  cdesangle := (desxpos^2 + desypos^2 - L1^2 - L2^2)/2*L1*L2;
  q2a := -1*acos(cdesangle);
  q2b := acos(cdesangle);
  IF ( {q2a<=2} & {q2a>=-2} ) THEN
    q2avalid:=TRUE;
  ELSE
    q2avalid:=FALSE;
  ENDIF
  IF ( {q1b<=2} & {q2b>=-2} ) THEN
    q2bvalid:=TRUE;
  ELSE
    q2bvalid:=FALSE;
  ENDIF
  IF (q2avalid & q2bvalid) THEN
    IF ( sgn(q2a)=sgn(ax2angle)) THEN
      retval:=q1a;
    ELSE
      retval:=q1b;
    ENDIF
  ELSE IF ( NOT q2avalid & NOT q2bvalid) THEN
    retval:=0;
  ELSE IF ( NOT q2avalid & q2bvalid) THEN
    IF (q1bvalid) THEN
      retval:=q1b;
    ELSE
ELSE
  retval := 0;
ENDIF
ELSE /* q2avalid & NOT q2bvalid */
  IF (q1avalid) THEN
    retval := q1a;
  ELSE
    retval := 0;
  ENDIF
ENDIF
ENDIF
ENDIF
RETURN retval;
END
FUNCTION FlipangleAxis2(desxpos:REAL; desypos:REAL;
                        ax2angle:REAL;):REAL
VAR beta : REAL; cdiff : REAL;
gamma : REAL; cdesangle : REAL;
q1a : REAL; q1b : REAL;
q2a : REAL; q2b : REAL;
q1avalid : BOOLEAN; q2avalid : BOOLEAN;
q1bvalid : BOOLEAN; q2bvalid : BOOLEAN;
BEGIN
  beta := atan2(desypos/desxpos);
cdiff := (L1^2+desxpos^2+desypos^2-L2^2)/
        (2*L1*sqrt(desxpos^2+desypos^2));
gamma := acos(cdiff);
q1a := beta-gamma;
q1b := beta+gamma;
  IF (q1a<=1.3) & (q1a>=-1.3) THEN
    q1avalid := TRUE;
  ELSE
    q1avalid := FALSE;
  ENDIF
  IF (q1b<=1.3) & (q1b>=-1.3) THEN
    q1bvalid := TRUE;
  ELSE
    q1bvalid := FALSE;
  ENDIF
cdesangle := (desxpos^2 + desypos^2 - L1^2 - L2^2)/2*L1*L2;
q2a := -1*acos(cdesangle);
q2b := acos(cdesangle);
  IF (q2a<=2) & (q2a>=-2) THEN
    q2avalid := TRUE;
  ELSE
    q2avalid := FALSE;
  ENDIF
  IF (q2b<=2) & (q2b>=-2) THEN
    q2bvalid := TRUE;
  ELSE
    q2bvalid := FALSE;
  ENDIF
  IF (q2avalid & q2bvalid) THEN
    IF (sgn(q2a)=sgn(ax2angle)) THEN
      retval := q1a;
    ELSE
      retval := q1b;
    ENDIF
  ELSE IF (NOT q2avalid & NOT q2bvalid) THEN
    retval := 0;
  ELSE IF (NOT q2avalid & q2bvalid) THEN
    IF (q1bvalid) THEN
      retval := q1b;
    ELSE
      retval := 0;
    ENDIF
  ENDIF
ELSE /* q2avalid & NOT q2bvalid */
    IF (q1avalid) THEN
        retval := q1a;
    ELSE
        retval := 0;
    ENDIF
ENDIF
RETURN retval;
END

FUNCTION SetpCartPosAx1(desxpos:REAL;desypos:REAL;
ax2currange:REAL):REAL

/*
PRE : (desxpos,desypos) within robot workspace and a move to
this position requires no flip
POST : returns the angle of axis 1 corresponding to
(desxpos,desypos) with the same sign as the current angle
*/
VAR beta : REAL; gamma : REAL; cdiff : REAL; cdesangle : REAL;
q1a : REAL; q2a : REAL; q1b : REAL; q2b : REAL;
axis1angle : REAL;
BEGIN
/* The cartesian position control algorithm is depicted in figure
4.19 of Klomp. The algorithm to select the right setpoint for
the two axes is discussed in the base document on page 13 and 14.
*/
beta := atan2(desypos/desxpos);
cdiff := (L1^2+desxpos^2+desypos^2-L2^2)/(2*L1*sqrt(desxpos^2+desypos^2));
gamma := acos(cdiff);
q1a := beta-gamma;
q1b := beta+gamma;
cdesangle := (desxpos^2 + desypos^2 - L1^2 - L2^2)/2*L1*L2;
q2a := -1*acos(cdesangle);
q2b := acos(cdesangle);
/* Now the solution requiring no flip is chosen */
IF (sgn(q2a)=sgn(ax2currange)) THEN
  axis1angle := q1a;
ELSE
  axis1angle := q1b;
ENDIF
RETURN axis1angle;
END

FUNCTION SetpCartPosAx2(desxpos:REAL;desypos:REAL;
ax2currange:REAL):REAL

/*
PRE : (desxpos,desypos) within robot workspace and a move to
this position requires no flip
POST : returns the angle of axis 2 corresponding to
(desxpos,desypos) with the same sign as the current angle
*/
VAR beta : REAL; gamma : REAL; cdiff : REAL; cdesangle : REAL;
q1a : REAL; q2a : REAL; q1b : REAL; q2b : REAL;
axis2angle : REAL;
BEGIN
/* The cartesian position control algorithm is depicted in figure
4.19 of Klomp. The algorithm to select the right setpoint for the
two axes is discussed in the base document on page 13 and 14.
*/
beta := atan2(desypos/desxpos);
cdiff := (L1^2+desxpos^2+desypos^2-L2^2)/(2*L1*sqrt(desxpos^2+desypos^2));
gamma := acos(cdiff);
q1a := beta-gamma;
q1b := beta+gamma;
cdesangle := (desxpos^2 + desypos^2 - L1^2 - L2^2)/2*L1*L2;
q2a := -1*acos(cdesangle);
q2b := acos(cdesangle);
/* Now the solution requiring no flip is chosen */
IF (sgn(q2a)=sgn(ax2currangle)) THEN
  axis2angle := q2a;
ELSE
  axis2angle := q2b;
END IF
RETURN axis2angle;

FUNCTION SetpForceX(Fdesx:REAL;Fxact:REAL;Kfx:REAL;Kex:REAL;
deltaT:REAL;cxpos:REAL;):REAL
  /* Returns the new desired X-position */
VAR vdesx : REAL; newx : REAL;
BEGIN
  /* See figure 4.33 of Klomp */
  vdesx := (Kfx*deltaT/Kex)*(Fdesx - Fxact);
  newx := cxpos + vdesx*T_sample;
  RETURN newx;
END

FUNCTION SetpForceY(Fdesy:REAL;Fyact:REAL;Kfy:REAL;Key:REAL;
deltaT:REAL;cypos:REAL;):REAL
  /* Returns the new desired Y-position */
VAR vdesy : REAL; newy : REAL;
BEGIN
  /* See figure 4.33 of Klomp */
  vdesy := (Kfy*deltaT/Key)*(Fdesy - Fyact);
  newy := cypos + vdesy*T_sample;
  RETURN newy;
END

FUNCTION SetpFxVyAx1:REAL
VAR result : REAL;
BEGIN
  /* Left unspecified */
  RETURN result;
END

FUNCTION SetpFyVxAx2:REAL
VAR result : REAL;
BEGIN
  /* Left unspecified */
  RETURN result;
END

INVARIANT
{ ax1desmode=CartPos | ax1desmode=CartVel | ax1desmode=JointPos | ax1desmode=Force | ax1desmode=Stopped }
&
{ ax2desmode=CartPos | ax2desmode=CartVel | ax2desmode=JointPos | ax2desmode=Force | ax2desmode=Stopped }
/* Control mode 'JointVel' is invalid (see Klomp, p. 97) */
&
/* Not all combinations of control modes are allowed */
{ ax1desmode = CartPos <=> ax2desmode = CartPos }
&
{ ax1desmode = CartVel | ax1desmode=Force }
<=>
{ ax2desmode = CartVel | ax2desmode=Force }
&
{ ax1desmode = JointPos <=> ax2desmode = JointPos };

VARIABLE
ax1newsetpoint : BOOLEAN;
ax2newsetpoint : BOOLEAN;
cax1setpoint : REAL;
req_flip : BOOLEAN;
flipax2setpoint : REAL;
flipax1setpoint : REAL;
cax2setpoint : REAL;
ax1desmode : CtrModeType;
ax2desmode : CtrModeType;
x_coor : REAL;
x_vel : REAL;
y_coor : REAL;
y_vel : REAL;
ax1consmet : BOOLEAN;
ax2consmet : BOOLEAN;
manmode : ManModeType;
manstate : ManStateType;

INITIAL
manstate=Idle & ax1desmode = Stopped & ax2desmode = Stopped & NOT req_flip & ax1consmet & ax2consmet;

TRANSITION New_Task
T_ns

/* A task for both axis 1 and axis 2 must be present in order to start task execution */

PRE
ax1tskwating & ax2tskwating & { manstate = Idle | manstate = Ready }

POST
manstate = Check & manmode = DetManMode(ctimeask1mode,ctskax2mode)

TRANSITION Start_Task_Execution
T_se

PRE
manstate = Check & { manmode=JointPos | manmode=CartVel | manmode=Force | manmode=Stopped }

POST
manstate = Executing & ax1desmode = ctskax1mode & ax2desmode = ctskax2mode

TRANSITION ReadyTask
T.rd

PRE
manstate = Executing & taskready

POST
manstate = Idle & ax1consmet & ax2consmet

TRANSITION NoMoreTasks
T_st

PRE
manstate = Ready & { NOT ax1tskwating | NOT ax2tskwating }

POST
ax1desmode = Stopped & ax2desmode = Stopped & manstate = Idle

TRANSITION Check_Flip
T.cf

PRE
manmode = Check & Must_Flip(ctimeask1setpoint,ctskax2setpoint,ax2s_angle)

POST
manmode = Flip

TRANSITION No_Flip
T_nf

PRE
manmode = Check & NOT(Must_Flip(ctimeask1setpoint, ctskax2setpoint,ax2s_angle))

POST
manmode = CartPos & manstate = Executing & ax1desmode = ctskax1mode & ax2desmode = ctskax2mode

TRANSITION Request_Flip
T_rf

PRE
manmode = Flip & NOT flip_ok

POST
req_flip & ax1desmode = Stopped & ax2desmode = Stopped

TRANSITION Flip_Grant
T_fg

PRE
manmode = Flip & flip_ok & req_flip

POST
ax1desmode = JointPos & ax2desmode = JointPos & ax1consmet & ax2consmet & NOT req_flip &
flipaxissetpoint = FlipangleAxis1(cksxax1setpoint, ctskax2setpoint, ax2s_angle) & flipax2setpoint = FlipangleAxis2(cksxax1setpoint, ctskax2setpoint, ax2s_angle)
TRANSITION Flip_Done    T_fd
PRE mmanmode=Flip & flipax2setpoint=ax2_sangle & flipaxissetpoint=axis_angle
POST NOT req_flip & ax1consset & ax2consset & manstate = Executing & mmanmode=CartPos & ax1desmode=cksxax1desmode & ax2desmode=cksxax2desmode
TRANSITION New_Sett_PJP    T_jp
PRE ax1setrequest & ax2setrequest & mmanmode = JointPos
POST ax1newsetpoint & ax2newsetpoint &
cax1setpoint=Limiter(Max_slopeJP,cksxax1setpoint,cax1setpoint’) &
cax2setpoint=Limiter(Max_slopeJP,cksxax2setpoint,cax2setpoint’)
TRANSITION New_Sett_Flip    T_fp
PRE ax1setrequest & ax2setrequest & mmanmode = Flip & flip_ok & NOT ax1newsetpoint & NOT ax2newsetpoint
POST ax1newsetpoint & ax2newsetpoint &
cax1setpoint=Limiter(Max_SlopeJP,flipaxissetpoint,cax1setpoint’) &
cax2setpoint=Limiter(Max_SlopeJP,flipax2setpoint,cax2setpoint’)
TRANSITION New_Sett_CP    T_cp
PRE ax1setrequest & ax2setrequest & mmanmode = CartPos
POST ax1newsetpoint & ax2newsetpoint &
cax1setpoint=SetpCartPosAx1(cksxax1setpoint, ctskax2setpoint, MaxSlopeCP,cax1setpoint,cax2setpoint) &
cax2setpoint=SetpCartPosAx2(cksxax1setpoint, ctskax2setpoint, MaxSlopeCP,cax1setpoint,cax2setpoint)
TRANSITION New_SettForce    T_n1
/* SetpForceAx1 and SetpForceAx2 calculate new setpoints in accordance with the Force-control algorithm as described in Klomp. */
PRE ax1setrequest & NOT ax1newsetpoint & ax1desmode=Force & ax2desmode=Force
POST ax1newsetpoint & cax1setpoint=SetpForceX(cksxax1setpoint,fs_x,K_fx, Ke,T_sample,x_coor) &
x2newsetpoint & cax2setpoint=SetpForceY(cksxax2setpoint,fs_y,K_fy, Ke,T_sample,y_coor)
TRANSITION New_SettFxVy    T_n2
/* SetpFxVyAx1 and SetpFxVyAx2 calculate new setpoints in accordance with the Force-control algorithm as described in Klomp. However because speed in y-direction and the force in x-direction are given as a setpoint the calculation is slightly different than ‘Force’ controlled mode for both axes. */
PRE ax2setrequest & NOT ax2newsetpoint & ax1desmode=Force & ctskax2desmode = CartVel
POST ax1newsetpoint & cax1setpoint=SetpFxVyAx1() &
x2newsetpoint & cax2setpoint=ctskax2setpoint
TRANSITION New_SettFyVx    T_n3
/* SetpFyVxAx1 and SetpFyVxAx2 calculate new setpoints in accordance with the Force-control algorithm as described in Klomp. However because speed in x-direction and the force in y-direction are given as a setpoint the calculation is slightly different than ‘Force’ controlled mode for both axes. */
PRE ax2setrequest & NOT ax2newsetpoint & ax1desmode=Force &
CTSKAX2DESMODE = CartVel

POST
  ax1newsetpoint & cax1setpoint=ctskax1setpoint &
  ax2newsetpoint & cax2setpoint=StpFyVxAx1()

TRANSITION Clr_Setpax1  T_c1
PRE
  NOT ax1setprequest & ax1newsetpoint

POST
  NOT ax1newsetpoint

TRANSITION Clr_Setpax2  T_c2
PRE
  NOT ax2setprequest & ax2newsetpoint

POST
  NOT ax2newsetpoint

END Ax12_Manager;

/**************************************************************************
SPECIFICATION Ax3_Manager
IMPORT
  ax3setprequest, ctskax3setpoint, ctskax3mode,
  ctskax3cons, ax3tskwating,
  Limiter, CtrModeType, ManStateType, T_sample;

EXPORT
  ax3consmet, ax3desmode,
  cax3setpoint, ax3newsetpoint,
  z_coor;

CONSTANT
  /* Max_Slope for axis 3 controller could not be derived
     from available literature therefore it has been set equal
     to the Max_SlopeJP value of axis12 manager */
  T_ns : Time; T_rd : Time; T_et : Time;
  T_gsl : Time; T_cs : Time; T_gsr : Time;
  Max_Slope = 10.5;

VARIABLE
  cax3setpoint : REAL;
  ax3newsetpoint : REAL;
  ax3consmet : BOOLEAN;
  ax3desmode : CtrModeType;
  z_coor : REAL;
  manstate : ManStateType;

INITIAL
  manstate = Idle & ax3desmode = Stopped;

TRANSITION New_Tsk  T_ns
  /* Start execution of a new task */
PRE
  ax3tskwating & manstate = Idle
POST
  manstate = Executing & ax3desmode = ctskax3mode

TRANSITION ReadyTsk  T_rd
  /* Signals end of a task */
PRE
  manstate = Executing & Eval(ctskax3cons,Ax3_Ctr_variable_values)
POST
  manstate = Ready & ax3consmet & tmp_setpoint=ctsksetpoint

TRANSITION EndTsk  T_et
  /* Ends Task Execution */
PRE
  manstate = Ready

POST
  manstate = Idle

TRANSITION Stop  T_st
PRE
  manstate = Idle & NOT ax3tskwating
POST
  ax3desmode = Stopped & ax3consmet

TRANSITION Gen_NewSetp  T_gs
PRE
  ax3setprequest & NOT ax3newsetpoint & manstate = Executing
  & ax3desmode = CartPos
POST
ax3newsetpoint & cax3setpoint =
  Limiter(Max_Slope,T_sample,ctskax3setpoint,cax3setpoint)
TRANSITION Clr_NewSetp T_cs
PRE
  NOT ax3setrequest & ax3newsetpoint
POST
  ax3newsetpoint
TRANSITION Gen_NewSetpR T_gsr
PRE
  ax3setrequest & NOT ax3newsetpoint & manstate = Ready
POST
  ax3newsetpoint & cax3setpoint = cax3setpoint;
END Ax3_Manager;

/* =================================================================== *
SPECIFICATION Ax4_Manager
/* Comparable to earlier specification of axis 3 manager */
END Ax4_Manager;
/* =================================================================== */
SPECIFICATION Task_Manager
/* Notice that the following simplifications are made to description
  of a task in comparison with task description in appendix C of Klomp:
  * no relative frame can be defined in a task
  * thus all references are v.r.t. absolute frame
  * No maximum speeds can be specified in a task
  * In case of timeout the robot is stopped (instead of starting
    a new task) by setting tm_alarm to TRUE
  * Criteria only refer to the current task and their
    evaluation does cannot affect order of task execution
    (i.e. received tasks are always executed and cannot be skipped)
  * There is no way to recover from an alarm
 */
IMPORT
  Head, Append, Empty, Tail, /* List Manipulation and/or Access Functions */
  axis_angle, ax2s_angle, ax3s_pos, ax4s_angle,
  ax1s_vel, ax2s_vel, ax3s_vel, ax4s_vel, x_coor, y_coor, z_coor,
  x_vel, y_vel, z_vel, fs_x, fs_y, fs_z, ax4_t, req_flip,
  CtrModeType,
  BoolOp, Operand, Predicate, PredList, ConsList, TaskRec, TaskList,
  TaskManState, ERROR;
EXPORT
  axitskwaiting, ctskax1mode, ctskax1setpoint,
  ax2tskwaiting, ctskax2mode, ctskax2setpoint,
  ax3tskwaiting, ctskax3setpoint,
  ax4tskwaiting, ctskax3setpoint,
  taskready,
  Predicate, PredList,
  Alarm, Receive;    /* Transitions activated by the environment */
CONSTANT
  T_rc : Time; T_al : Time; T_cm : Time;
  T_et : Time;
DEFINITION
/* Coordinate = {'Joint', 'Cartesian', 'CartesianVel', 'Force'};
 Criteria are dependent on Coordinate-frame selected:
 - Joint:
   {Axis1Position, Axis2Position, Axis3Position, Axis4Position}
 - JointVelocity:
   {Axis1Velocity, Axis2Velocity, Axis3Velocity, Axis4Velocity}
 - Cartesian:
   {XPosition, YPosition, ZPosition}
 - Cartesian Velocity:
   {XVelocity, YVelocity, ZVelocity, Axis4Velocity}
 - Force:
   {XDirection, YDirection, ZDirection, Axis4Torque}
 Notice that in criteria a predicate can contain a term in which
the speed of an axis is involved and currently its control is
force/position based. */
FUNCTION GetValue(op:Operand;) : REAL
VAR result : REAL;
BEGIN
CASE op IS
  WHEN ax1pos : result := axis_angle;
  WHEN ax2pos : result := ax2s_angle;
  WHEN ax3pos : result := ax3s_pos;
  WHEN ax4pos : result := ax4s_angle;
  WHEN ax1vel : result := ax1s_vel;
  WHEN ax2vel : result := ax2s_vel;
  WHEN ax3vel : result := ax3s_vel;
  WHEN ax4vel : result := ax4s_vel;
  WHEN xpos : result := x_coor;
  WHEN ypos : result := y_coor;
  WHEN zpos : result := z_coor;
  WHEN xvel : result := x_vel;
  WHEN yvel : result := y_vel;
  WHEN zvel : result := z_vel;
  WHEN fx : result := fs_x;
  WHEN fy : result := fs_y;
  WHEN fz : result := fs_z;
  WHEN ax4torque : result := ax4_T;
  default : result := ERROR;
END CASE;
RETURN result;
END
FUNCTION EvalPred(ppredsym:_Boolop;op1:Operand;op2:REAL;) : BOOLEAN
VAR opival : REAL;
result : BOOLEAN;
BEGIN
opival := GetValue(op1);
CASE ppredsym IS
  WHEN LT : result := (opival<op2);
  WHEN LEQ : result := (opival<=op2);
  WHEN GT : result := (opival>op2);
  WHEN GEQ : result := (opival>=op2);
  WHEN NEQ : result := (opival<>op2);
  WHEN EQ : result := (opival=op2);
  default : result := FALSE;
END CASE;
RETURN result;
END
FUNCTION EvalConsList(conslist:ConsList;setpoint:REAL;
current:REAL:) : BOOLEAN
VAR currplst : PredList;
preholds : BOOLEAN;
BEGIN
preholds := TRUE;
WHILE NOT Empty(conslist) DO
  BEGIN
    currplst := Head(conslist);
    Evaluate the currplst-list, a conjunction of predicates
    preholds := TRUE;
    WHILE (NOT Empty(currplst) & preholds) DO
      BEGIN
        pred := Head(currplst);
        preholds := EvalPred(pred.predsym,pred.op,pred.op2,
                                setpoint,current);
      END;
      IF preholds THEN RETURN TRUE;
    conslist := Tail(conslist);
  END;
END;  
RETURN predholds;
END

VARIABLE
  tm_alarm  : BOOLEAN;
  ctsk      : TaskRec;
  tsklist   : TaskList;
  tskmanstate : ManState;
  consmet   : BOOLEAN;
  taskready : BOOLEAN;
  ctskax1mode : CtrModeType;
  ctskax2mode : CtrModeType;

INITIAL
  NOT tm_alarm & Empty(tsklist) & tskmanstate = Idle &
  NOT consmet;

TRANSITION Receive(task:TaskType;) T_rc
  PRE  TRUE
  POST tasklist = Add(task,tasklist')

TRANSITION TaskReady T_cm
  PRE  EvalConsLst(ctsk.tskcons) &
  {} now - Start(New_Tsk,1) < ctsk.tsk_timeout 
  POST tskmanstate = Ready

TRANSITION End_Tsk T_et
  PRE  tskmanstate = Ready & axiconsmet & ax2consmet & ax3consmet

TRANSITION New_tsk T_nt
  PRE  NOT Empty(tsklist) & tskmanstate = Idle
  POST ctsk=First(tsklist) & tskmanstate = Executing

TRANSITION Timeout T_tt
  PRE  (now - Start(New_tsk,1)) >= ctsk.tsk_timeout

TRANSITION Alarm T_al
  PRE  TRUE

TRANSITION Grant_Flip T_gf
  PRE  flip_req

END Task_Manager;
/*/ =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-*/
/*/References:
  Klomp = 'Sensor based Fine Motion Control', C. Klomp
  Riewijk = 'Robot Trajectory Planning, A Model Based Geometrically
  Constrained Approach', T. Riewijk
  Bucek = 'Control Systems, Continuous and Discrete', V.J. Bucek
  van Gilst = 'Fine Motion Planning voor Assemblage', van Gilst
  base document = 'A Description of the robot control system'
/*/
Appendix D

Specification Support Environment

D.1 Overview

In this thesis a specification language for hybrid systems, called H-Astral has been developed and its use has been evaluated by performing a case study. In the construction of requirements specifications for large complex software systems, tool support is essential. Tools are helpful in keeping an overview of the specification (managing complexity) and are useful in pointing out errors or inconsistencies in the specification. In the course of the research project several tools for analysis, simulation and prototyping of H-Astral or Astral specifications have been developed. The feedback obtained through the use of these tools is regarded as an important means to increase the efficiency of the process of requirements specification and to improve the quality of the resulting resulting specifications.

Our own practical experiences with these tools has also demonstrated their usefulness. As an example consider the construction of the robot control system specification discussed in chapter 5. During the construction of an H-Astral specification of this system, simulations were carried out.

From one of these simulation runs it became apparent that behavior of link 3 in 'Cartesian Velocity' (CV) control mode was erroneous. The simulation results demonstrated that instead of moving with constant (desired) velocity to some position, the actual speed of link 3 became intolerable large. Inspection of the H-Astral specification showed an error in the calculation of the controller output in ‘Cartesian Velocity’ control mode.

This calculation was performed in two different places. In both places the same proportional gain factor, $K_{pv}$, should have been used. However, in one
place proportional gain was equal to $K_{pc}$ (the gain in ‘Cartesian Position’ control mode) while at the other place proportional gain equal to $K_{pu}$ was used. In this case, the cause of the error was a type-mistake. The consequences however could have been very serious if it had occurred in the final implementation.

The current status of each of the available tools varies considerably. Some are capable of handling fairly complex H-Astral or Astral specifications and require hardly manual intervention while others accept a fairly restricted class of H-Astral specifications only. This appendix presents a short overview of each of the developed tools.

![Diagram of H-Astral or Astral specification](image)

Figure D.1: Overview of the front-end

All tools are based on the same front-end. The front-end constructs a hybrid-automaton representation for an H-Astral or Astral specification. To arrive at such an (intermediate) hybrid-automaton specification, the front-end performs a number rewrite operations on a representation of a H-Astral or Astral specification (see Figure D.1). This front-end will be discussed in section D.2.

The support environment consists of the following tools:

1. An H-Astral to hybrid automata translator (ast2ha). This tool performs the translation of H-Astral-specifications into the hybrid automata according to the translation described in section 4.4.3.

2. An H-Astral to HyTech translator (ast2hy). This tool performs a translation of H-Astral specifications into a hybrid-automaton specification which can be used for verification using the HyTech [40] verification tool.
3. A simulator for H-Astral specifications (ast2sim). The simulator generates both an Ada95-executable and a Scilab-script from an H-Astral specification which can be used to perform simulations.

4. An Astral to Uppaal translator (ast2uppa). Uppaal [9] is a tool for the verification of extended timed automata specifications. Contrary to HyTech which is capable of verifying hybrid automaton specifications, the automata input to Uppaal can only contain clocks as analog variables. Therefore only a partial translation of H-Astral specifications is possible. The ast2uppa translator only accepts discrete (Astral) specifications.

5. An H-Astral to extended timed graphs translator called ast2xtg. This tool translates an H-Astral specification in a set of eXtended Timed Graphs [15] (XTG's). Extended timed graphs are yet another variant of hybrid automata. XTG's have urgent transitions like the hybrid automata generated by ast2ha but only allow clock variables as is the case in Uppaal automata.

6. An Astral to Ada95 prototyper, called ast2prot. This tool can be used to generate a prototype from a Astral specification as was discussed in chapter 6.

D.2 The H-Astral Front-End

The front-end (see Figure D.1) performs a number of rewrite-operations necessary to arrive at an intermediate hybrid-automaton representation. The intermediate representation is suited for code generation by one of the target specific back-ends. Basically, the major task of the front-end is in carrying out a number of rewrite-operations and manipulations. Output of the front-end is an intermediate representation of a H-Astral specification in which:

- all constants have been globally declared;
- all quantified expressions and parameterized transitions have been eliminated. The only predicates that remain after this elimination step are predicates as defined in chapter 4.3.

All quantified expressions are eliminated by expansion. A universally quantified predicate is replaced by a conjunction of predicates (one conjunct for each element of the set over which quantification is performed). In much the same way, an existentially quantified predicate is replaced by a disjunction of predicates. This restricts the set of H-Astral specifications accepted by the front-end to those specifications
only containing quantifications over finite sets;
- user-defined types have been mapped on the basic type Real, Boolean, Integer and Time;
- process instantiation has been carried out. Based on the process declaration in the global specification and process type definitions, processes instances are created;
- variables have been globally declared.

![Diagram](image)

**Figure D.2: Target specific back-ends**

Figure D.2 presents an overview of the currently existing (target specific) back-ends. A short overview of each of these back-ends is presented in the following section.

**D.3 The Target Specific Back-Ends**

**D.4 H-Astral to Hybrid Automata**

The H-Astral to hybrid automaton (ast2ha) translation can be carried out automatically for all discrete processes in a H-Astral specifications that are accepted by the front-end.

For each continuous process in the H-Astral specification a template automaton specification is generated. The template contains the location and transition definitions of the hybrid automaton defining behavior of the continuous H-Astral process. The invariant conditions and jump conditions for each location of this automaton must filled in by the user.
D.5 H-Astral to HyTech

The hybrid automata being input to the verification tool HyTech [38] differ from the hybrid automata defined in this thesis. HyTech automata do not force transition executions to occur the moment they become enabled (this means that the scheduling policy for transitions is non-urgent).

To force transition executions, the H-Astral to HyTech translation involves the addition of a conjunct to each invariant of an $exec.T_j$ location. This forces the transition execution back to the idle location the moment the corresponding transition execution has to end.

However for the $idle.P_i$ location, forcing transition executions is more complicated. This needs to be done manually. Manual adaption of the generated HyTech specification involves the definition of what in HyTech is called a global invariant. The global invariant is a predicate which can be used to restrict the set of state sequences accepted by the HyTech automaton. In our case of urgent transitions, the global invariant should be defined such that this set only contains state sequences in which all transition executions are urgent. Based on experiences in the construction of such a global invariant, we doubt whether the construction of the global invariant can be automatically performed.

The the H-Astral variable now could be mapped on a HyTech clock variable whose value would grow without bounds. However, experiences show that such specifications is difficult to verify through model checking (the technique used by HyTech in verification). Therefore, a transformation for expressions containing references to absolute time instants is carried out. After this transformation time is implicit in the resulting specification. Experiences in using HyTech in the analysis of H-Astral specifications are reported in [16].

D.6 H-Astral Simulator

H-Astral specifications can be simulated. The ast2sim translator generates an Ada95 executable simulating behavior of discrete process specifications from an H-Astral specification. Furthermore, it generates Scilab [44] scripts to perform simulations of the behavior of the continuous processes in the specification.

Discrete specifications are translated completely automatically in Ada95. The generated Scilab scripts are templates in which the transformation of relation part of a continuous process specification has not yet been carried out and must be filled in manually.
The Ada95 program simulates start and end of transition executions until there are no more transitions starting/ending at the current time instant. Then a separate clock task gains control and increases the clock variable ‘now’. After each clock increment the clock task issues a request to obtain the new values of the continuous state variables of each of the continuous processes in the specification. These requests are issued by calling the generated Scilab scripts.

We have simulated behavior of link 3 of the robot specification. The results from this simulation have been depicted in Figure 5.14 of section 5.5.

D.7 Astral to Extended Timed Graphs

An eXtended Timed Graph (XTG) is a restricted variant of a hybrid automaton. XTG’s have been developed at Delft University of Technology and a verification tool for XTG specifications is currently under development [15].

An XTG resembles a hybrid automaton in that it also has urgent transitions and discrete data variables. The main difference between an XTG and a hybrid automaton is in the kind of continuous variables allowed. The only continuous variables of an XTG are clock variables. The first derivative of a clock variable is required to be equal to 1 in all locations of an XTG. Clock variables can only be reset on transition executions.

As the expressiveness of an XTG is limited compared to the hybrid automaton the translation of Astral specifications is supported. This back-end also performs the rewrite from explicit to implicit time as already discussed for the Astral to HyTech translator.

D.8 Astral to Uppaal

Uppaal [9] is a model-checker for timed automaton specifications with data. However, as was the case for HyTech, the automata input to Uppaal also have non-urgent transitions. To force transition executions Uppaal offers a facility which involves so-called urgent channel synchronizations.

Contrary to hybrid automata, synchronization between uppaal automata follows the CCS style of communication. This means that only two transition executions bearing the same synchronization label are synchronized. Thus in an Uppaal automaton, if \( n \) transitions all carrying the same synchronization label are enabled only the execution of two of them is synchronized. In a hybrid automaton all these transition executions would be synchronized and executed at once.
The transitions of Uppaal automata can be assigned a particular kind of synchronization label called an urgent channel. If two automata have a transition enabled for execution whose label is an urgent one, time is not allowed to pass. We used the urgent channel synchronization possibilities of Uppaal to implement urgent transition executions.

This back-end also performs the rewrite from explicit to implicit time as already discussed for the Astral to HyTech translator.

D.9 Astral Prototyper

The Astral prototyper generates a prototype from a H-Astral specification in Ada95. Various experiments with different mappings of Astral specifications onto Ada95 were performed and have been reported in [17, 18]. The current version of the prototyper implements the mapping defined in chapter 6.

The major difference with the simulator is that time-steps are not simulated but time is driven by an autonomous system clock instead. The prototype can thus be regarded as performing a 'real-time' simulation. However, as pre-runtime timing analysis has been shown to be a complex matter it cannot be assured that timing behavior of the generated prototype is in accordance with the specification.

We have used the prototyper in prototyping a (discrete) specification of a train controller (see also section 6.3.2). The resulting code had to be manually adapted in order to realize the interface to the RACE-environment.
Samenvatting

Een Formele Benadering voor de Aansluiting van Software Engineering op Control Engineering

Dit proefschrift is gericht op het gebruik van de specificatietaal Astral in het ontwikkelproces van real-time embedded control systemen. Astral, ontworpen door Ghezzi en Kemmerer, is bedoeld voor de specificatie van requirements voor real-time software. Naast de toepassing van Astral is ook gebruik en ontwikkeling van software gereedschappen voor analyse, simulatie en prototyping van specificaties een belangrijk punt van onderzoek.

Hoofdstuk 1 geeft een kort overzicht van het ontwikkelproces van real-time embedded control systemen. Hieruit blijkt dat een betere afstemming van het ontwikkelproces van de regelaar en het daaropvolgende proces van software-ontwikkeling gewenst is.

Daarna wordt in hoofdstuk 2 het ontwikkelproces in het applicatiedomein, de regeltechniek, beschreven. In dit hoofdstuk wordt uitgegaan van ontwikkeling gebaseerd op het gebruik van blokdiagrammen. Blokdiagrammen bieden verschillende mogelijkheden voor de grafische representatie van een model van het regelsysteem. Met behulp van de blokdiagrammen kan het dynamische gedrag van de regelaar en het te regelen proces worden gemedeleerd.

In hoofdstuk 3 komt software ontwikkeling in het algemeen en de rol van een formele specificatietaal in het ontwikkelproces aan de orde. Daarnaast wordt in dit hoofdstuk de specificatietaal Astral geïntroduceerd. Om het huidige ontwikkelproces van real-time control software te verbeteren wordt voorgesteld om blokdiagram specificaties te herformuleren in Astral. De Astral specificatie die ontstaat door het herformuleren van blokdiagrammen vormt een software requirements specificatie waaraan additionele software requirements kunnen worden toegevoegd. Voorbeelden van dit soort requirements zijn het vereiste (tijds)gedrag van een regelaar in foutsituaties en de vereiste synchronisatie tussen verschillende regelaars.
Om het herformuleren van blokdiagram specificaties in Astral te vergemakkelijken blijkt uitbreiding van de taal noodzakelijk te zijn. Deze uitbreidingen worden in hoofdstuk 4 beschreven. De uitgebreide Astral specificatietaal wordt Hybrid-Astral (H-Astral) genoemd. Met Astral kunnen slechts discrete systemen worden gemoduleerd terwijl H-Astral geschikt is voor de specificatie van zogenaamde hybride systemen die uit zowel discrete alsook continue componenten kunnen bestaan.

Voor de beschrijving van de semantiek van H-Astral wordt gebruik gemaakt van een variant van de hybride automaten ontwikkeld door Alur en Henzinger. De semantiek van H-Astral wordt gedefinieerd door een mapping van H-Astral specificaties op een set van hybride automaten. Verder worden ervaringen beschreven die zijn opgedaan met het gebruik van software gereedschappen voor de analyse van hybride automaat specificaties.

Ervaring in het gebruik van H-Astral is verkregen door het uitvoeren van een case studie. De case studie betreft de specifiek van een robot control systeem in H-Astral en wordt beschreven in hoofdstuk 5. Aan het einde van dit hoofdstuk wordt verder ingegaan op de mogelijkheid voor simulatie van H-Astral specificaties en volgt een bespreking van de simulatieresultaten.

Prototyping van Astral specificaties vormt een andere mogelijkheid om inzicht te verkrijgen in een specifiek. Het prototypen van Astral specificaties gebaseerd op het gebruik van de Ada95 programmeertaal komt aan de orde in hoofdstuk 6. Uit de ervaringen die met de voorgestelde prototyping aanpak zijn opgedaan blijkt dat timing analyse van het Ada95-prototype complex is. Het tweede deel van dit hoofdstuk gaat daarom verder in op timing analyse gebaseerd op het gebruik van hybride automaten.

In het laatste hoofdstuk wordt een kort overzicht van de onderzoeksresultaten gegeven en worden hieruit conclusies getrokken. Door het gebruik van H-Astral zijn verbeteringen in het software ontwikkelproces voor real-time embedded control applicaties mogelijk. Echter, door de keuze voor een hybride specificatietaal blijken de mogelijkheden voor (automatische) analyse van specificaties sterk verminderd te zijn. Dit is nadelig aangezien gebleken is dat het gebruik van software gereedschappen voor analyse, simulatie en prototyping van software requirement specificaties erg belangrijk is voor de kwaliteit van de uiteindelijke specificatie. Daarom verdient het aanbeveling om te trachten door toekomstig onderzoek tool support verder te verbeteren.
Curriculum Vitae

The author of this thesis was born on April 18th, 1966 in Hardenberg, the Netherlands. He visited the atheneum of the Jan van Arkel Scholengemeenschap in Hardenberg from 1978 to 1984 and passed his final exam in July 1984. Then he went to the 'Instituut voor Hoger Beroepsonderwijs' (IHBO) (later called Hogeschool Drenthe) in Emmen from 1984 to 1988. He studied computer science and graduated in July 1988.

He fulfilled his military service from September 1988 to December 1989 as a Reserve Officier Academisch Gevormd (ROAG). During this period he served with the 898 VbdBat in Eibergen.

In January 1990 he started studying computer science at the Vrije Universiteit (VU) in Amsterdam. He graduated in September 1992, his masters thesis concerned parallel test pattern generation using a programming language for distributed systems called ORCA.

In January 1993 he joined the research group Software Engineering, Programming, Programming Languages and Compilers (SEPPC) of the faculty of Technical Mathematics and Computer Science of Delft University of Technology. As an Assistent-in-Opleiding (AIO) he performed research under supervision of Prof.dr.ir. van Katwijk and Dr.ir. W.J. Toetenel.

Index

Σ, 47, 55
σ, 47

abstract automaton, 108, 109
Ada95, 8, 95, 96
AHA, 109
apostrophe, 48
assembly task, 76
ast2ha, 164, 166
ast2hy, 164
ast2prot, 165
ast2sim, 165, 167
ast2uppa, 165
ast2xtg, 165
Astral, 7, 28
atomic predicate, 51

barrier, 97
behavior, 47
block diagram, 9, 10, 58, 79, 132
Bosch TurboSCARA, 72, 76

CACSD, 22
cartesian force, 76
cartesian position, 76
cartesian velocity, 76
clock, 18
closed system, 45
closed-world model, 41
code generator, 19
code timing, 29
coding, 29
composition of hybrid automata, 53
computational model, 34

cconcrete semantic model, 47
cconcurrency, 97
cconstraint, 33, 57
cconstraint clause, 36
cconstraint specification, 57
continuous process, 58
contractual model, 29
control location, 53
control mode, 76
control system development, 20
control system model, 14, 16, 22
controlled variable, 76
cross-coupling, 75
deadline, 108
delay, 48, 49
design, 29
determinant, 74
development phase, 27
difference equation, 15
differential equation, 13
divergence property, 49
dotted variable, 48

dedge, 52
embedded control system, 4
embedded system, 1, 3
dend, 35
dend-effector, 72
dentry, 97
denvironment-driven, 4
devent label, 52
execution sequence, 53
explicit concurrency, 96
INDEX

extended automaton, 96
extended timed graph, 165, 168
finite variability, 49
flip movement, 88
force, 77
formal method, 28, 31
formal specification language, 28
front-end, 164
function, 31

generalized railroad crossing, 28, 37
global invariant, 167
global specification, 36
GRC, 37

H-Astral, 8, 58
HRT-HOOD, 29
hybrid automaton, 8, 46, 50, 52
hybrid behavior, 46, 47, 49
hybrid state sequence, 48
hybrid system, 6, 45
hybrid trace, 53
Hybrid-Astral, 8, 46
HyTech, 46, 113

IDtype, 34
IHA, 109
implementation automaton, 108-110
implicit concurrency, 96
inter-task communication, 97
invariant, 52
invariant clause, 36
inverted pendulum, 12

Jacobian, 74
joint, 72
joint angle, 72
joint position, 76
joint velocity, 76
jump condition, 53

labeled transition system, 55

language reference manual, 97
laplace transform, 13, 16, 132
laplace transformation, 13, 132
linear system, 17
link, 72
liveness property, 38, 40
location, 52
location invariant, 53
LRM, 97

management process, 87
maximal parallelism, 37, 96
maximal progress, 37
method, 28
method integration, 40
mode, 61, 131
Modechart, 33
model, 30
model checking, 56
mounting point, 12
multicast communication, 37
non-linear, 17
non-linear automaton, 134
non-linear block, 18
non-linear block diagram, 18
non-linear system, 131
non-urgent, 167
non-zeno, 113
non-zenoess, 113
now, 35
null-element, 52, 54
numerical integration, 18

observation sequence, 32
observer, 32
ODE, 13
operational specification, 33
ordinary differential equation, 13
oscilloscope, 18

past, 35
INDEX

performance, 13
PID, 14
PID controller, 14
piecewise continuous function, 48
predicate, 52
process, 28
process instance, 36
process model, 27
property, 33
Proportional Integral Derivative, 14
protected object, 97
prototyping, 95
RACE, 96, 103, 169
reactive, 4
region, 52
RELATION, 59
relation, 47
release constraint, 107
rendezvous, 97
request automaton, 112
requirements elicitation, 3
response time, 15
restriction, 47
robot control system, 71
robot job, 90
robot task, 77
safety property, 38, 40
safety-critical system, 1
schedule clause, 36
schedule constraint, 61, 65, 99
scheduler automaton, 112
scheduling automaton, 108
SCICOS, 18
Scilab, 22
scope, 57
self, 35
sequence, 31, 47
set, 31
shared variable, 39
simulation, 90
snapshot, 32
software development model, 30
software requirements specification, 1, 29
source state, 47
spiral model, 29
stability, 13
stable, 15, 68
start, 35
state, 47
state sequence, 32, 46
state trajectory, 47, 48
state transition, 53
state variable, 34
state variable set, 47
state-space form, 50
Statecharts, 33
steady state, 13
step, 53
structure, 57
structured method, 4
summing junction, 11
synchronized execution, 54
synchronous scheduling, 56
syntax diagram, 121
system perspective, 2
take-off, 11
target state, 47
task, 97
tasking, 97
testing, 29
Time, 34
time step, 53
time-invariant linear system, 13
timed automaton, 113
timing analysis, 96
torque, 12
transfer function, 131, 132
transform, 10
transition clause, 35
transition step, 53, 55
translational semantics, 60
typing, 58

Uppaal, 113
urgency semantics, 55
urgent channel, 169
user-defined type, 35

valuation, 47
variable clause, 34

waterfall model, 29

XTG, 165, 168

z-transform, 16
z-transformation, 15
zero-order-hold, 17
ZOH-block, 17, 19, 81
Bibliography


